Trabalho Prático 2

João Cunha, Hugo Cardoso, and Válter Carvalho
(Grupo 2, PL4)

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a85006,a84775,a84464}@alunos.uminho.pt

Resumo Criação de uma rede de gateways que ficarão à escuta numa porta, terão um servidor alvo e utilizam um peer para enviar pacotes para esse servidor. Criação de um novo PDU para que se consiga realizar este processo. Utilização de sockets TCP e UDP.

1 Introdução

O objetivo principal deste trabalho prático é criar uma rede de entidades que abstraem a conexão a um servidor alvo, de modo a evitar o *sniffing* de pacotes que possam causar divulgações de identidade do utilizador que envia o pedido. São chamadas essas *gateways* "AnonGW", que são inicialmente configurados com o servidor que têm de proteger e a porta que ficarão à escuta para novos clientes que pretendam estabelecer conexões a esse servidor.

2 Arquitetura da Solução

Como explorado inicialmente, o grupo acabou por utilizar **Java 8** porque satisfaz todos os requisitos necessários à realização deste projeto: tem uma enorme biblioteca de *sockets*, métodos para ler dados deles, criação de pacotes UDP, criação de ligações TCP e uma forte componente de facilidade de utilização, por causa de toda a abstração que a linguagem nativamente fornece.

Quanto ao código fonte em si, o grupo estabeleceu as seguintes classes:

- **AnonGW**: classe principal onde inicializa todo o processo. São criadas duas *threads*: **TCPWorker** e **UDPWorker**.
- **UDPWorker**: é uma *thread* que fica à escuta na porta 6666 por pacotes UDP, dependendo do pacote que recebe cria uma nova *thread*. Se o pacote for do tipo "pedido"(na direção cliente->servidor) a *thread* criada será **ServerReaderWriter**; se for do tipo "resposta"(na direção servidor->cliente) a *thread* será **ClienteWriter**; se for do tipo "ack", apenas adiciona à **Tabela** a confirmação que o pacote chegou ao *gateway* para o cliente respetivo.
- **TCPWorker**: é uma *thread* que fica à escuta na porta dada por argumento por conexões TCP, responsável pela multiplexagem dos clientes. Por cada nova conexão, lança uma nova *thread* que tratará de cada cliente individualmente, **ClienteReader**.
- **Tabela**: Tem guardado o estado atual da troca de pacotes. Sempre que se recebe um *ack*, se for de "resposta"é adicionado à tabela de resposta, se for de "pedido", é adicionado à tabela de pedidos. Para além das duas tabelas, estabelece uma ligação IDCliente->Socket para se estabelecer a conexão Gateway->Cliente.
- **ServerReaderWriter**: envia o pacote atual ao servidor recebido no **UDPWorker**, se for o último estabelece conexão TCP ao servidor e cria os pacotes de resposta, enviando-os na sua totalidade ao peer anterior por UDP, removendo o cliente desses pacotes da tabela de respostas. Envia, também, um *ack* ao peer anterior a indicar que recebeu o pacote.

- **Pacote**: é o pacote trocado por UDP entre os *gateways*, a ser explorado mais tarde, na secção 3.1.
- **Encriptacao**: realiza a encriptação do pacote UDP, utilizando uma palavra-chave comum a todos os *gateways*. Realiza, também, o processo inverso de desencriptação do mesmo pacote.
- **ClienteWriter**: envia ao cliente a resposta ao pedido inicial por TCP, enviando ao peer anterior um *ack*, removendo a entrada relativa ao cliente da tabela de pedidos caso seja o último pacote.
- **ClienteReader**: *thread* que efetivamente realiza a leitura de pacotes TCP do cliente, criando o pacote UDP respetivo e enviando ao *peer* escolhido por *round-robbin*.
- **Arguments**: classe auxiliar que guarda os diversos campos recebidos por argumento no *gateway*, isto é: *peers*, *password*, *keys* (a ver na secção 4.2), *port*, *target server*.

3 Especificação do protocolo UDP

3.1 Formato do PDU

Para garantirmos que os pacotes fazem efetivamente a ponte lógica entre os *peers*, tivemos de desenvolver o nosso próprio pacote para transmitir via UDP. Para além dos cabeçalhos habituais UDP, os nossos pacotes são constituídos por:

- **Tipo**: ocupa no total 1 byte e indica se for um pacote do tipo resposta (1) ou do tipo pedido (0).
- **Ack**: ocupa no total 1 byte e indica se for um pacote do tipo *ack* (valor 1), se não for tem valor 0. Serve apenas para controlos de informação no UDP, porque o protocolo em si não garante que os pacotes chegam ao destino.
- **Último**: ocupa no total 1 byte e indica se for o último pacote para um dado cliente/servidor (será 0 se não for o último e 1 se for).
- **ID do pacote**: ocupa 4 bytes e indica o ID do pacote. É utilizado juntamente com os *acks* para saber quando os pacotes chegam e por que ordem o fazem.
- **Alvo**: ocupa 4 bytes no total e indica o IP para onde se pretende que o pacote seja enviado via TCP.
- **Peer**: ocupa 4 bytes e indica o IP do *peer* que enviou o pacote, para no *peer* que recebe a mensagem ser possível reenviar de volta ao *peer* que enviou originalmente.
- **ID do cliente**: ocupa 4 bytes e indica o ID do cliente, não é enviado o IP para garantir a segurança do cliente, visto que vai contra o propósito deste protocolo enviar algo tão caracterizador como o IP do cliente, o do **Alvo** é uma exceção porque é conhecido a toda a gente, não é necessário esconder.
- **Porta**: ocupa 4 bytes e indica a porta que será utilizada para comunicar com o servidor alvo.
- **Tamanho**: ocupa 4 bytes e indica o tamanho total da parte de dados do pacote, visto que enviamos tamanhos fixos de pacotes o último pacote pode não ter esse tamanho na totalidade, logo é necessário este campo.
- **Data**: ocupa os restantes bytes do pacote e indica os dados a enviar para o cliente/servidor.

Como se pode verificar, o *overhead* dado por este protocolo é de **27 bytes** por pacote. Por defeito, os pacotes estão programados a terem o tamanho fixo de **1027 bytes**, logo, por TCP são lidos sempre **1000 bytes** no máximo e adicionados como campos de "Data"no pacote, tanto do cliente como servidor e são acrescentados estes novos cabeçalhos do nosso PDU, para além dos do UDP como já foi comentado.

3.2 Interações

Com a definição anterior do pacote, foi possivel implementar as seguintes funcionalidades: **Entrega Ordenada**, **Encriptação e Confidencialidade**, **Multiplexagem**.

As interações existentes entre os peers acontecem quando são recebidos os seguintes pacotes com as seguintes características:

- **Pedido**: o pacote é enviado para o próximo *peer*, como visto acima, com o byte do tipo a 0. Isso significa que no **UDPWorker**, vai ser tratado este pacote como sendo um pacote que é necessário enviar ao servidor, criando para isso a *thread* **ServerReaderWriter**.
- **Resposta**: o pacote é enviado para o próximo *peer*, como visto acima, com o byte do tipo a 1. Isso significa que no **UDPWorker**, vai ser tratado este pacote como sendo um pacote que é necessário enviar ao cliente, criando para isso a *thread* **ClienteWriter**.
- ***ACK* de pedido**: significa que o *peer* recebeu com sucesso um pacote com o byte do tipo a 0, ou seja, confirma-se que foi recebido aquele pedido, atualizando a tabela com o último pedido recebido de um dado cliente.
- ***ACK* de resposta**: significa que o *peer* recebeu com sucesso um pacote com o byte do tipo a 1, ou seja, confirma-se que foi recebido aquela resposta, atualizando a tabela com a última resposta recebido do *peer* que comunica com o servidor para um dado cliente.
- **Último**: significa que o pacote é o último pacote de um dado tipo (seja ele resposta ou pedido) e, portanto, corresponde à ação de eliminar a entrada do cliente da tabela dos pedido, caso seja um pedido, ou da de respostas, caso seja uma resposta.

4 Implementação

4.1 Entrega Ordenada

A entrega ordenada é garantida porque é implementado no **ClienteReader** e no **ServerReaderWriter** um sistema de esperas que garante que os pacotes chegam na ordem com que foram lidos do cliente/servidor. Em antemão, todos os pacotes lidos são numerados por um ID único e esse ID é incremental, ou seja, dois pacotes seguidos diferem em 1 no seu ID.

Coloque-se a situação hipotética: o último pacote confirmado pelo *ACK* é o pacote com ID 4, do tipo pedido. No **ServerReaderWriter**, este bloqueará até confirmar que na tabela de respostas o ID também é 4. Esta interação simboliza que o *peer* que enviou o pacote do tipo pedido ainda não recebeu o pacote com ID 4, caso permaneça bloqueado. Mal receba a confirmação da resposta, avança com o envio de outra.

A título de exemplo coloque-se agora outra situação hipotética: o último pacote confirmado pelo *ACK* é o pacote com ID 1, do tipo resposta. No **ClienteReader**, este bloqueará até confirmar que na tabela de pedidos o ID também é 1. Isto representa que o *peer* que utilizou como intermediário para o servidor (e portanto envia pacotes do tipo resposta) ainda não confirmou o pacote de pedido, caso permaneça bloqueado. Mal receba a confirmação do pedido, avança com o envio de outro.

4.2 Encriptação

O método que foi utilizado para encriptar os nossos dados foi um de chave simétrica. Esta encriptação utiliza 3 parâmetros, dados por argumento, que são a *password*, *KEY1* e *KEY2*. Há duas operações que são relevantes: a encriptação e a desencriptação de um conjunto de bytes.

Começando pela encriptação:

1. A *password* com K bytes é repetida até se obter uma nova *password* com N bytes, que é o tamanho do *array* de bytes que se pretende encriptar. Por exemplo, para a *"abc"*, e para um *array* com 4 bytes, o resultado final será *"abca"*.
2. A *password* é rodada KEY1 vezes para a direita (pensar na *password* como um array circular, rodar 1 vez significa que todos os bytes sobem um indice, o último passa para o início).
3. O *array* de bytes é rodado KEY2 vezes para a esquerda.
4. É realizado um XOR entre a *password* e o *array* de bytes.
5. O *array* resultante do XOR é submetido a mais uma rotação de KEY1+KEY2 vezes, para a direita.

A desencriptação faz a operação inversa à encriptação, porém os primeiros passos são exatamente iguais:

1. A *password* com K bytes é repetida até se obter uma nova *password* com N bytes, que é o tamanho do *array* de bytes que se pretende encriptar. Por exemplo, para a *"abc"*, e para um *array* com 4 bytes, o resultado final será *"abca"*.
2. A *password* é rodada KEY1 vezes para a direita.
3. O *array* é submetido a uma rotação -(KEY1+KEY2) vezes, ou seja, KEY1+KEY2 vezes para a esquerda.
4. É realizado um XOR entre a *password* e o *array* de bytes.
5. O *array* resultante do XOR é rodado -KEY2 vezes, ou seja, KEY2 vezes para a direita.

Estes passos garantem que os dados são encriptados durante toda a sua trajetória, sendo apenas desencriptados quando se necessita de ler o seu conteúdo. Um intruso que esteja à escuta apanhará apenas bytes sem sentido, pelo que garantimos a segurança de comunicação.

Parte-se do pressuposto que todos os *peers* partilham a mesma *password*, *KEY1* e *KEY2*, caso contrário este método não funciona.

4.3 Multiplexagem

Como foi mencionado anteriormente, o programa funciona muito à base de *threads*, em virtude de serem potencialmente mais que um cliente a conectar-se a um determinado *peer*. Para além disto, são multiplexados também os canais de leitura de pacotes UDP e TCP porque têm de estar simultaneamente a lidar com tráfego destes dois protocolos, assim como os extremos de escrita e leitura nos dois sentidos (cliente para o servidor e servidor para o cliente).

Há uma situação em particular que merece atenção em específico, que é a parte de enviar novamente o pacote de volta para o cliente num dado *peer*. Como o canal TCP e o canal UDP estão completamente separados em *threads* diferentes, teve de se arranjar maneira de criar um *pipe* de ligação entre estes dois porque como há mais que uma conexão aos clientes, nunca se sabe ao certo a qual o cliente a enviar de volta esse pacote de resposta.

A solução foi a criação de uma tabela de endereçamento, que guarda como chave o ID do cliente e cujo valor é o seu *Socket*, nomeadamente para abrir o seu extremo de escrita e, por fim, enviar a parte dos dados do pacote UDP. Em todas as iterações (ler um pacote de cada vez do tipo UDP), é aberto e fechado o socket cujo ID é o ID do cliente que vem no pacote, lidando assim com este problema.

5 Testes e Resultados

Para se compilar o programa, foi incluido um Makefile que possui todas os comandos necessários para a compilação:

1. javac AnonGW.java

2. rm -R ./exec/* (para caso já exista limpar o que já foi compilado)
3. mkdir -p ./exec
4. mv *.class ./exec

Salienta-se que é necessário garantir que há pelo menos o Java 8 instalado na máquina para se poder utilizar.

De seguida, pode-se então executar o programa (após mudar a diretoria para o /exec/ da pasta onde está gravado o código fonte):

– java AnonGW -target <IP> -port <INT> -peers <IP1> <IP2> ... <IPn> -password <STRING> -keys <INT> <INT>

A título de exemplo para testes, vamos considerar que existem os seguintes *peers* com a seguinte informação, no ambiente da topologia Core fornecida pela equipa docente:

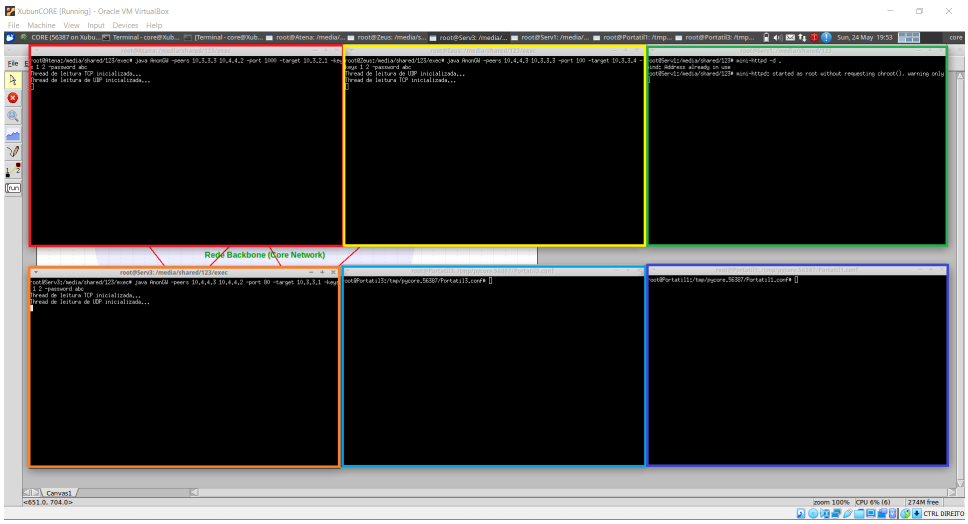
1. **Peer 1**: java AnonGW -target 10.3.2.1 -port 1000 -peers 10.3.3.3 10.4.4.2 -password abc -keys 1 2
2. **Peer 2**: java AnonGW -target 10.3.3.4 -port 100 -peers 10.4.4.3 10.3.3.3 -password abc -keys 1 2
3. **Peer 2**: java AnonGW -target 10.3.3.1 -port 80 -peers 10.4.4.3 10.4.4.2 -password abc -keys 1 2

Considere-se também que existem 3 clientes a fazerem pedidos para o **Peer3** utilizando o comando **wget** para 1 ficheiro de texto pequeno (cabe em 1 pacote), 1 ficheiro de mp3 (bastante grande, exige bastantes pacotes) e 1 ficheiro PDF (grande mas não exige tantos como o de mp3) disponibilizados no servidor alvo do **Peer2**.

Considere-se, ainda, que o servidor 10.3.3.1 é um servidor de HTTP, usando o comando **mini-httpd**.

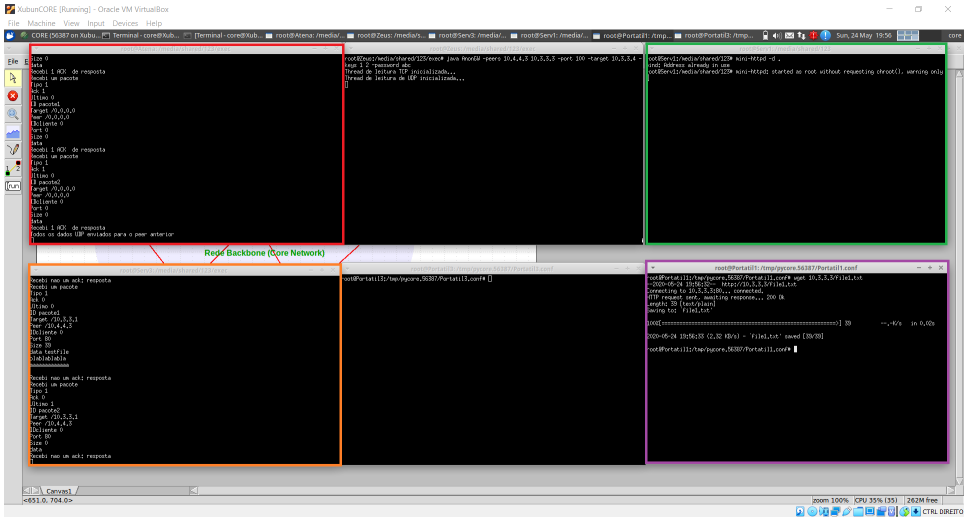
Serão utilizadas as cores para sinalizar as interações de interesse entre os vários testes.

Inicialmente, tem-se a seguinte situação:



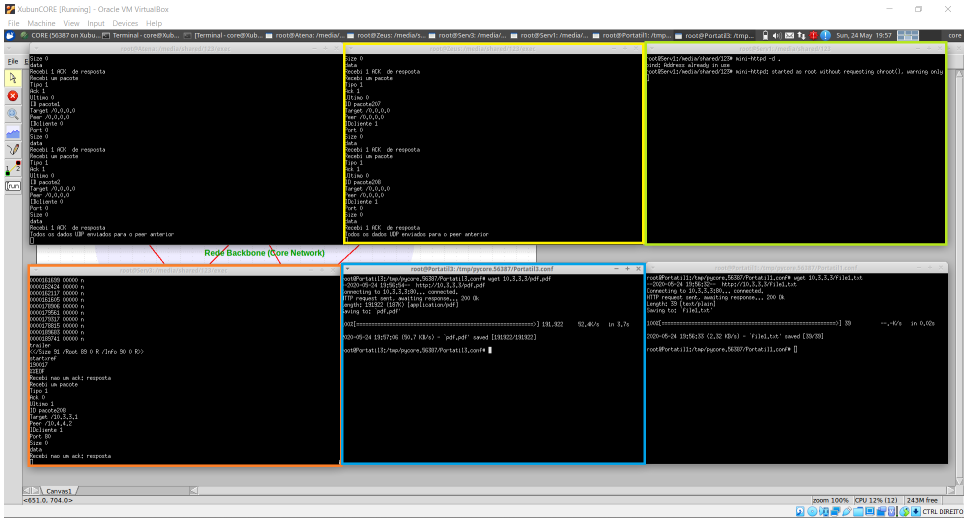
Os *peers* (vermelho é o **Peer1**, amarelo é o **Peer2**, laranja é o **Peer3**) estão à espera de novos clientes (terminais azul e roxo). Verificamos, como dito anteriormente, que o servidor (verde) é o alvo do **Peer3**.

O primeiro cenário de teste é um só cliente fazer um pedido de um ficheiro muito pequeno, o **file1.txt**, que é o que se vê na imagem que se segue:



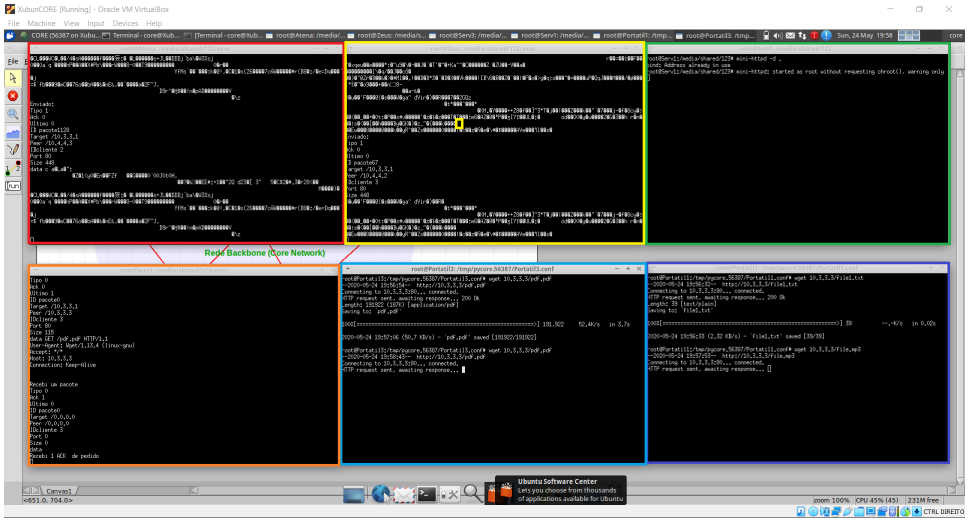
Verificamos que o **Peer3** escolheu como seu *peer* o **Peer1** e o cliente recebeu o seu ficheiro.

Passando agora para outro cliente, este decide fazer agora o *download* de um ficheiro pdf, já relativamente grande e que ocupa mais que um pacote para ser enviado:



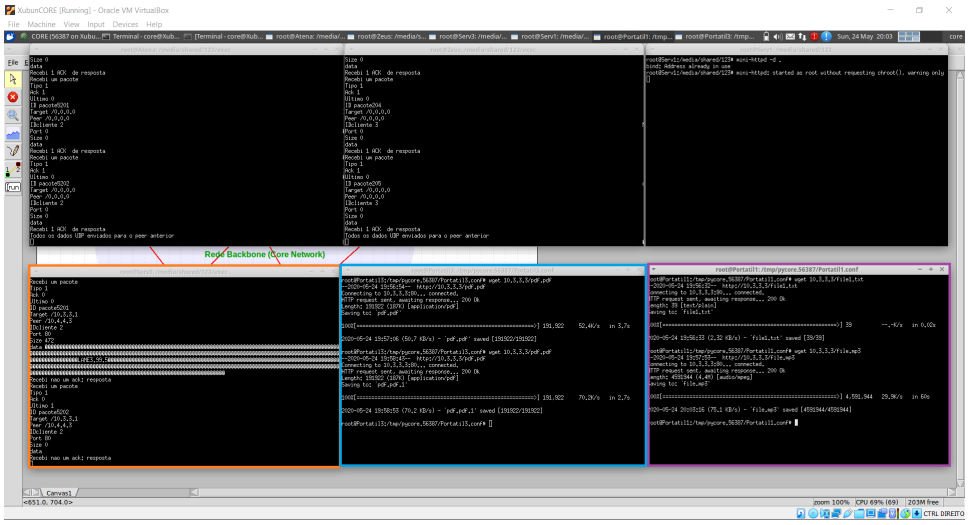
Como podemos verificar, o **Peer3** escolheu agora como seu *peer* o **Peer2** e o cliente recebeu o PDF na totalidade, após a troca de milhares de pacotes.

Passando agora para um teste da multiplexagem, serão feitos os *downloads* de dois ficheiros enormes por dois novos clientes, um pedirá um pdf e o outro um ficheiro mp3:



Reparamos (pelas impressões na consola) que o **Peer1** está a receber o ficheiro mp3 e o **Peer2** está a receber o ficheiro pdf, a pedido do **Peer3** (o terminal roxo é o cliente com ID 2 e o terminal azul é o cliente com ID 3).

Por fim, o **Peer3** reenvia os pacotes para o cliente respetivo e as transferências terminam:



Como as transferências terminaram, podemos considerar que os testes foram um sucesso.

6 Conclusão

O grupo considera que no geral este trabalho foi uma mais valia para a nossa formação como engenheiros porque essencialmente tivemos de re-implementar o protocolo TCP usando UDP, o que nos elucidou bastante sobre o funcionamento e *shortcomings* de ambos.

O facto de ser necessário ter em conta que há uma extra camada de segurança não facilitou o projeto devido à troca de *acks*, mas o grupo conseguiu superar isso e criar um projeto estável com multiplexagem, garantia de ordem de entrega e completamente cifrado, no entanto, é melhorável nomeadamente com controlo de congestão (que não era um requisito desta fase).

Em suma, este 2º trabalho foi bastante interessante e mostrou o quão poderosos são estes dois protocolos (TCP e UDP) e a sua utilidade individual.