LEIAME - EP3

Disciplina: Programação para Redes - MAC 448/5910 / IME-USP

Professor: Daniel Macêdo Batista

1 Integrantes

Nome	**NUSP**
Carlos Eduardo Leão Elmadjian	5685741
Renan Fichberg	7991131

2 Arquivos

O diretório **ep3-carlos-renan** deve conter os seguintes arquivos:

2.1 Arquivos relativos ao simulador

1. **AgentEnum.java** – Enum com os tipos de agentes da rede.
2. **Agent.java** – classe abstrata que representa um agente da rede.
3. **Clock.java** – classe responsável pelo tempo de execução.
4. **DNSServer.java** – classe que representa um servidor DNS.
5. **DuplexLink.java** – classe que representa um enlace da rede do tipo *duplex-link.*
6. **FTPClient.java** – classe que representa um cliente FTP.
7. **FTPServer.java** – classe que representa um servidor FTP.
8. **Host.java** – classe que representa um computador da rede.
9. **HTTPClient.java** – classe que representa um cliente HTTP.
10. **HTTPServer.java** – classe que representa um servidor HTTP.
11. **InputReader.java** – classe responsável por manipular os arquivos de entrada do simulador e interpretá-los.
12. **Makefile** – instruções de compilação para o *make.*
13. **NetSim.java** – classe que contém o código-fonte do simulador.

14. **Node.java** – classe abstrata que representa um nó. Um nó pode ser tanto um computador (*host*) quanto um roteador (*router*).
15. **Packet.java** – classe que representa um pacote de transmissão, cujos dados se situam na camada de rede.
16. **Router.java** – classe que representa um roteador da rede.
17. **RouterBuffer.java** – classe que representa um buffer do router. Funciona em FIFO.
18. **SimulatorLogger.java** – classe responsável por escrever os arquivos de saída (e criar diretórios com o caminho desejado, caso necessário).
19. **Sniffer.java** – classe que representa um *sniffer*.
20. **TCP.java** – classe que armazena os dados do TCP na camada de transporte.
21. **TransportLayer.java** – classe abstrata que representa uma camada de transporte.
22. **UDP.java** – classe que armazena os dados do UDP na camada de transporte.
23. **README.pdf** – documentação complementar, relatório de implementação e guia geral do programa (este documento).

2.2 Arquivos relativos a testes e apresentação

1. **slides.pdf** – slides de apresentação do trabalho.
2. **testes.txt** – arquivo contendo as amostras individuais dos testes pedidos, com cálculo da média, desvio padrão e intervalo de confiança em 95%.
3. **/inputs/cenario_simple.txt** – arquivo contendo uma configuração simples da rede.
4. **/inputs/cenario_medio.txt** – arquivo contendo uma configuração para o simulador um pouco mais densa que a anterior.
5. **/inputs/cenario_complexo.txt** – arquivo com a configuração mais complexa para o simulador.
6. **/inputs/ftp_example.txt** – configuração de cenário para uso do protocolo FTP.
7. **file.txt** – arquivo usado nas transferências.
8. **graficos.tar.gz** – arquivo contendo seis gráficos criados a partir dos dados coletados.

3 Compilação e execução

Para compilar, você pode utilizar o programa *make* sem nenhum parâmetro:

```
$make
```

Para executar o programa do simulador gerado na compilação, você deve passar como parâmetro o nome do arquivo que contém os dados de entrada, como no exemplo abaixo:

```
$java NetSim input_example.txt
```

Este arquivo de entrada deve estar **obrigatoriamente** dentro do subdiretório **inputs**.

4 Relatório do projeto

Escolhemos implementar esse projeto em Java em função da vasta documentação disponível, uma ampla comunidade de apoio, a oferta de bibliotecas específicas para nossos propósitos, IDEs dedicadas, como o Eclipse, e familiarização. Todos estes fatores facilitaram o desenvolvimento. Em particular, utilizamos intensivamente a troca de mensagens entre objetos e referências para montarmos nossa arquitetura de rede.

4.1 Simulador

A implementação do simulador foi pensada considerando o fato de que o programa deveria saber como lidar com 6 protocolos distintos em 3 camadas da rede diferentes. Optamos por fazer a intermediação dos dados da entrada com um *parser* (**InputReader.java**) capaz de armazenar e interpretar os dados lidos. Essa camada entre os recursos da entrada e os objetos que desejam acessá-los proporcionou uma maior organização ao código.

A estratégia usada para fazer a leitura dos dados do arquivo de entrada foi com o uso de expressões regulares. É criado um vetor com todos os padrões que são esperados pelo simulador e, conforme acontece a leitura de uma nova linha no arquivo de entrada, o programa vê se aquela linha se enquadra em algum dos padrões esperados. É importante ressaltar que delegamos ao usuário a responsabilidade de montar uma entrada adequada. A execução prossegue ininterrupta mesmo que seja lido algo não esperado. O arquivo de entrada deve ter extensão **.txt**, **obrigatoriamente**.

Quanto às saídas, o simulador está preparado para receber qualquer nome de arquivo, com ou sem diretórios. Se o usuário quiser criar um arquivo dentro de 3 diretórios, o programa se encarregará disso. Esses diretórios serão subdiretórios do arquivo que contém o código-fonte.

O programa não faz verificação para ver se já existe um arquivo com o nome pedido e muito menos checa se 2 *sniffers* estão escrevendo no mesmo arquivo de saída (log). Mais uma vez, fica ao cargo do usuário não fornecer entradas que se enquadrem nestas categorias. Se nenhum rota for passada, isto é, se o programa ler no arquivo de entrada, na REGEXP específica, uma string vazia (“ ”), então o log deste sniffer será criado por default em um subdiretório chamado **logs**, e o arquivo se chamará **nome_do_sniffer.log**.

Além disso, ponderamos se deveríamos aceitar entradas com disparos simultâneos para o mesmo agente e concluímos que conceitualmente isso seria incoerente, pois mesmo que divisássemos uma forma de armazenar disparos de ação simultâneos (com uma fila, por exemplo), ainda assim essas ações seriam executadas *sequencialmente*. Portanto, em configurações de simulação com disparos simultâneos para o mesmo agente, apenas a última delas irá ocorrer.

Com relação aos cabeçalhos, foi considerado ainda que não há opções, portanto, os valores usados são 20 bytes no cabeçalho IP (considerando que é IPv4), 20 bytes no cabeçalho TCP e 8 bytes no cabeçalho UDP.

Neste programa, presumimos que todas as transferências UDP possuem menos que o tamanho MSS padrão de 1460 bytes definidos para o EP. Para o caso do TCP, foi implementado um algoritmo de fragmentação de pacotes e remontagem. O envio de pacotes fragmentados foi feito utilizando um controle de congestionamento com *slow start* e uma janela de envio de crescimento exponencial.

Por fim, embora o programa do simulador não seja uma unidade central controladora das entidades da rede, ela é, todavia, responsável por delegar os comandos da aplicação dos quais cada agente ficará incumbido. Os *hosts* e *routers*, entretanto, são completamente independentes (foram criados como *threads*)

4.2 Experimentos

Os experimentos foram realizados em uma máquina com processador Intel(R) Core(TM) i5-2500K, de 3.30GHz, 8GB de RAM, sistema operacional GNU/Linux (distro Arch) 64-bit, *kernel version* 3.17.2-1.

Três cenários de configuração foram utilizados para os testes. No cenário simples (**cenario_simples.txt**), o esquema de distribuição dos nós e endereços é basicamente idêntico ao exemplo do enunciado. Algumas alterações foram empregadas somente no programa de disparos do simulador.

Para um cenário médio, criamos uma rede com oito hosts e quatro roteadores centralizados (sem outras redes dentro de sub-redes). Além disso, criamos tabelas de roteamento grandes para verificar se o experimento provocaria uma demanda maior de processamento.

Para o cenário complexo, utilizamos um arquivo de configuração disponível em *https://github.com/dacortez/* Fizemos pequenas modificações nessa configuração para adaptá-la ao nosso programa. Trata-se de um modelo mais complexo, com sub-redes dentro de sub-redes, dois roteadores centrais (r2 e r3), seis ao todo, além de 12 hosts.

Como o enunciado do EP3 exigia que as transferências entre hosts demandassem arquivos de pelo menos 500KB, optamos por definir que todas as transferências seriam de um mesmo arquivo **(file.txt)**. Isso acarretou uma lentidão, talvez inesperada, mas justificável visto os mecanismos de *timeouts*, controle de congestionamento e fragmentação de pacotes.

Os experimentos revelaram que o uso da CPU é praticamente constante dentro do patamar de uso de cada cenário. Observou-se, contudo, um ligeiro incremento da variância no tempo de relógio conforme o grau de complexidade dos cenários ia progredindo. O uso de CPU também se mostrou mais intenso no cenário intermediário e menos intenso no cenário complexo. A explicação para isso está no fato de que a estrutura de rede construída para o caso médio foi menos distribuída e com vários disparos próximos, enquanto que no caso complexo ocorre o oposto.

Em todos os casos, foi utilizado para fins de medição o programa de console GNU Time. Trata-se de um aplicativo mais sofisticado do que o comando *time* do Bash, já que aquele é capaz de medir não só o tempo de execução de um programa como também seu consumo de processamento (em porcentagem).

5 Guia de implementação

Nesta seção, iremos discutir brevemente a implementação do código por meio das classes em Java e seus principais métodos. Informações mais detalhadas estão contidas nos comentários presentes no código-fonte.

5.1 Agent.java

- É uma classe abstrata que representa um agente da rede. Os agentes da rede são os presentes em **AgentEnum.java**. Ela possui as assinaturas dos métodos usados por um agente.

5.2 Clock.java

Apesar de no final nem todos os métodos de Clock serem usados, eles foram implementados para caso houvesse a necessidade de ter mais ou menos precisão na contabilidade. O último

método abaixo, que também não é usado, foi idealizado para caso tivesse uma simulação muito longa, mostrasse o tempo em um formato mais legível para humanos.

- **execution_time_in_nanos()** - Retorna o tempo de execução em nanossegundos.
- **execution_time_in_milis()** - Retorna o tempo de execução em milissegundos.
- **execution_time_in_seconds()** - Retorna o tempo de execução em segundos.
- **hms_format(long)** - Retorna um string no formato horas/minutos/segundos.

5.3 DNSServer.java

- **notify_agent(Packet)** - Alerta recebimento de pacote.
- **process_DNS_query(ApplicationLayer, String)** - Processa uma requisição DNS e responde com um pacote que contém o endereço IP.

5.4 DuplexLink.java

Esta classe representa um *duplex-link* da rede, isto é, um canal de comunicação de duas vias independentes em sentidos contrários.

- **set_link(String, String)** - Conecta-se aos extremos do enlace.
- **has_edges(String, String)** - Checa se o link é formado pelos dois nós recebidos (em String).
- **get_Node(String)** - Retorna o nó associado a um nome específico.
- **is_host_point(String)** - Verifica se o nó é um *host*.
- **public boolean is_router_point(String)** - Verifica se o nó é um *router*.
- **void forward_packet(Node, Packet)** - Passa para frente o pacote de um nó para o próximo.

5.5 FTPClient.java

- **receive_command(String)** - Recebe um comando.
- **Packet build_packet(String, String)** - Cria um pacote com dados da camada de aplicação.
- **process_command(String)** - Processa um comando recebido do simulador e o transforma em um pacote.

5.6 FTPServer.java

- **process_packet(Packet)** - Processa um pacote recebido de um *host*.
- **process_FTP_request(String)** - Interpreta uma requisição FTP e a responde.
- **read_file(String)** - Lê um arquivo do servidor.
- **FTP_response(String)** - Cria o cabeçalho FTP de acordo com os dados requisitados.

5.7 Host.java

Esta é uma das principais classes do simulador. O Host abriga métodos responsáveis pela manipulação de todo tipo de informação atrelada à camada de transporte. Além disso, todo Host faz a intermediação entre o enlace e o agente de aplicação que está instalado na máquina.

- **run()** - Executa uma instância do tipo *Host*.
- **send_packet(Packet)** - Envia pacote pelo enlace do host.
- **receive_packet(DuplexLink, Packet)** - Recebe pacote pelo enlace do host.
- **open_connection(Packet)** - Inicia uma conexão TCP (faz o *3-way-handshake*).
- **build_raw_TCP_packet(Packet)** - Constrói pacote TCP sem a camada de aplicação.
- **chop_data(Packet, int)** - Fragmenta pacotes de para um MSS de 1460 bytes.
- **send_TCP_packet(Packet)** - Envia um pacote TCP de acordo com a política de controle de congestionamento.
- **got_ACK(int)** - Verifica se algum pacote da fila tem o ACK esperado.
- **send_with_congestion_control(Packet[])** - Política de controle de congestionamento.
- **reply_if_isACK(Packet)** - Se um pacote TCP tiver bit ACK ligado, responde host acusando recebimento.
- **reply_if_isSYN(Packet)** - Se um pacote estiver com o bit SYN ligado, responde o host para fazer handshake.
- **assembly_packet(Packet)** - Monta um pacote que chegou fragmentado.
- **DNS_lookup(String)** - Faz requisição do endereço IP de um host.
- **DNS_resolve()** - Extrai o endereço devolvido por um servidor DNS.

- **build_UDP_packet(Packet)** - Recebe um pacote com a camada de aplicação e insere uma camada UDP.
- **send_UDP_packet(Packet)** - Envia um pacote UDP.

5.8 HTTPClient.java

- **receive_command(String)** - Recebe um comando.
- **notify_agent(Packet)** - Alerta recebimento de pacote.
- **build_packet(String, String)** - Cria um pacote com dados da camada de aplicação.
- **process_command(String)** - Processa um comando recebido do simulador e transforma-o num pacote.

5.9 HTTPServer.java

- **notify_agent(Packet)** - Alerta recebimento de pacote.
- **process_packet(Packet)** - Processa um pacote recebido de um *host*.
- **process_HTTP_request(String)** - Interpreta uma requisição HTTP e a responde.
- **read_file(String)** - Lê um arquivo do servidor.
- **HTTP_response(String)** - Cria o cabeçalho HTTP de acordo com os dados requisitados.

5.10 InputReader.java

- **read_input(String)** - Leitor de entrada. Obter os dados para gerar simulação.
- **parse_line(String)** - Faz o parsing individual de cada linha e encaminha para a função apropriada.
- **set_host(String)** - Cria uma instância nova de um *host*.
- **set_router(String, String)** - Cria uma instância nova de um *router*.
- **set_duplex_link(String, String, String, String)** - Cria uma instância nova de um enlace do tipo *duplex-link*.
- **configure_host(String, String, String, String)** - Configura um *host* com IP, *default gateway* e DNS.

- **configure_router(String, String)** - Configura um *router* associando um IP a uma interface do enlace.
- **configure_router_route(String, String)** - Define as rotas do roteador.
- **configure_router_specs(String, String, String)** - Define as especificações de desempenho do roteador.
- **set_agent(String, String)** - Cria instância de um agente da rede (aplicação ou *sniffer*).
- **attach_app_agent(String, String)** - Associa um agente de aplicação a um *host*.
- **attach_sniffer_agent(String, String, String, String)** - Associa um agente *sniffer* a um enlace.
- **set_simulation(String, String)** - Define o programa principal.

5.11 NetSim.java

- **main(String[])** - Função principal.
- **schedule(InputReader, Timer)** - Agendador de disparos de comandos para os agentes.
- **schedule_finish(Long, Timer)** - Agenda o fim do programa, com um *timer* e um disparo em segundos.
- **start_countdown()** - Imprime a contagem regressiva para inicialização do simulador de forma bonitinha. :3
- **void init_all_nodes(InputReader)** - Roda todos os *hosts* e *routers* da simulação.
- **boolean check_arguments(String[])** - Checa argumentos de entrada.
- **invalid_argument_notification()** - Notifica erro com relação aos argumentos da linha de comando e termina execução do programa.

5.12 Node.java

- É uma classe abstrata que representa um nó da rede. Ela tem as assinaturas de métodos comuns às entidades que são nós (*hosts* e *routers*).

5.13 Packet.java

- **decrease_ttl()** - Diminui o TTL do pacote.
- **clone_to_packet(Packet)** - Clona um pacote.

5.14 Router.java

- **void run()** - Executa uma instância do tipo *Router*.
- **get_link(int)** - Retorna o enlace associado à porta.
- **have_port(int)** - Checa se a porta existe no atributo de portas e IP.
- **have_ip(String)** - Checa se o IP existe no atributo de portas e strings.
- **retrieve_associated_ip(int)** - Busca o IP associado à porta.
- **retrieve_associated_port(String)** - Busca a porta associado ao IP.
- **update_routing()** - Atualiza a tabela de roteamento para eliminar valores que correspondem a IPs.
- **receive_packet(DuplexLink, Packet)** - Recebe pacote de um enlace.
- **send_packet(Packet, int)** - Roteia o pacote de uma interface para outra.
- **route_from_packet(Packet)** - Define a interface para onde deve ser enviado o pacote.
- **process_package()** - Processa todos os pacotes contidos nos *buffers*.

5.15 RouterBuffer.java

- **put_packet(Packet)** - Adiciona o pacote à fila.
- **pull_packet()** - Remove um pacote da fila.
- **is_full()** - Verifica se o *buffer* está cheio.
- **is_empty()** - Verifica se o buffer está vazio.
- **show_packets()** - Exibe todos os pacotes que estão na fila.

5.16 SimulatorLogger.java

- **set_log_file_name(String)** - Cria o log.
- **split_file_name(String)** - Trabalha o PATH, vendo os sub-diretórios que devem ser criados e o nome do arquivo para escrever a saída.
- **have_dir(String)** - Checa se precisa criar sub-diretórios de acordo com os dados de entrada.
- **write_to_log(Clock, Packet)** - Escreve no log do *sniffer*.
- **build_message(Clock, Packet)** - Constrói a mensagem que deve ser escrita no arquivo de saída e imprimida no *prompt*.

5.17 Sniffer.java

- **set_residence(Object)** - Define onde a aplicação está localizada.
- **write_capture(Packet)** - Invoca a função de escrita do log para escrever as informações da captura do pacote.,

5.18 TCP.java

- **clone_to(TransportLayer)** - Clona um objeto TCP.

5.19 TransportLayer.java

- É uma classe abstrata que representa uma camada de transporte. Ela tem as assinaturas dos métodos usados por uma camada de transporte e alguns atributos comuns.

5.20 UDP.java

- Basicamente *setters* e *getters*.