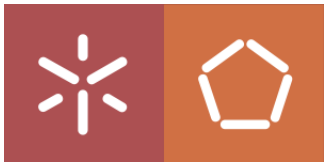

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Comunicações por Computador

Trabalho Prático 2

Ano Letivo de 2024/2025

Network Monitoring System

Flávia Alexandra da Silva Araújo (A96587)
Joshua David Amaral Moreira (A105684)
Miguel Torres Carvalho (A95485)

Grupo 10 - PL1

7 de dezembro de 2024

CC

Resumo

No presente relatório, é apresentada a solução desenvolvida para a Unidade Curricular de **Comunicações por Computador**, como parte do projeto final do semestre - ***Network Monitoring System***. Este trabalho teve como objetivo projetar um sistema capaz de monitorizar o tráfego de rede, bem como o desempenho dos dispositivos, entre um servidor centralizado e vários agentes distribuídos, permitindo a execução de tarefas de monitorização, com a respetiva recolha de métricas de desempenho e envio de alertas previamente configurados. Adicionalmente, foi desenvolvida a análise de dados recolhidos e a apresentação dos mesmos em formato gráfico, permitindo a visualização do desempenho da rede e dos dispositivos monitorizados.

No sistema desenvolvido implementaram-se dois protocolos aplicacionais:
AlertFlow: destinado ao envio de alertas quando certas condições de monitorização definidas são excedidas, utilizando, como protocolo de transporte, o *Transmission Control Protocol* (TCP).
NetTask: implementado para a comunicação de tarefas de monitorização e recolha de métricas, sobre o *User Datagram Protocol* (UDP), este protocolo aplicacional foi desenvolvido de forma a assegurar a comunicação robusta e confiável entre o servidor e os agentes.

O trabalho foi concluído com a implementação e validação das principais funcionalidades do sistema, demonstrando a sua eficácia em ambientes de teste representativos em ambientes virtualizados. Este relatório documenta detalhadamente o *design*, implementação, utilização e resultados obtidos.

Palavras-Chave: *Network Monitoring System*, TCP, UDP, *AlertFlow*, *NetTask*, *python3*, *iperf3*, *ping*, *Comunicações por Computador*.

Índice

Índice de Figuras

Índice de *Snippets*

1 Arquitetura da Solução

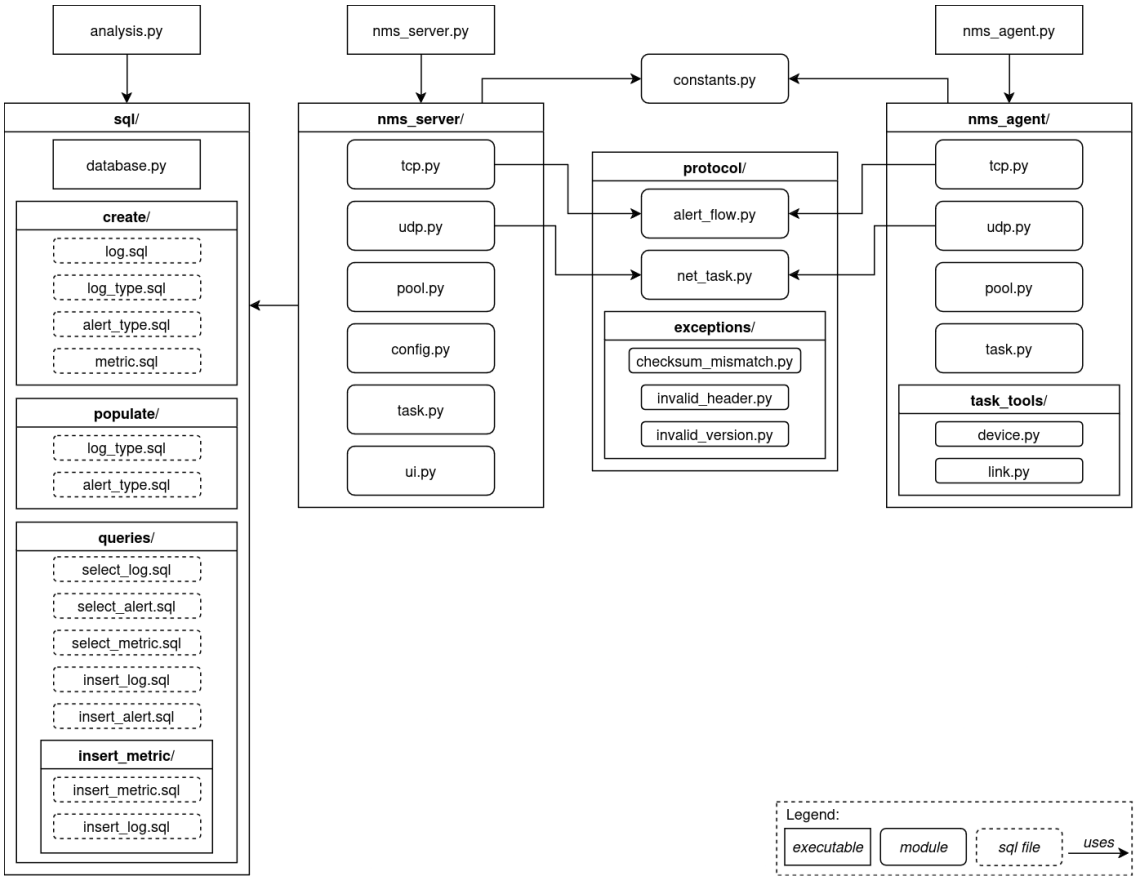

Figura 1.1: Arquitetura da solução *Network Monitoring System*

2 Especificações dos Protocolos Aplicacionais

2.1 *AlertFlow*

O protocolo *AlertFlow*, operado sobre o TCP, foi desenvolvido para a comunicação de alertas de agentes para o servidor, sendo assim utilizado monodirecionalmente. O envio de alertas ocorre quando certas condições de monitorização definidas são excedidas, sendo estas previamente indicadas pelo servidor, via o protocolo *NetTask*. Este protocolo é orientado à conexão, ou seja, cada vez que um agente envia um alerta, é estabelecida uma nova conexão TCP com o servidor, sendo esta terminada após o envio da mesma.

2.1.1 Formato de Cabecalho e Descrição de Campos

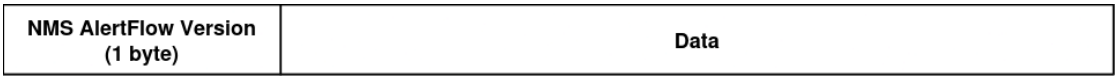

Figura 2.1: Formato do cabeçalho do protocolo *AlertFlow*

- ***NMS AlertFlow Version*** (1 *byte*): versão do protocolo, para assegurar a compatibilidade de versões entre o servidor e os agentes;
- ***Data/Payload*** (n *bytes*): carga útil da mensagem com tamanho variável. Utiliza a encodificação *UTF-8*, com um formato JSON para a transmissão de dados.

2.1.2 Descrição de Funcionalidades

Compatibilidade de Versões

A compatibilidade de versões é garantida através do campo *NMS AlertFlow Version* do cabeçalho da mensagem, permitindo a identificação da versão do protocolo entre o servidor e os agentes. Se a versão do protocolo do emissor for diferente da versão do recetor, é apresentada uma mensagem de erro com as diferentes versões. Esta é meramente indicativa à qual o formato usado nos dados da mensagem.

2.2 *NetTask*

O protocolo *NetTask* é essencial para a funcionalidade harmonizada do ***Network Monitoring System***, sendo este usado para a maioria das comunicações entre o servidor e os agentes, tais como, a primeira conexão de um agente ao servidor, o envio de tarefas pelo servidor, o envio de resultados de tarefas pelos agentes, e a terminação de conexões nos dois sentidos, sendo um protocolo orientado aos datagramas.

Como este opera em cima da camada de transporte UDP, o protocolo *NetTask* foi desenvolvido para ser robusto e adaptável a condições adversas de rede, garantindo a entrega fiável e integral de mensagens, sobretudo em rotas deterioradas, com perdas ou duplicação de pacotes, latências elevadas e taxas de débito variáveis.

Para combater tais adversidades, o protocolo aplicacional *NetTask* responsabiliza-se pelas funcionalidades que serão exploradas no seguinte subcapítulo 2.2.2.

Nos próximos subcapítulos, serão detalhadas as especificações do protocolo, nomeadamente o formato do cabeçalho e descrição dos respetivos campos, descrição de funcionalidades e diagramas de sequência que ilustram o comportamento do protocolo em situações normais e adversas.

2.2.1 Formato de Cabeçalho e Descrição de Campos

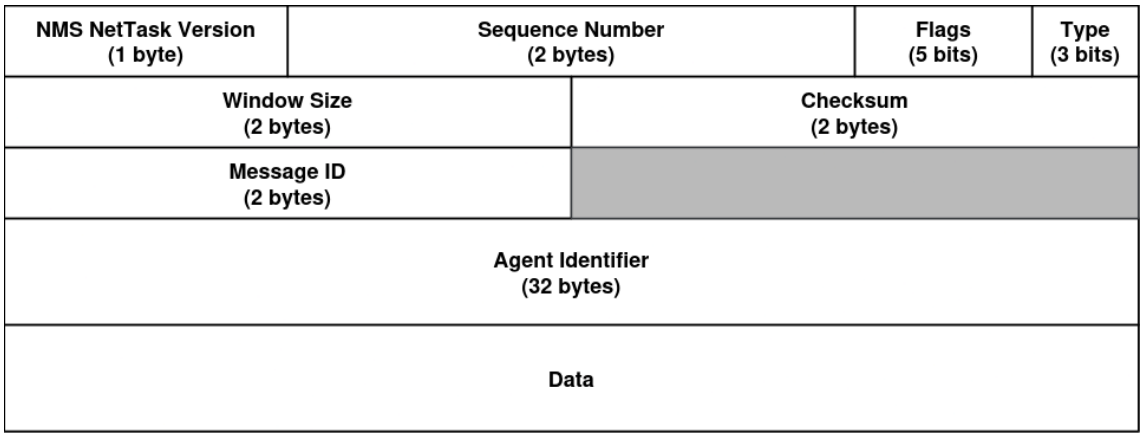

Figura 2.2: Formato do cabeçalho do protocolo *NetTask*

- ***NMS NetTask Version*** (1 *byte*): versão do protocolo, para assegurar a compatibilidade de versões entre o servidor e os agentes;
- ***Sequence Number*** (2 *bytes*): número de sequência da mensagem, para a ordenação de pacotes, deteção de pacotes duplicados e identificação de *acknowledgements*;

- ***Flags*** (5 *bits*): *flags* de controlo:

  ACK (1º *bit*): *Acknowledgement*, utilizado para confirmar a receção de pacotes;

  RET (2º *bit*): *Retransmission*, indica que o pacote é uma retransmissão;

  URG (3º *bit*): *Urgent*, indica que a mensagem é urgente, esquivando-se de mecanismos de controlo de fluxo;

  WP (4º *bit*): *Window Probe*, utilizado para o controlo de fluxo;

  MF (5º *bit*): *More Fragments*, para (des)fragmentação de pacotes.

- ***Type*** (3 *bits*): tipo da mensagem:

  0 ***Undefined***: mensagem indefinida, utilizada para testes ou quando nenhum tipo de mensagem é aplicável, por exemplo no envio de *window probes*;

  1 ***First Connection***: primeira conexão de um agente ao servidor;

  2 ***Send Tasks***: envio de tarefas pelo servidor;

  3 ***Send Metrics***: envio de resultados de tarefas pelos agentes;

  4 ***EOC (End of Connection)***: terminação de conexões nos dois sentidos;

  ***** ***Reserved***: reservado para futuras extensões (de 5 a 7);

- ***Window Size*** (2 *bytes*): indica o tamanho da janela de receção, para o controlo de fluxo;

- ***Checksum*** (2 *bytes*): soma de verificação da mensagem, para a deteção de erros;

- ***Message Identifier*** (2 *bytes*): identificador da mensagem, utilizado para a desfragmentação e ordenação de pacotes;

- ***Agent Identifier*** (32 *bytes*): identificador do agente, podendo este ser recetor ou emissor da mensagem;

- ***Data/Payload*** (n *bytes*): carga útil da mensagem com tamanho variável, contendo a informação a ser transmitida nas mensagens do tipo *Send Tasks* e *Send Metrics*. Utiliza a encodificação *UTF-8*, com um formato JSON para a transmissão de dados.

2.2.2 Descrição de Funcionalidades

Retransmissão de Pacotes Perdidos

A retransmissão de pacotes perdidos é efetuada quando o emissor não recebe um *acknowledgement* de um pacote enviado, após um determinado intervalo de tempo. Para a concretização desta funcionalidade, o emissor guarda todos os pacotes enviados em memória, reenviando os pacotes não confirmados após o intervalo de tempo definido no fichero `constants.py` sobre a nomenclatura `RETRANSMIT_SLEEP_TIME`. Os pacotes mantém o cabeçalho original, sendo apenas ativada a *flag RET*.

(Des)Fragmentação e Ordenação de Pacotes

A fragmentação de pacotes ocorre quando a carga útil da mensagem excede o tamanho máximo permitido, sendo este tamanho predefinido para 1500 *bytes*, no ficheiro `constants.py` sobre a nomenclatura `BUFFER_SIZE`, uma vez que este é o valor máximo para o *Maximum Transmission Unit* (MTU).

Na fragmentação de pacotes, os dados da mensagem (campo *Data*) são divididos em fragmentos, de forma a que, com adição do cabeçalho, o tamanho total do pacote não exceda o tamanho máximo permitido. Os primeiros fragmentos são marcados com a *flag MF*, com exceção do último fragmento, indicando que não existem mais fragmentos a serem enviados. Para a identificação dos fragmentos, é atribuído um *Message Identifier* único, sendo este igual ao número de sequência do primeiro fragmento. Os números de sequência dos fragmentos são incrementados de forma sequencial.

Na desfragmentação, o recetor mantém em memória os fragmentos recebidos, cada pacote recebido é guardado em um *array* caso seja fragmentado. Para identificar se um pacote é ou não um fragmento, é verificado se a *flag MF* está desativada e se o *Message Identifier* é igual ao número de sequência do pacote. Se tal não se verificar, o pacote é guardado em memória até que todos os fragmentos sejam recebidos, ou seja, devem existir todos os fragmentos tal que o seu número de sequência esteja entre os seus *Message Identifiers* e o número de sequência do último fragmento. Este procedimento garante que a mensagem é desfragmentada apenas quando todos os fragmentos são recebidos, uma vez que a ordem de chegada destes não é garantida.

Uma vez que todos os fragmentos são recebidos, estes são ordenados pelo número de sequência e os dados da mensagem são concatenados para formar a mensagem original.

O campo *Message Identifier* permite que fragmentos de diferentes mensagens sejam distinguidos, garantindo a integridade dos pacotes reconstruídos.

Deteção e Manuseamento de Pacotes Duplicados

A deteção e manuseamento de pacotes duplicados, tal como na (des)fragmentação e ordenação de pacotes, recorre ao *Sequence Number* da mensagem. O recetor mantém um registo dos números de sequência de pacotes recebidos para a deteção de pacotes duplicados, descartando pacotes que já tenham sido recebidos anteriormente, e reenviando um *acknowledgement* ao emissor para confirmar a receção do pacote de forma a evitar futuras retransmissões desnecessárias.

Deteção de Erros

A deteção de erros é efetuada através da soma de verificação da mensagem (*Checksum*), calculada com base no cabeçalho e na carga útil da mensagem. O recetor, ao receber um pacote, calcula a soma de verificação e compara-a com a recebida. Se a soma de verificação calculada for diferente da recebida, o pacote é descartado, e o emissor uma vez que não recebe um *acknowledgement* reenviará o pacote.

Compatibilidade de Versões

A compatibilidade de versões é garantida através do campo *NMS NetTask Version* do cabeçalho da mensagem, permitindo a identificação da versão do protocolo entre o servidor e os agentes. Se a versão do protocolo do emissor for diferente da versão do recetor, é apresentada uma mensagem de erro com as diferentes versões. Uma possível alteração na implementação seria descartar a mensagem, evitando possíveis incompatibilidades e erros, contudo, a decisão foi manter a mensagem de erro e tentar processar o pacote, uma vez que a diferença de versões pode não impedir uma troca sem conflitos, deste modo, não se sobrecarrega a largura de banda com múltiplas retransmissões de pacotes.

Controlo de Fluxo

O controlo de fluxo é efetuado através do tamanho da janela de receção (*Window Size*), indicando ao emissor o número de pacotes que o recetor pode receber sem congestionar a ligação. O valor inicial da janela de receção é definido no ficheiro `constants.py` sobre a nomenclatura `INITIAL_WINDOW_SIZE`, sendo este valor de 32 pacotes. O emissor, ao enviar um pacote, indica a sua janela de receção, sendo esta referente à quantidade de espaço disponível na sua lista de pacotes por desfragmentar e ordenar, uma vez que na implementação atual, é criada uma *thread* para processar pacotes recebidos, então apenas os pacotes fragmentados são guardados em memória, servindo estes para a indicação do tamanho da janela de receção. Quando a janela de receção do recetor é menor ou igual a zero, o emissor, antes de enviar pacotes, aguarda até que essa incremente, de forma a não congestionar a rede e o recetor. Para esta medição dos *window sizes* dos recetores, é, posteriormente, criada uma *thread* responsável por enviar *window probes* aos mesmos com *window sizes* menores ou iguais a zero, de forma a que estes respondam com a sua janela de receção.

2.2.3 Diagramas de Sequência

First Connection

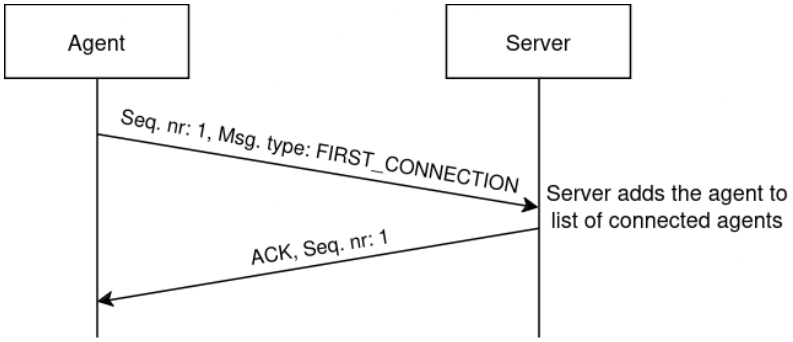

Figura 2.3: Diagrama de sequência do protocolo *NetTask - First Connection*

End of Connection (EOC)

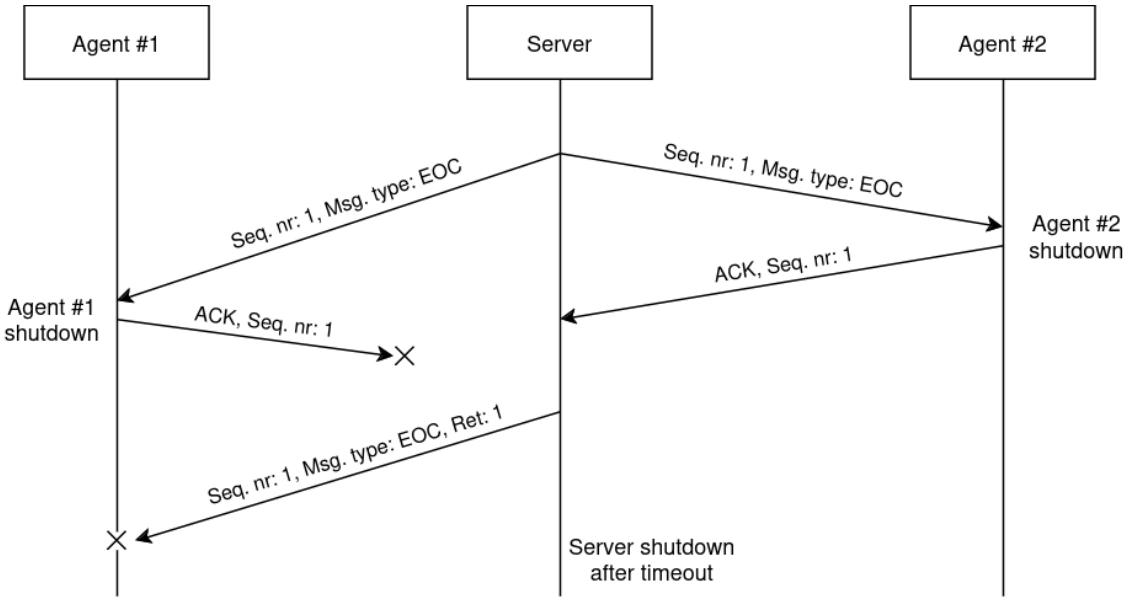

Figura 2.4: Diagrama de sequência do protocolo *NetTask - End of Connection*

Neste exemplo, o servidor é interrompido, sendo enviado um pacote *EOC* para todos os agentes, terminando todas as execuções. O mesmo pode ser feito quando um agente é interrompido, sendo enviado um pacote *EOC* para o servidor, e este remove o agente da lista de agentes conectados.

Note que, em rotas deterioradas, a mensagem de *End of Connection* pode não ser recebida, então é necessário determinar um tempo para um *timeout* adequado as condições da rede, de forma a garantir a terminação da conexão com sucesso.

Retransmissão, Manuseamento de Pacotes Duplicados, Desfragmentação, Ordenação e Deteção de Erros

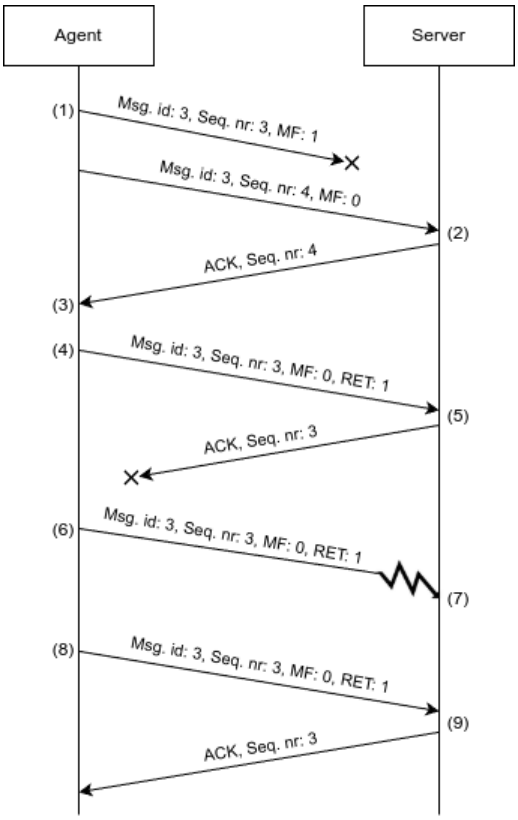

Figura 2.5: Diagrama de sequência do protocolo *NetTask* - Retransmissão, Manuseamento de Pacotes Duplicados, Desfragmentação, Ordenação e Deteção de Erros

1. O agente envia uma métrica, com dados superiores ao `BUFFER_SIZE`, subtraído do *header size*, sendo este fragmentado em dois pacotes;

2. O servidor recebe o pacote, e como o *Message Id* é diferente do *Sequence Number*, deduz que faltam mais fragmentos, adicionando-o ao seu *buffer* de pacotes a desfragmentar, e envia o ACK;

3. O agente remove o pacote de número de sequência 4 da lista de pacotes por retransmitir;

4. O agente, após aguardar `RETRANSMIT_SLEEP_TIME` segundos, retransmite o pacote não *acknowledged*;

5. O servidor possui todos os fragmentos, ordena e desfragmenta os mesmos, guardando a métrica na base de dados. Seguidamente, envia o ACK, que será perdido;

6. O agente, ao não receber o ACK, retransmite o pacote pela segunda vez, que, pela

sua rota, sofre de alterações no seu conteúdo;

7. O servidor, ao receber o pacote, calcula o *checksum* e verifica que é diferente do expectado, descartando o pacote. Como tal, não envia um ACK;

8. O agente, ao não receber o ACK, retransmite o pacote novamente;

9. O servidor descarta o pacote duplicado, pois, no passo (5), já tinha guardado o número de sequência 3. O servidor reenvia o ACK para evitar retransmissões desnecessárias.

Controlo de Fluxo

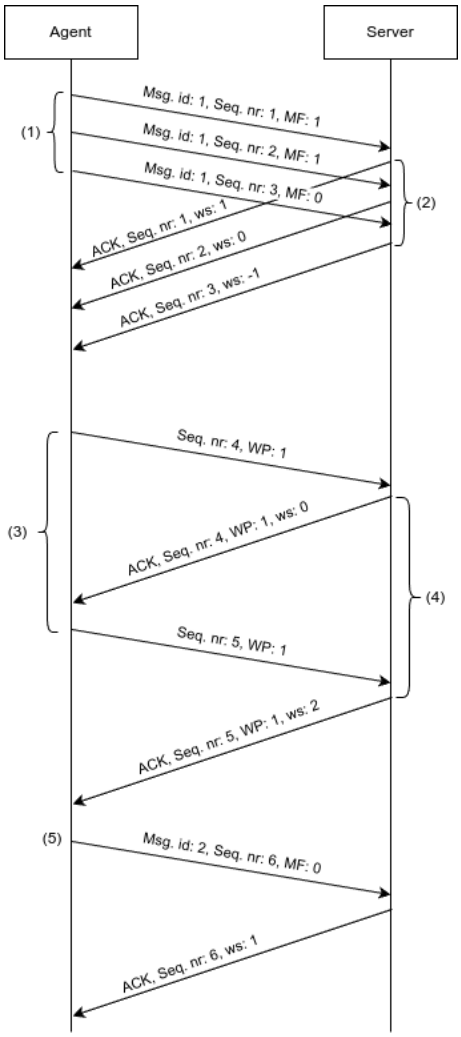

Figura 2.6: Diagrama de sequência do protocolo *NetTask* - Controlo de Fluxo

O agente pretende enviar dois pacotes, sendo o primeiro de grande porte, num curto espaço de tempo. Este primeiro tem que ser fragmentado e enviado em três pacotes distintos **(1)**. Entretanto, a *window size* encontra-se a dois. O servidor, ao receber os pacotes, encarrega-se de enviar os respetivos ACKs, atualizando a sua *window size*, até que, no terceiro pacote, ela encontra-se a -1 **(2)**. O agente necessitava enviar mais um pacote, porém, a *window size* é inferior a zero, então sabe que o servidor tem o *buffer* cheio, pelo que envia *window probes* **(3)** para o servidor até este lhe indicar que a sua *window size* voltou a ser superior a zero **(4)**. Posto isto, o agente pode finalmente enviar o pacote em falta **(5)**.

3 Implementação

3.1 Parâmetros dos Executáveis

3.1.1 `nms_server.py`

```
$ ./nms_server.py --help
usage: nms_server.py [-h] [-c CONFIG] [-v]

Network Management System Server

options:
  -h, --help         show this help message and exit
  -c CONFIG, --config CONFIG
                     Configuration file path
  -v, --verbose      Enable verbose output
```

Snippet 3.1: Parâmetros do executável *nms_server.py*

3.1.2 `nms_agent.py`

```
$ ./nms_agent.py --help
usage: nms_agent.py [-h] [-s SERVER] [-v]

Network Management System Agent

options:
  -h, --help         show this help message and exit
  -s SERVER, --server SERVER
                     Server IP
  -v, --verbose      Enable verbose output
```

Snippet 3.2: Parâmetros do executável *nms_agent.py*

3.2 Ficheiro de Configuração

O ficheiro de configuração permite a definição de tarefas aos agentes pelo servidor. Este é carregado no início da execução do servidor, e é enviado a cada agente, aquando da sua conexão, as tarefas as quais este deve executar. O sistema desenvolvido suporta as seguintes métricas e alertas:

- **Métricas de dispositivo**:
  - ***CPU Usage***: percentagem de utilização da CPU;
  - ***RAM Usage***: percentagem de utilização da memória RAM;
  - ***Interfaces Stats***: número de pacotes por segundo, para cada interface de rede definida.
- **Métricas de rede**:
  - ***Bandwidth***: consumo de largura de banda, em *Mbps* (*iperf* via TCP);
  - ***Jitter***: variação de atraso, em milissegundos (*iperf* via UDP, *ping*);
  - ***Packet Loss***: percentagem de pacotes perdidos (*iperf* via UDP, *ping*);
  - ***Latency***: atraso entre o envio e a receção de pacotes, em milissegundos (*ping*).
- **Alertas**: aquando os valores obtidos excedem os limites definidos.
  - ***CPU Usage***;
  - ***RAM Usage***;
  - ***Interfaces Stats***;
  - ***Packet Loss***;
  - ***Jitter***.

Neste ficheiro, é possível definir os parâmetros de utilização de cada ferramenta usada no cálculo das métricas, por exemplo, no caso do `iperf3`, é possível definir a duração do teste, a camada de transporte, se tal é *server* ou *client*, entre outros. Estes parâmetros são utilizados por tarefa, o que permite a diferentes agentes executarem o mesmo teste com diferentes parâmetros.

Outra consideração importante na utilização do `iperf3` é a necessidade de definir as tarefas de ambos os agentes, os quais em modo cliente ou servidor, de forma a que seja possível executar o teste.

3.3 Bibliotecas Utilizadas

Para a implementação do ***Network Monitoring System***, foram utilizadas as seguintes bibliotecas *Python*:

- `socket`: para a comunicação entre o servidor e os agentes, permitindo a troca de mensagens entre os sistemas, tanto em TCP como em UDP;
- `threading`: para a execução *multi threading* de tarefas, servidores TCP e UDP, retransmições de pacotes, entre outros, permitindo multiprocessamento a ambos os agentes como o servidor;
- `json`: para a codificação e descodificação de *payloads* de pacotes em formato JSON;
- `psutil`: para a recolha de métricas de desempenho dos dispositivos, permitindo a obtenção de informações sobre a utilização da CPU, memória RAM e interfaces de rede;
- `iperf3`: *iperf wrapper* para a execução de testes de largura de banda, *jitter* e *packet loss*, permitindo a obtenção facilitada desses dados aquando o *iperf* é executado como cliente;
- `subprocess`: utilizado para correr os comando nativos *ping* e *iperf3* (modo servidor), permitindo obter o *return code*, *stdout* e *stderr* do comando.
- `matplotlib`: para a criação de gráficos, permitindo a visualização das métricas recolhidas e a sua evolução ao longo do tempo.

3.4 Detalhes Técnicos da Implementação

3.4.1 Definição de Constantes

Ao longo da implementação, foram definidas várias constantes, que permitem a configuração do sistema. Estas constantes são definidas no ficheiro `constants.py`, e são as seguintes:

- `ENCODING`: a definição de codificação de mensagens, pré-definida como ***UTF-8***, sendo esta configurável, para, por exemplo, *ASCII*, *ISO-8859-1*, entre outras;
- `BUFFER_SIZE`: o tamanho máximo de um pacote, definido como **1500 *bytes***, este, o valor máximo para o *Maximum Transmission Unit* (MTU) de uma rede Ethernet;
- `SO_NO_CHECK`: este valor permite desativar o cálculo do *checksum* em pacotes UDP, permitindo um controlo de erros personalizado;
- `RETRANSMIT_SLEEP_TIME`: intervalo de tempo, em segundos, para retransmitir pacotes não confirmados. Definiu-se como **5 segundos**, permitindo uma rápida recuperação de pacotes perdidos sem causar sobrecarga excessiva na rede. A definição deste valor teve em conta que a retransmissão de pacotes precisa de um tempo de resposta curto para garantir o desempenho, porém, não tão curto como, por exemplo, nos serviços de *streaming*, onde a latência é crítica. Manter este valor significamente maior reduz a sobrecarga computacional e de tráfego de rede, sendo ótimo para a aplicação desenvolvida;
- `INITIAL_WINDOW_SIZE`: o *window size* inicial foi definido como **32 pacotes**. Este valor é suficiente para lidar com tráfego usual da aplicação, sem causar congestionamento na rede;
- `WINDOW_PROBE_SLEEP_TIME`: intervalo de tempo, em segundos, antes de enviar *window probes* a agentes com *window sizes* menores ou iguais a zero. Definiu-se como **5 segundos**, permitindo uma verificação regular da janela de receção sem causar sobrecarga excessiva na rede;
- `EOC_ACK_TIMEOUT`: tempo máximo de espera, em segundos, para o recebimento de confirmações de término de conexão. O intervalo foi definido como três vezes o tempo de retransmissão para garantir três tentativas de retransmissão antes de encerrar a conexão.

3.4.2 Base de Dados

A base de dados foi implementada em *MySQL*, onde se realizou a criação das tabelas, bem como o seu povoamento e *queries* para a obtenção e inserção de *logs*, métricas e alertas. Estes ficheiros `sql` são posteriormente utilizados no módulo `database.py` para as respetivas operações. O servidor, ao receber as métricas, alertas e novas conexões, guarda estes dados na base de dados, permitindo a análise e visualização dos mesmos.

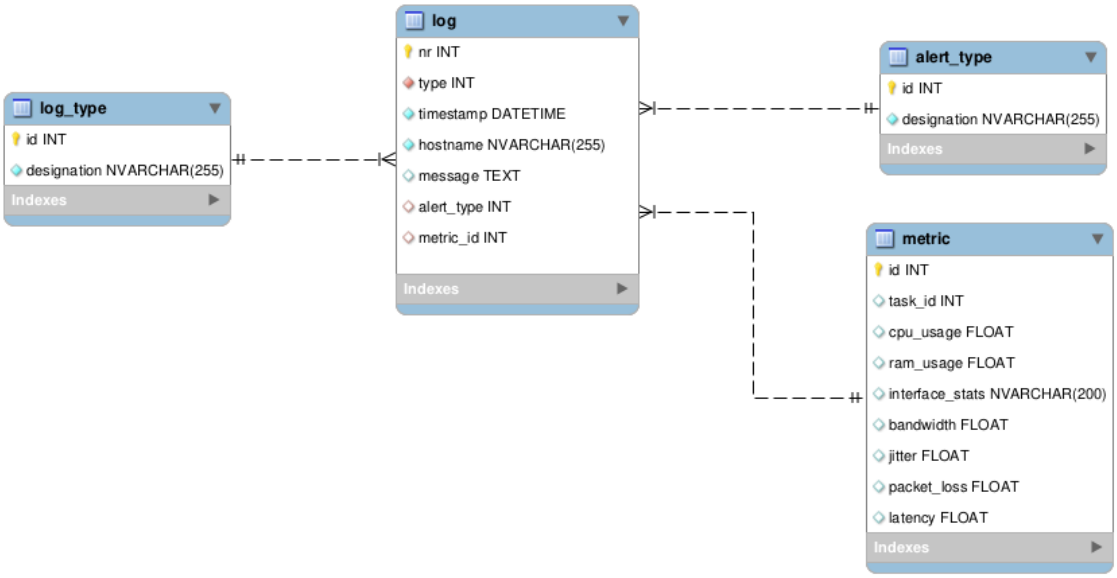

Figura 3.1: Base de Dados do *Network Monitoring System*

3.4.3 Análise Gráfica de Métricas

Para a funcionalidade de análise gráfica de métricas, foi criado o executável `analysis.py` que, após a recolha de métricas da base de dados, permite a criação e visualização de gráficos. Este, permite a visualização das seguintes seis métricas:

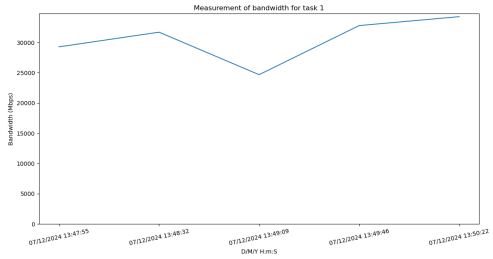

Figura 3.2: *Bandwidth*

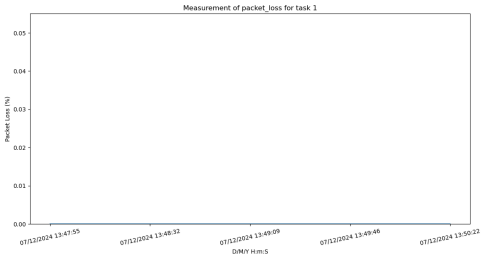

Figura 3.3: *Packet Loss*

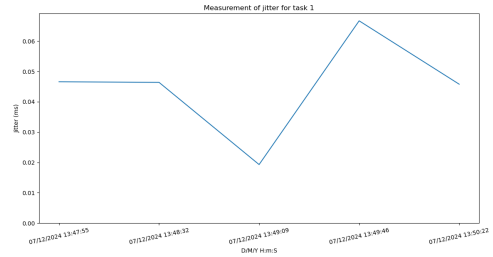

Figura 3.4: *Jitter*

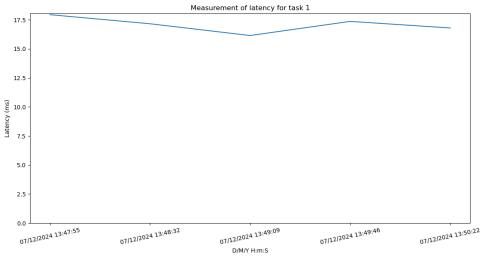

Figura 3.5: *Latency*

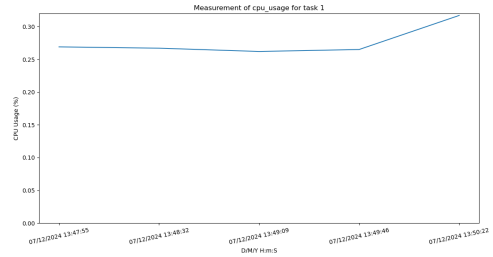

Figura 3.6: *CPU Usage*

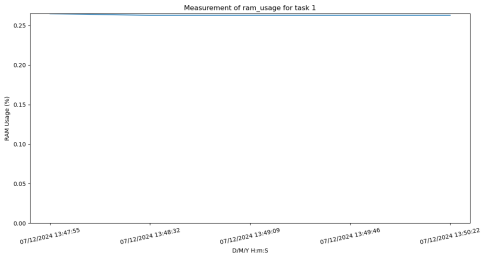

Figura 3.7: *RAM Usage*

4 Testes e Resultados

Para além de testes automáticos para certas funcionalidades protocolares desenvolvidas, as quais serão abordadas na secção seguinte, foram realizados testes manuais com auxílio da ferramenta *Wireshark* para a captura de pacotes, permitindo a análise de mensagens trocadas entre servidor e agentes.

O teste apresentado foi realizado na seguinte topologia de rede virtual:

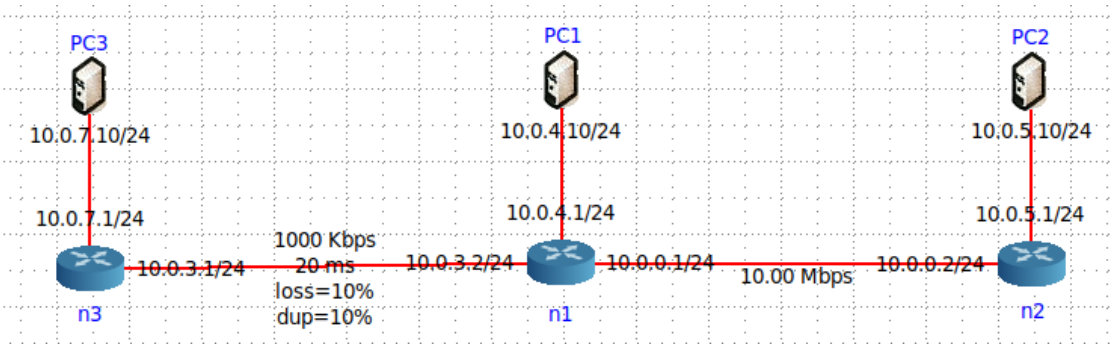

Figura 4.1: Topologia de Rede Utilizada para Testagem

Para tal foi inicializado o servidor com o ficheiro de configuração `config/demo.json`, localizado na raíz do projeto, sendo este de tamanho reduzido para propósitos de demonstração, e o agente `PC2` como executor da respetiva tarefa, que testa a latência com o comando *ping*, de somente três pacotes ICMP, para o `PC3` e verfica a utilização da CPU e RAM, enviando alertas se a utilização de memória exceder os 10%.

Deste modo obtemos o seguinte *output* do servidor e do agente `PC2`:

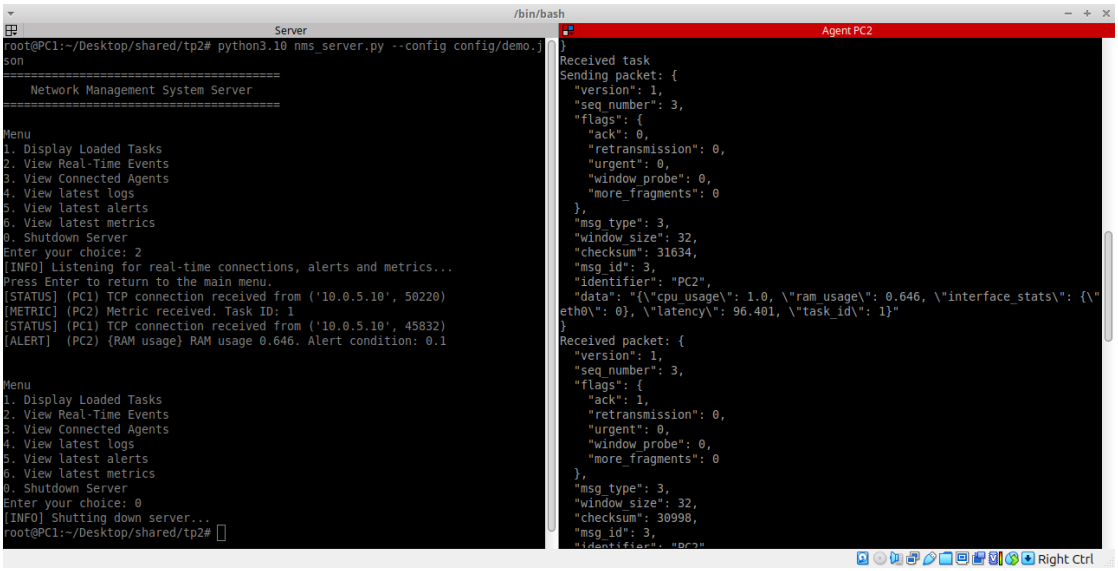

Figura 4.2: *Output* do Servidor e do Agente PC2

Na seguinte captura de pacotes, é possível observar toda a troca de mensagens entre o servidor e o agente `PC2`, desde a primeira conexão, envio da tarefa, envio de métricas (UDP/*NetTask*) e alertas (TCP/*AlertFlow*), até à requisição do término de conexão.

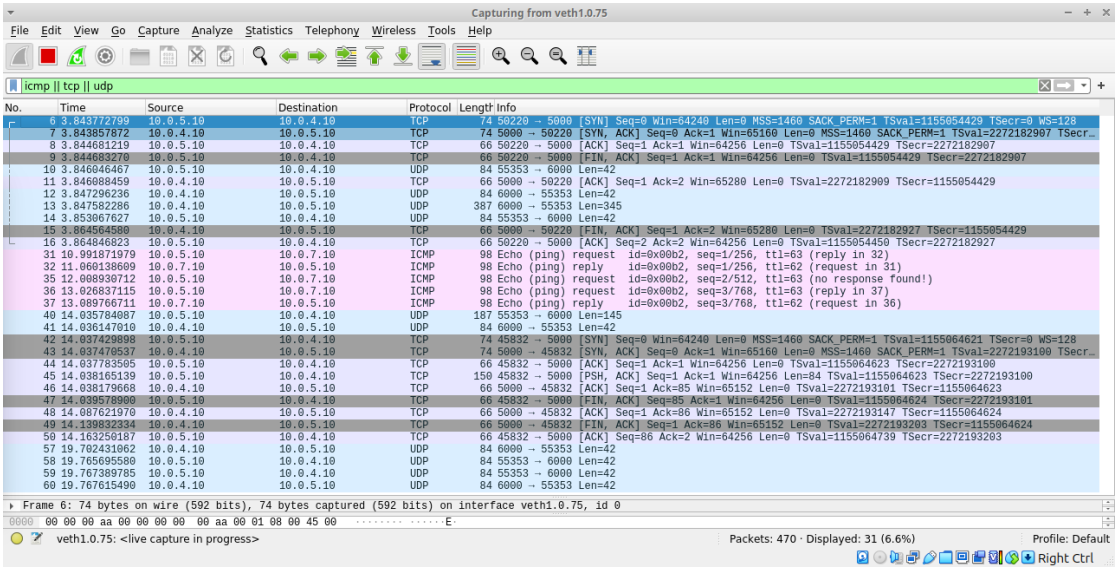

Figura 4.3: Captura de Pacotes com o *Wireshark*

Na captura anterior, é possível observar a troca de mensagens sob o protocolo de camada de transporte, TCP, entre o agente `PC2` e o servidor, no qual na seguinte imagem é apresentado o conteúdo do payload de um alerta *Alert Flow* enviado pelo agente `PC2` ao servidor, indicando que a utilização de memória excedeu os 10%.

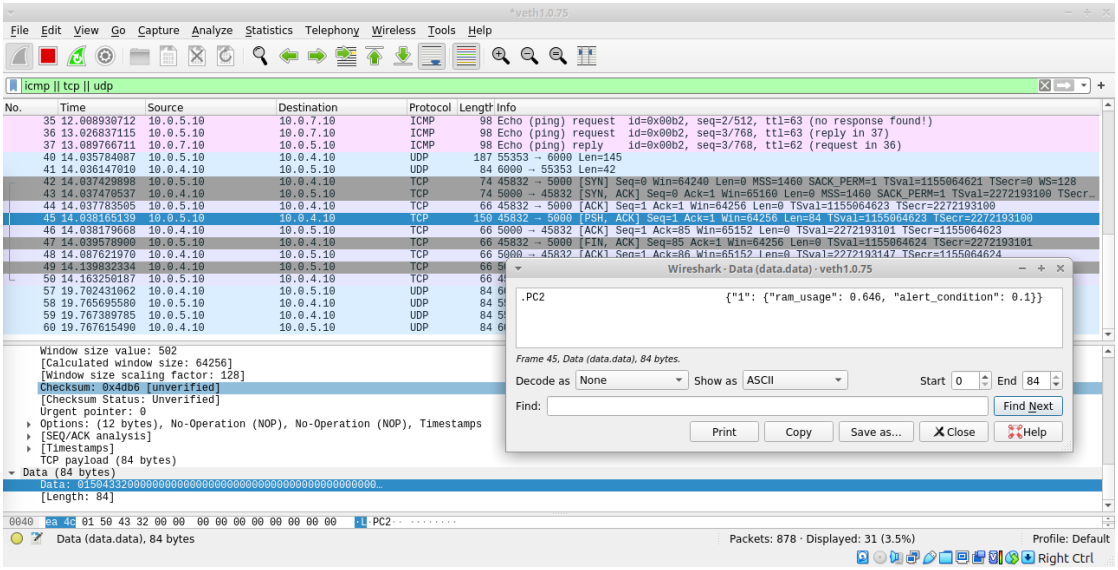

Figura 4.4: Captura dos dados do alerta *Alert Flow*

Poderá consultar a captura efetuada em `topologies/demo.pcapng` na raíz do projeto.

Como supramencionado, foram realizados testes automáticos para certas funcionalidades, das quais o cálculo de *checksum* e a fragmentação e ordenação de pacotes. Para consultar os testes desenvolvidos, poderá aceder à diretoria `tests/` na raíz do projeto.

5 Trabalho Futuro e Conclusões

5.1 Trabalho Futuro

Como trabalho futuro, propõe-se a implementação das seguintes funcionalidades, para uma maior completude do sistema desenvolvido:

- **Encriptação de Mensagens**: adicionar encriptação de mensagens para garantir a confidencialidade e integridade dos dados transmitidos entre o servidor e os agentes, através da incorporação do uso de *nonces* de forma a prevenir ataques *Man-In-The-Middle*. Adicionalmente, uma forma de garantir a autenticidade das mensagens seria a utilização de uma autoridade de certificação para a emissão de certificados digitais para os agentes e o servidor, garantindo totalmente a confidencialidade, integridade, autenticação e não repúdio do originador.
- **Reatribuição de Tarefas em *Runtime***: permitir a reatribuição de tarefas em tempo de execução. Tal poderia ser implementado através do uso do identificador de uma tarefa, permitindo assim a sua adição e atualização, para além disso poderia ser adicionado ao *User Interface* a possibilidade de adicionar tarefas, removê-las e atualizá-las. A nível protocolar, seria apenas necessário adicionar um novo tipo de mensagem, por exemplo, *Remove Task*, que permitiria a remoção de tarefas. Para a opção de atualização de tarefas, apenas seria necessário que o agente atualizasse a tarefa com o mesmo identificador.

5.2 Conclusões

A realização deste projeto permitiu a aplicação prática dos conhecimentos adquiridos durante o semestre na Unidade Curricular de **Comunicações por Computador**, nomeadamente no que diz respeito à conceção e implementação de protocolos aplicacionais para a comunicação entre sistemas.

A implementação do ***Network Monitoring System*** permitiu a integração de vários conceitos e técnicas de comunicação, como a fragmentação de pacotes, retransmissão de pacotes perdidos, controlo de fluxo, deteção de erros e ordenação de pacotes, garantindo a fiabilidade e robustez do sistema.

Por conseguinte, o sistema desenvolvido demonstrou ser eficaz na monitorização e análise de tráfego de rede, permitindo a recolha de métricas de desempenho, a sua análise gráfica e a execução de tarefas de monitorização de forma eficiente e fiável. A sua implementação permitiu a atualização da nossa perceção do quanto damos por garantidos os protocolos de comunicação que utilizamos diariamente.