Relatório TP2

Luis Duarte, Ulisses Araújo, Paulo Lima

A81931
A84010
A89983

Resumo. Neste trabalho foi criado um programa que permite implementar uma rede anónima que recebe pedidos de cliente numa porta TCP e reencaminha-os para uma porta UDP de um outro nodo da rede, e cuja função é esconder acessos a um Servidor HTTP.

1 Introdução

O desafio que nos foi apresentado neste trabalho foi de essencialmente criar um túnel UDP "anonimizador", isto é, um túnel por onde seja possível fazer pedidos de transferência a um servidor HTTP e receber as respetivas respostas sem o servidor destino ter qualquer conhecimento de quem realizou o pedido.

Para isto foi então implementado o programa designado por AnonGW que é executado em cada nodo da rede e recebe pedidos de cliente numa conexão TCP, e numa conexão UDP, recebe pedidos de outros nodos da rede. Isto é, o cliente manda um pedido para um dado nodo através de uma conexão TCP, esse nodo reencaminha-o para um outro nodo através de um túnel UDP, e o último nodo reencaminha o mesmo para o servidor destino, através de uma ligação TCP, seguindo o caminho inverso para entregar a resposta ao cliente.

De seguida explicaremos então o procedimento seguido para a implementação deste programa e os respetivos protocolos definidos.

2 Especificação do protocolo UDP

2.1 Formato das mensagens protocolares (PDU)

Neste trabalho foram definidos quatro tipos de PDU's diferentes, e um PDU que serve de base para todos os outros, de forma a atingir um protocolo que cumprisse o proposto. O tamanho máximo de um dos nossos pacotes é então de 4149 bytes, e de seguida explicaremos a estrutura de cada um deles.

PDU Base: Este é o PDU que serve de "molde" para todos os outros, pois todos eles têm estes dados em comum. Este PDU contém então, um byte que indica o tipo de PDU (0 = Pedido de conexão, 1 = Ack para o servidor, 2 = Dados para o servidor, 3= Ack do servidor, 4= Dados do servidor), um inteiro que indica o ID do cliente a que se refere, e uma string que contém o IP do nodo da rede que recebeu esse pedido do cliente.

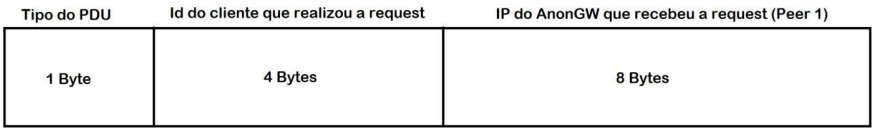

Fig. 1. Estrutura de um PDU base

PDU_ACK: Este PDU, como o nome indica, é o que trata dos ACK's, tendo todos os dados do PDU base, e ainda um byte que indica o seu sub-tipo (0 = recebeu pedido de conexão, 1 = recebeu confirmação do pedido de conexão entregue, 2 = recebeu dados) e por fim, outro inteiro que indica o número do ACK.

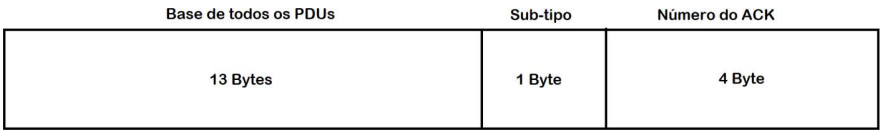

Fig. 2. Estrutura de um PDU ACK

PDU_Data: Este é o PDU que trata do transporte de dados dos pedidos de transferência, e como todos os outros contém o PDU base, mas também contém um inteiro que representa o SEQ, outro inteiro que guarda o número de bytes de dados que o pacote carrega, uma SecretKey que é a chave usada para encriptação do lado do nodo que envia e desencriptação do lado do recetor, e finalmente um array de bytes que contém os dados do pacote.

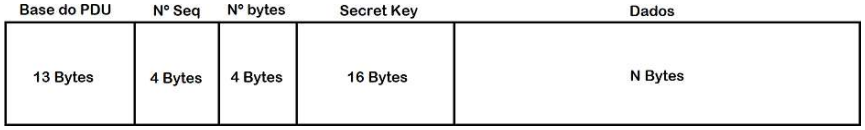

Fig. 3. Estrutura de um PDU data

PDU_Request_Connection: Contém só os dados do PDU base, tendo sido criado só para instanciar de uma maneira diferente em Java.

PDU_Close_Connection: Caso análogo ao PDU_Request_Connection.

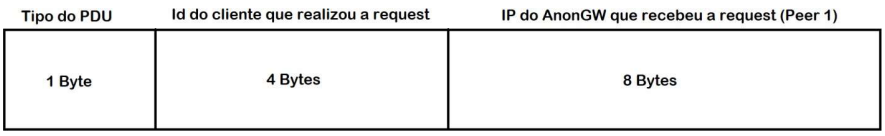

Fig. 4. Estrutura dos PDUs Request_Connection e Close_Connection

2.2 Interações

Existem três fases principais quando um cliente pretende contactar o servidor, e realizar uma transferência, com recurso ao túnel anonimizado desenvolvido, sendo estas: estabelecimento da conexão, troca de dados e, término da conexão.

Na primeira fase, o cliente contacta um primeiro nodo da rede disponível, via uma comunicação TCP. O primeiro nodo reconhece que um novo cliente está a tentar conectar-se e cria um socket TCP entre ambos para troca de informação. Escolhe ainda, de forma aleatória, um outro nodo da rede, e estabelece contacto com este via UDP, enviando-lhe um PDU de pedido de estabelecimento de conexão. Estes serão os dois nodos que permitirão ao cliente comunicar com o servidor sem nenhum dos dois se expor. O segundo nodo, ao receber via UDP um pedido de conexão, reconhece que um nodo está a tentar estabelecer uma conexão com este, e por isso, um novo cliente irá comunicar com o servidor. Desta forma, envia de volta ao primeiro nodo uma confirmação de receção do pedido, ficando há espera que este lhe responda com a última confirmação que garante a confirmação da conexão. O primeiro nodo, ao receber a primeira confirmação, envia então a segunda confirmação, que ao ser recebida pelo segundo nodo, permite terminar a fase de conexão com sucesso. Esta fase é uma replica do "3-way handshake" realizado por conexões TCP.

Na segunda fase, com ambos os nodos já com uma ligação estabelecida, todos os dados que o cliente envia para o primeiro nodo via TCP, são enviados ao segundo nodo pelo primeiro via UDP. O segundo nodo, quando recebe os dados, envia uma mensagem a confirmar o bloco que acabou de receber, também via UDP, ao primeiro nodo. Caso o primeiro nodo não receba a mensagem de confirmação, dentro de um período de tempo pré-estabelecido, reenvia novamente os dados, pois estes podem ter sido perdidos. Os dados partilhados na conexão UDP são devidamente encriptados na origem, e desencriptados no destino, de forma a garantir que um intruso á rede, não os consiga obter e retirar informação útil destes. O segundo nodo, por cada cliente que trate, estabelece um socket TCP com o servidor de destino, de forma a reencaminhar-lhe os dados, obter a sua resposta e reencaminhá-la para o primeiro nodo do túnel, sendo isto feito de forma simétrica ao caminho contrário. O primeiro nodo, ao receber dados de resposta via UDP reencaminha-os para o cliente respetivo.

Na terceira e última fase, quando um cliente não pretende continuar a comunicar com o servidor, a conexão deve ser terminada, e todos os dados referentes a este, apagados de forma imediata. Para isto o primeiro nodo da rede, ao perceber que o cliente se pretende desconectar, envia um pedido de término de conexão ao segundo nodo com o qual se emparelhou para tratar do cliente. O segundo, ao

receber o pedido, envia de volta uma confirmação, elimina todos os dados referentes ao cliente e fecha o socket que tinha estabelecido via TCP com o servidor alvo. O primeiro nodo, ao receber a confirmação, elimina também os dados e fecha ainda o socket TCP estabelecido previamente com o cliente.

3. Implementação

Este trabalho foi realizado recorrendo à linguagem de programação Java, uma vez que era a que os membros do grupo mais se sentiam à vontade. Optou-se pela criação de um package secundário, denominado "PDU_Agent", unicamente para definição protocolar das mensagens trocadas via UDP, como estas podem ser construídas a partir de bytes, desconstruídas em bytes para serem comunicadas e ainda como são encriptados e desencriptados os dados de cada PDU comunicado na rede. Quanto ao mecanismo de encriptação foi usada a biblioteca "javax.crypto" e o algoritmo "AES".

No package primário foram desenvolvidas as classes que tratam da lógica da comunicação, sendo a classe principal, e que irá correr em cada nodo da rede anonimizadora, a classe "AnonGW". Esta classe permite guardar os clientes conectados em cada momento numa tabela de hash, para não existirem repetições e garantir acesso direto e constante por questões de eficiência, e ainda permite ao nodo conhecer todos os outros aos quais se pode conectar para estabelecer um túnel de comunicação. É ainda nesta classe que se implementa a parte de cada nodo da rede possuir uma thread sempre em execução, apenas responsável por escutar pedidos de ligação TCP, ou seja, pedidos de novos clientes, e a cada pedido lançar novas threads, pertencentes á classe "AnonGW_Worker", para tratarem do cliente que acabou de chegar, isto é por exemplo, escutar todo o seu tráfego via TCP e redirecioná-lo para um outro nodo via UDP, e também receber respostas via UDP desse nodo e reencaminhá-las via TCP para o cliente. Possui ainda uma outra thread sempre em execução, que pertence á classe "Receiver_UDP", responsável por escutar todo tráfego via UDP de outros nodos da rede para o nodo em questão. Caso esta última thread receba um novo pedido de conexão via UDP, ou seja, um pedido de conexão de um outro nodo da rede e de um novo cliente, lança novas threads, pertencentes também á classe "AnonGW_Worker", para tratarem de todo o tráfego recebido via UDP, redirecioná-lo para o servidor via TCP, e escutar respostas deste mesmo e encaminha-las via UDP para o nodo ao qual se emparelhou para tal cliente. Desta forma, cada cliente, que pretenda usar a rede desenvolvida, terá um conjunto de threads apenas respeitante a si mesmo, o que faz com que a eficiência seja alta e que vários clientes possam realizar transferências de forma completamente simultânea e paralela.

É na classe "Receiver_UDP", que é implementada parte da lógica de multiplexagem de clientes, nomeadamente, o algoritmo que permite a um nodo receber pacotes via UDP de nodos e clientes diferentes, e saber para onde os deve redirecionar. Para este efeito, esta classe implementa duas threads para realizar o trabalho proposto. A primeira, é apenas responsável por receber pacotes UDP, e quando recebe , acorda a segunda thread, denominada "Processor", que irá tratar de reconstruir o pacote de bytes no PDU correspondente, e guardá-lo na instancia de cliente correta e alertar as threads responsáveis por tal cliente, que existe um novo pacote acabado de chegar e que devem ser despoletadas as ações correspondentes, como por exemplo, caso seja uma pacote de dados, reencaminhá-lo para o servidor ou cliente, dependentemente do caminho que esteja a percorrer, e enviar uma mensagem de confirmação de chegada ao nodo que enviou o PDU via UDP.

A classe "AnonGW_Worker", permite, como referido anteriormente, lançar um conjunto de threads que irá tratar de cada cliente que se conecte. Um facto que foi tido em conta, e bastante importante em questões de eficiência, foi implementar os mecanismos necessários para todas as leituras e escritas , quer seja TCP ou UDP, não serem bloqueantes, no sentido de que, o primeiro nodo será capaz de escutar via TCP pedidos do cliente e enviar-lhe pedaços de resposta já processados de forma simultânea, ou por exemplo, o segundo nodo da ligação escutar dados via UDP e ao mesmo tempo estar a escutar via TCP do servidor pacotes já processados , ou até enviar via UDP ao primeiro nodo respostas que já possua. Outro mecanismo relevante implementado aqui, foi quando um dos nodos envia PDU de dados via UDP a um outro nodo, lançar uma nova thread, instância da classe "Ack_Waiter", que irá ser responsável por esperar a confirmação de receção, e caso não receba, enviar novamente e esperar novamente. Desta forma, o nodo pode continuar a enviar outros pedaços de dados, sem bloquear há espera de receber determinadas confirmações e, mesmo assim, garantir que não há perdas de dados. Outros dois factos importantes implementados nesta classe, é garantir a entrega ordenada, e garantir que não há repetições de dados enviados, pois, imaginemos o seguinte cenário:

O segundo nodo acabou de receber um PDU de dados via UDP, e por isso irá enviar um ACK de confirmação ao primeiro nodo. Mas, este ACK é perdido, e por isso, o primeiro nodo irá retransmitir o pacote novamente. O segundo nodo irá receber um pacote de dados repetido e, irá perceber esse facto através do número de sequência do PDU e, irá ignorar o pacote para não haver erros na transmissão. No entanto irá retransmitir o ACK para o primeiro nodo perceber que o pacote foi transmitido com sucesso.

As últimas duas classes que interessa referir, são a classe "Client", e a classe "Sender_UDP". A primeira permite instanciar um novo cliente e, possui variáveis que permitem alertar as threads que tratam deste e estão adormecidas até certo evento acontecer, de forma a não consumirem CPU numa espera ativa, e possui ainda variáveis que o permitem identificar unicamente e guardar todos os dados e pacotes necessários em auxílio ao algoritmo de comunicação implementado. A classe "Sender_UDP" é na verdade bastante simples, possuindo apenas todas as funcionalidades de transmissão via UDP entre dois nodos, dos diferentes PDU's existentes.

Por último, são apresentados DSS que ilustram então o algoritmo implementado:

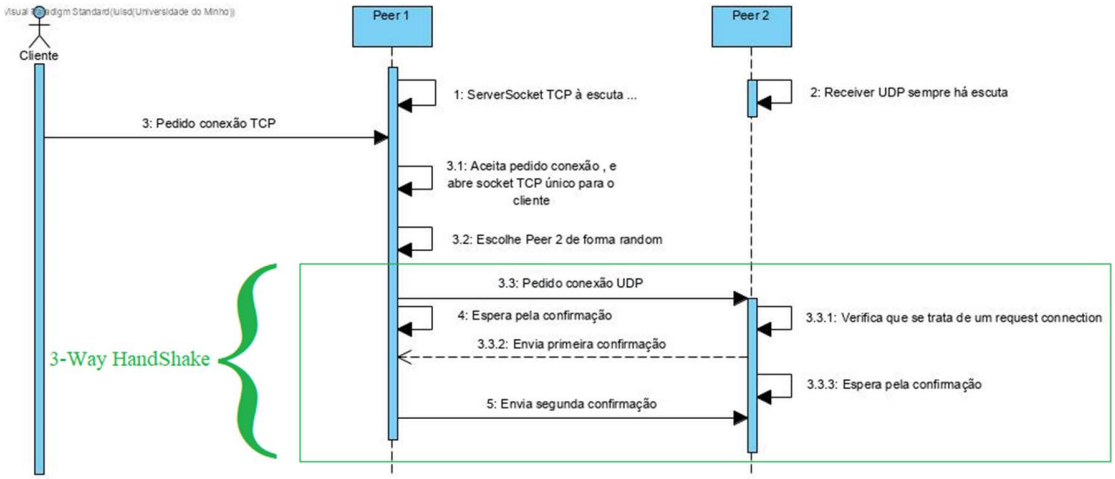

Fig.5. DSS referente ao estabelecimento de uma conexão

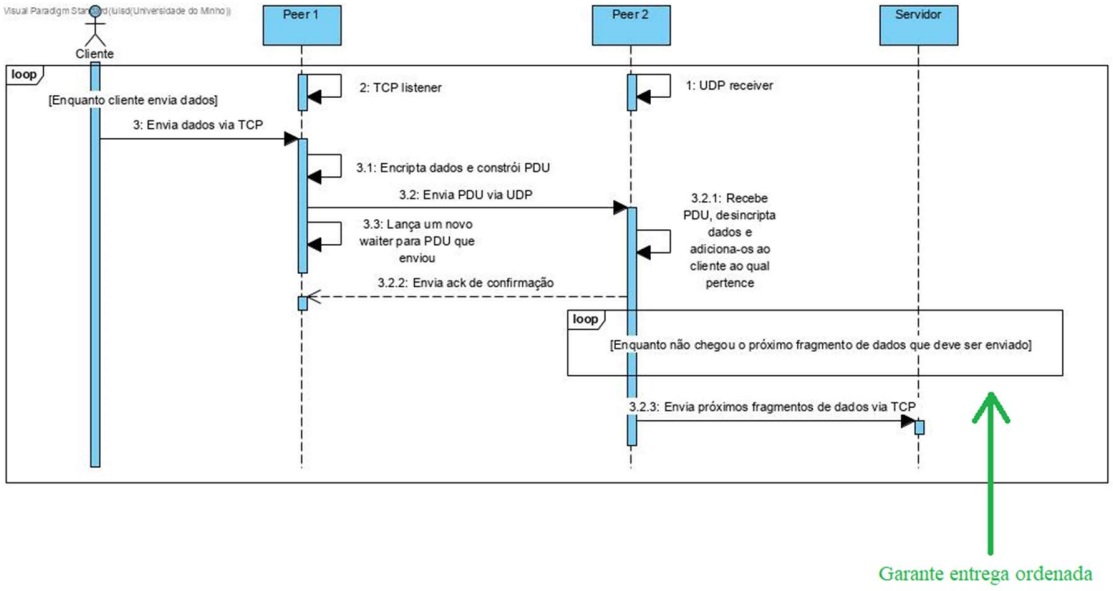

Fig. 6. DSS referente à transferência de dados sentido cliente -> servidor

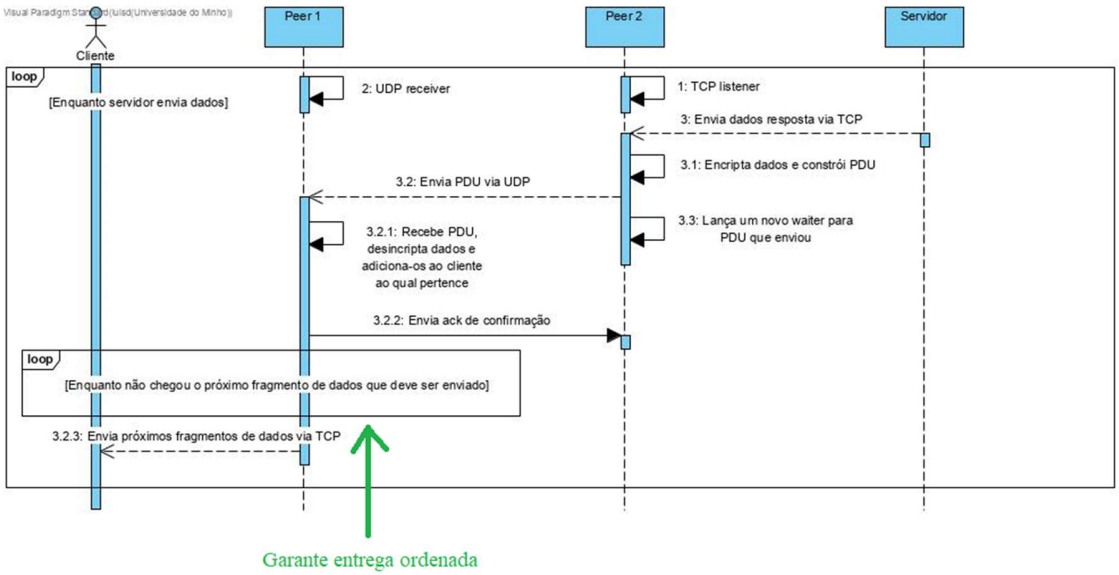

Fig. 7. DSS referente à transferência de dados sentido servidor -> cliente

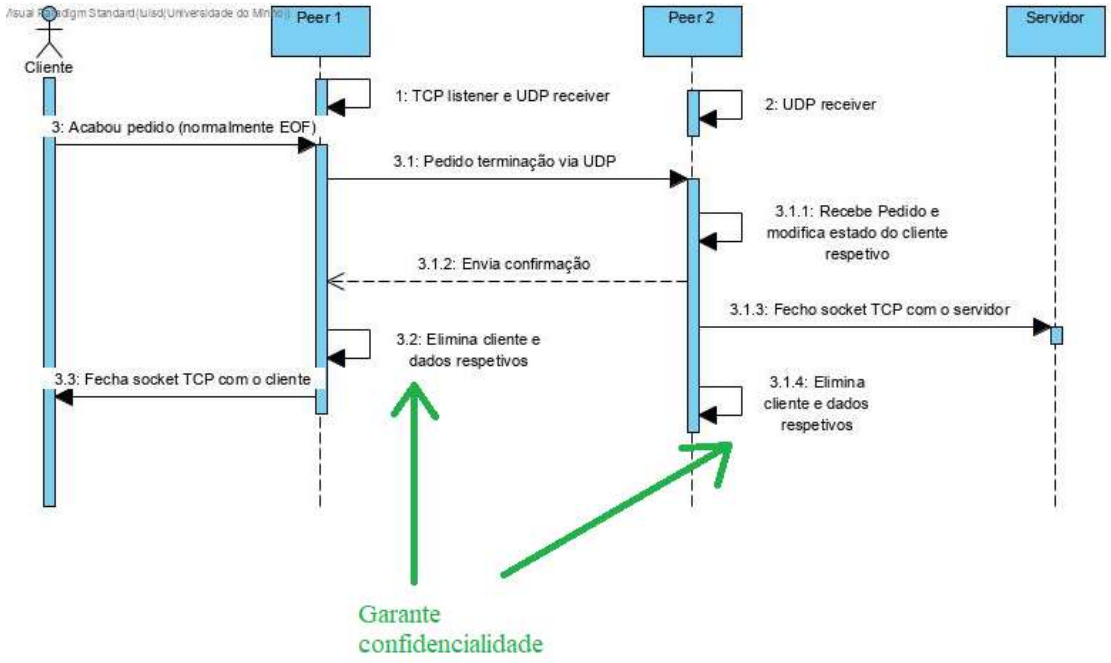

Fig. 8. DSS referente ao término de conexão

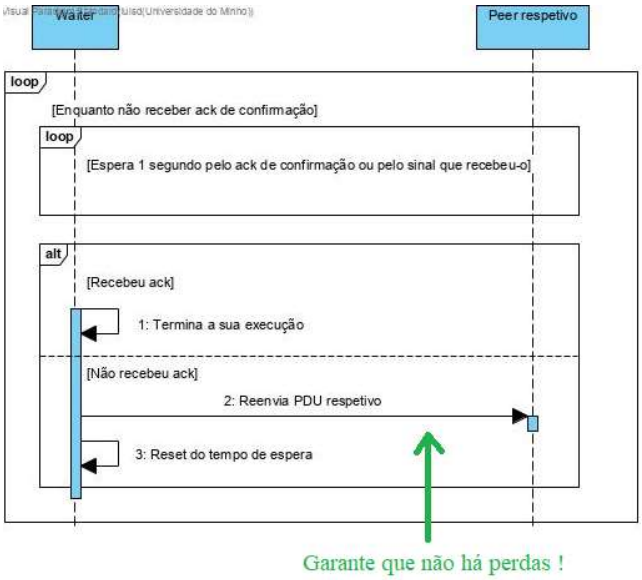

Fig. 9. DSS referente ao controlo de perdas

4 Testes e Resultados

De seguida, são apresentados dois testes ilustrativos da execução do programa desenvolvido. O primeiro teste, mostra o funcionamento da encriptação e desencriptação de uma mensagem, enquanto o segundo mostra dois clientes a realizarem um pedido em simultâneo, de um ficheiro de tamanho razoável e numa rede com quatro nodos a trabalhar.

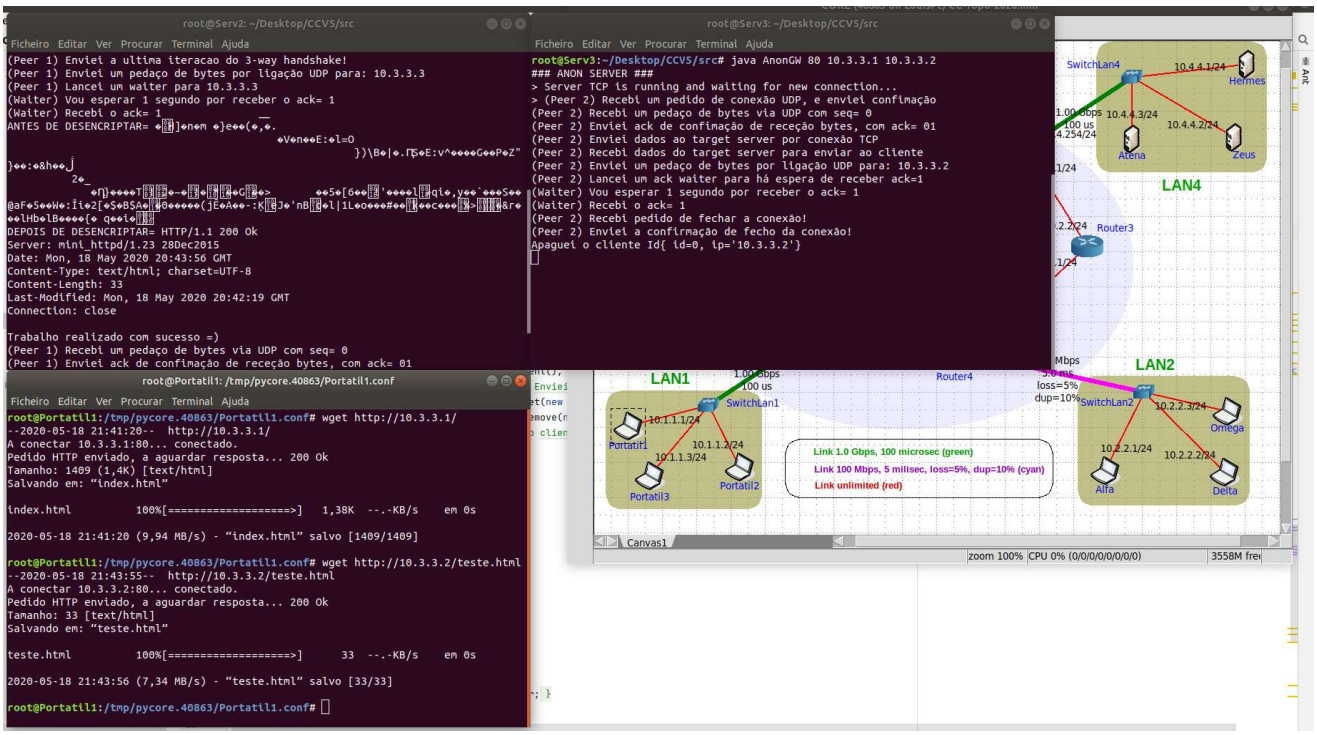

Fig. 10. Teste da encriptação dos dados

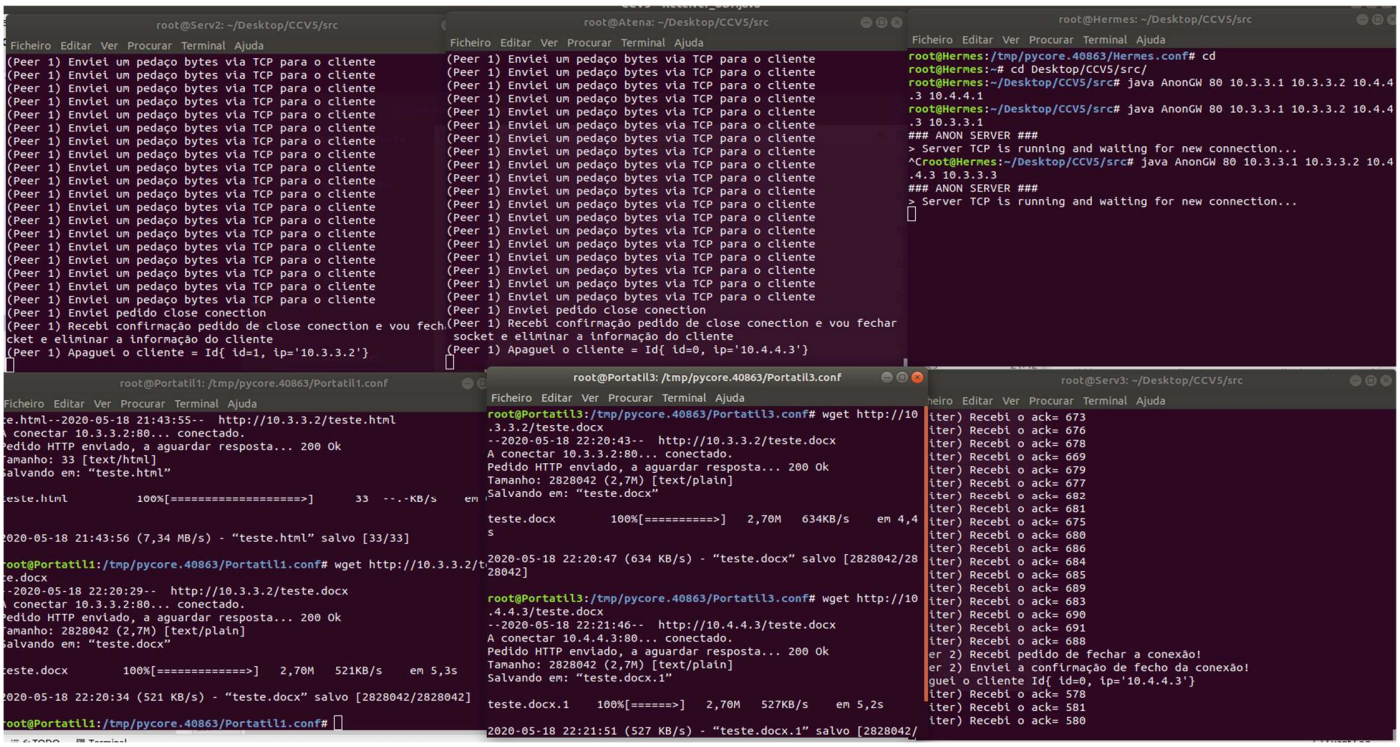

Fig. 11. Teste à transferência com dois clientes

5 Conclusões

Foi desenvolvida com sucesso uma solução para o problema que nos foi apresentado, que inclui a entrega ordenada dos dados, a confidencialidade dos mesmos visto que são cifrados por AES antes de serem enviado e desencriptados na receção, a multiplexagem de clientes, visto que suporta mais do que um cliente a realizar pedidos ao mesmo tempo, e o controlo de perdas, visto que é garantido o reenvio de pacotes perdidos. Tendo sido também desenhado um protocolo adequado para todo este funcionamento como especificado anteriormente. Poderia, no entanto, também ter sido realizada a autenticação de origem, mas devido a falta de tempo não foi possível.