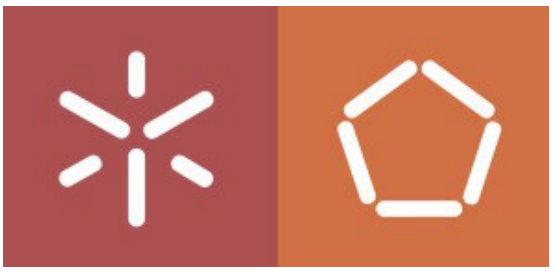

Universidade do Minho Escola de Engenharia

Comunicações por Computador

Relatório Trabalho Prático nº2 PL6 Grupo 5

LEI - 3º Ano 1º Semestre
Ano Letivo 2024/2025

João Serrão
A104444

José Vasconcelos
A100763

Tomás Melo
A104529

1 Introdução

Este relatório descreve a implementação de uma rede composta por um servidor central, denominado *NMS Server*, e vários dispositivos clientes, denominados *NMS Agents*. O *NMS Server* é responsável pelo registo dos clientes e pelo armazenamento das métricas geradas durante a execução das tarefas (*Tasks*) atribuídas a estes dispositivos. Inicialmente, os *NMS Agents* são registados no servidor, que posteriormente lhes envia as tarefas a serem executadas. Durante a execução, os *Clients* retornam as métricas coletadas ao servidor.

A comunicação entre o *NMS Server* e os *NMS Agents* é viabilizada pelo protocolo *NetTask*, que opera sobre comunicação *UDP*. Adicionalmente, caso alguma métrica exceda o valor limite especificado na tarefa, o *Client* emite um alerta ao servidor por meio do protocolo *AlertFlow*, que utiliza comunicação *TCP*.

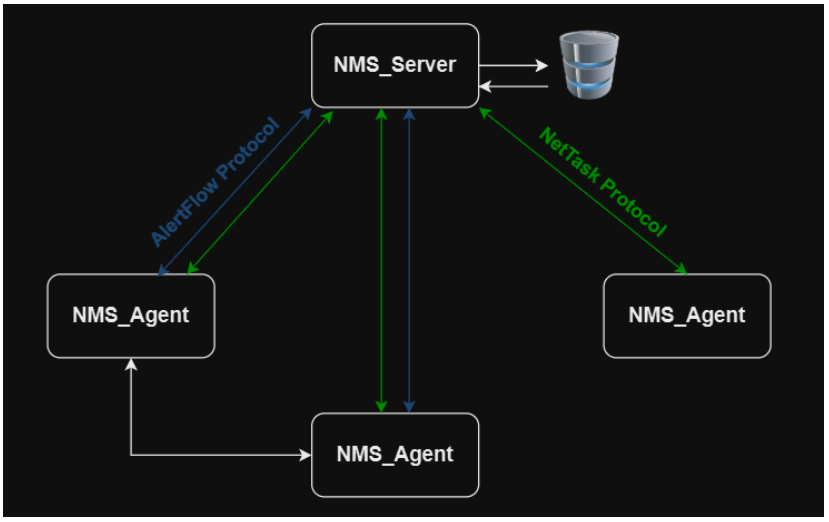

Figura 1. Esquema geral do funcionamento do programa

2 Arquitetura da solução

2.1 NMS Server

(Inicia com *server_main.py* lendo o caminho para o *storage*)

O programa *NMS Server* começa na classe *NMS_Server*, iniciando com a recepção do caminho desejado para a pasta storage. Em seguida, realiza o *parsing* do arquivo `tasks.json`, convertendo-o em um dicionário de tarefas (*Tasks*).

Posteriormente, são criados um *socket UDP* e um *socket TCP*, que ficam em espera por novas conexões. Quando um Agente é registrado, o servidor inicia o processamento das tarefas, criando uma instância de *NMS_server_UDP*. Essa instância é responsável por enviar, receber e retransmitir mensagens relacionadas ao protocolo *NetTask*, enviando a tarefa atual para cada dispositivo requisitado. Caso o dispositivo não exista, a tarefa é armazenada em uma lista de espera (*Waiting List*). Se for necessário realizar um teste com *iperf*, uma mensagem é enviada ao *Agent* correspondente.

Depois de processar todas as tarefas, o servidor entra em um *loop* para processar as tarefas pendentes na lista de espera, desta vez enviando por dispositivo, e não por tarefa.

Por fim, quaisquer resultados, métricas ou situações críticas recebidas são armazenados em um arquivo com o nome do dispositivo na pasta correspondente ao *ID* da tarefa (e.g., *"`/storage/T-1/n1.txt`"*).

2.2 NMS Agent (Client)

(Inicia com *client_main.py* lendo o nome do dispositivo e o IP do servidor)

O programa *Client* começa na classe *Client*, iniciando com a receção do nome do dispositivo associado e o endereço *IP* do servidor.

Posteriormente, o cliente envia seus dados ao servidor e aguarda uma resposta, podendo ocorrer timeout. Assim que a resposta é recebida, o cliente armazena o novo *port* informado pelo servidor e entra em estado de espera pela *task*.

Quando a *task* é recebida, o cliente faz o *parsing* da mensagem e cria entre 2 a 3 *threads*, dependendo se o *AlertFlow* é necessário:

- 1ª *thread* - Focada na execução das tasks e no envio dos resultados.
- 2ª *thread* - Responsável pela medição regular das métricas solicitadas.
- 3ª *thread* - Caso necessária, inicia uma conexão *TCP* para monitorar possíveis situações críticas. Nessas situações, todas as medições são enviadas ao servidor.

Por fim, o cliente permanece em execução até que o usuário decida encerrar o programa.

3 Especificação Protocolar

De forma a gerir e garantir uma boa comunicação entre o `NMS_Server` e os `NMS_Agents (Clients)`, foram criados dois protocolos, `NetTask Protocol`, que funciona sobre o *UDP*, mas que incorpora características do *TCP*, tais como a retransmissão de pacotes,

para o envio e receção de *Tasks* , assim como os resultados e métricas resultantes da execução das mesmas. O `AlertFlow Protocol`, tem como utilidade o envio de notificações de alerta para o `NMS_Server` sempre que houver alterações significativas nas métricas monitorizadas, funcionando sobre o *TCP*.

3.1 NetTask Protocol

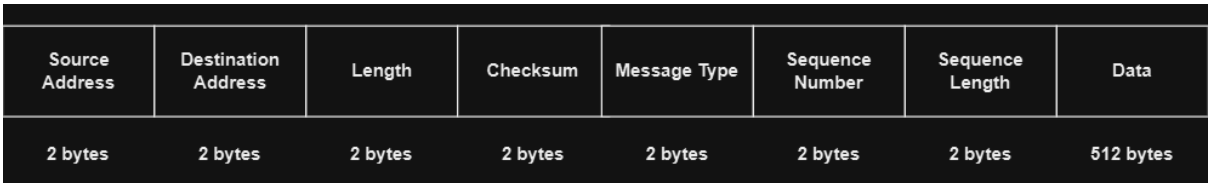

Figura 2. Formato da mensagem protocolar *NetTask*

Este protocolo possui 7 campos de cabeçalho, tendo cada um *2 bytes* de tamanho, ou seja, *14 bytes* no total. O tamanho da mensagem é de *512 bytes*, resultando num total de *526 bytes*. Caso a mensagem a ser enviada ultrapasse os *512 bytes*, esta será fragmentada criando um novo cabeçalho para cada mensagem em que o *sequence number* vai sendo incrementado.

***Source Address*:** Endereço de quem envia o datagrama
***Destination Address*:** Endereço destino do datagrama
***Length*:** Tamanho total da mensagem
***Checksum*:** *Bytes* de controlo da integridade
***Message Type*:** Tipos possíveis da mensagem:

- `0` - Mensagem de Registo (usada pelo *Server* e *Agent*)
- `1` - Mensagem de uma *Task* (usada pelo *Server*)
- `2` - Mensagem de Resultados (usada pelo *Agent*)
- `3` - Mensagem de Métricas (usada pelo *Agent*)
- `4` - Mensagem de *Iperf* (usada pelo *Server*)
- `5` - Mensagem de Retransmissão (usada pelo *Server*)

***Sequence Number*:** Índice do fragmento em questão
***Sequence Length*:** Número de fragmentos total da *Task*
Data: *512 bytes*

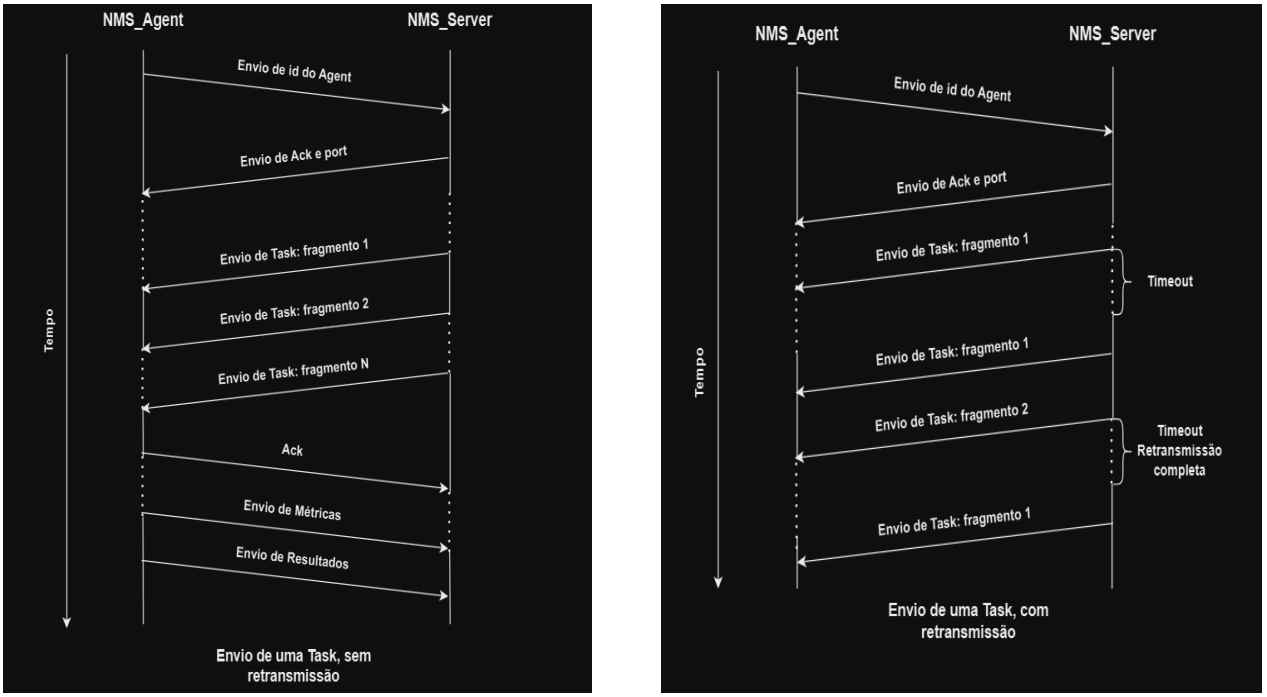

Figura 3. Exemplos de interações entre um determinado *NMS_Agent* e um *NMS_Server* de acordo com o *NetTask Protocol*

`SP` = Source Port | `DA` = Destination Address | `L` = length | `C` = checksum
`MT` = Message Type | `SN` = Sequence Number | `SL` = Sequence Length

Envio de id do Agente:

`SP` | `DA`(address inicial do Server) | `L` | `C` | `MT=0` | `SN` | `SL` (geralmente 1)

\+ *Id* do Agente (exemplo: "`n1`")

Envio de *Ack* e *port*:

`SP` | `DA` | `L` | `C` | `MT=0` | `SN` | `SL` (geralmente 1)

\+ *Port* único gerado para a comunicação com o agente (exemplo: `45311`)

Envio de *Task*:

`SP` | `DA` | `L` | `C` | `MT=1` | `SN` | `SL` (geralmente 2)

\+ A *task* atualmente a ser realizada

Envio de *Ack*:

`SP` | `DA`(*address* do *Server* com o novo *Port*) | `L` | `C` | `MT=1` | `SN` | `1`

\+ *Received*

Envio de Métricas:

`SP` | `DA`(*address* do *Server* com o novo *Port*) | `L` | `C` | `MT=3` | `SN` | `SL`

+ Métricas pedidas para medir

Envio de Resultados:

`SP|DA`(*address* do *Server* com o novo *Port*)`|L|C|MT=2|SN|SL`

+ Resultados da *Task* executada

Envio de *task* após *timeout*:(Indica para esquecer os pacotes antes recebidos)

`SP|DA`(*address* do *Server* com o novo *Port*)`|L|C|MT=5|SN|SL`(geralmente 2)

+ A *task* atualmente a ser realizada

3.2 AlertFlow Protocol

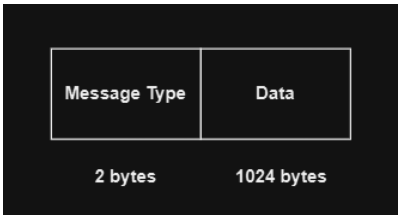

Figura 4. Formato da mensagem protocolar *AlertFlow*

Este protocolo possui 1 campo de cabeçalho, tendo *2 bytes* de tamanho.
O tamanho da mensagem é de *1024 bytes*, resultando num total de *1026 bytes*.

***Message Type*:** Tipos possíveis da mensagem:

- 1 - Mensagem de Começo de *AlertFlow*
- 2 - Mensagem de *AlertFlow*

Data: *1024 bytes*

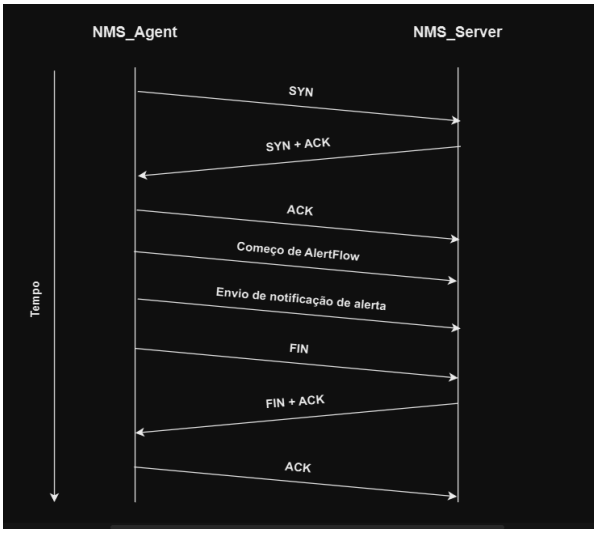

Figura 5. Diagrama de sequência do protocolo *AlertFlow*

`MT = Message Type`

Envio de mensagem de Começo de *AlertFlow*:

`MT = 1`

\+ Task id e device id

Envio de mensagem de de *AlertFlow*:

`MT = 2`

\+ Tempo atual, *cpu usage*, *ram usage*, *interface stats*, *packet loss* e *jitter*

4 Implementação - detalhes, parâmetros, bibliotecas de funções, entre outros

4.1 Arquitetura e Estrutura do Sistema

O sistema implementa um monitor de rede (NMS - Network Management System) que coordena tarefas entre um servidor central e múltiplos clientes distribuídos. Ele permite a coleta de métricas de desempenho de dispositivos e links, além de gerar alertas com base em condições predefinidas.

A comunicação utiliza os protocolos UDP e TCP para diferentes propósitos:

- UDP: Transmissão de tarefas e métricas (baixa sobrecarga, usa uma classe NMS_server_UDP para enviar e receber mensagens).
- TCP: Envio de alertas, garantindo confiabilidade.

4.2 Principais Componentes

1. **Servidor NMS**:
    - Gerencia tarefas definidas em JSON, controla a execução e processa os dados recebidos dos clientes.
    - Funções principais:
        - Configuração de sockets UDP e TCP.
        - Parsing do arquivo `tasks.json` para configurar tarefas.
        - Distribuição de tarefas e processamento de métricas.
        - Envio de mensagens para clientes e recebimento de dados.
2. **Clientes**:
    - Executam tarefas recebidas do servidor, monitoram métricas locais e enviam os dados para o servidor.
    - Funções principais:
        - Execução de comandos como `ping, iperf, e traceroute`.
        - Coleta de métricas (*CPU*, *RAM*, estatísticas de interface de rede).
        - Geração de alertas com base nas condições predefinidas.
3. **Configurações e JSON de Tarefas**:
    - Definido no arquivo `tasks.json`, contém:
        - Tipo de tarefa (ex.: `ping, traceroute`).
        - Dispositivos envolvidos.
        - Condições para alertas (uso de *CPU*/*RAM*, perda de pacotes, etc.).

4.3 Bibliotecas e Ferramentas

- **Bibliotecas Padrão**:
    - `socket`: Configuração de sockets *UDP* e *TCP*.
    - `struct`: Manipulação de mensagens binárias (encapsulamento de headers).
    - `subprocess`: Execução de comandos do sistema como `ping e iperf`.
    - `psutil`: Monitoramento de recursos locais (*CPU*, *RAM*, estatísticas de rede).
    - `threading`: Execução paralela de múltiplas tarefas.
    - `json`: Manipulação do formato *JSON* para tarefas e mensagens.
- **Ferramentas Externas**:
    - `iperf`: Teste de largura de banda.
    - `ping e traceroute`: Diagnóstico de rede.

5 Testes e Resultados

Com o objetivo de garantir a robustez e qualidade do nosso sistema, foram feitos vários testes com recurso a diferentes topologias ao longo da execução do trabalho prático. Numa fase inicial recorremos a topologias mais simples, mas que rapidamente tiveram a necessidade de

se tornarem mais complexas de modo a testar todos os requisitos que este trabalho prático exige. Para esta etapa, destacamos as duas seguintes topologias de rede utilizadas:

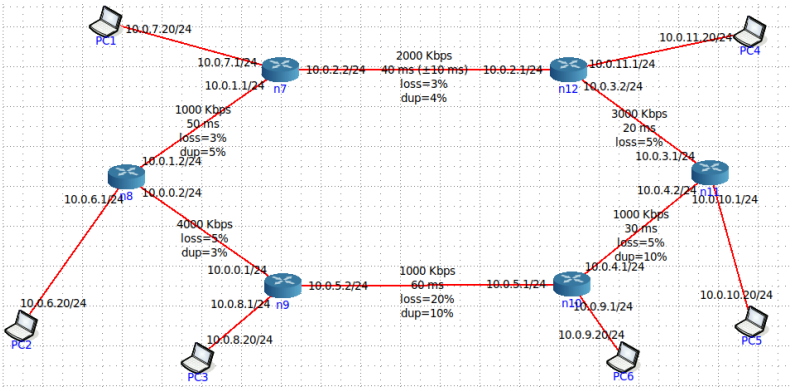

Figura 6. Topologia exemplo criada

PC1 corresponde a *n1*, *PC2* corresponde a *n2*, *PC3* corresponde a *n3*, *PC4* é aberto como servidor, *PC5* corresponde a *n4* e *PC6* corresponde a *n5*.

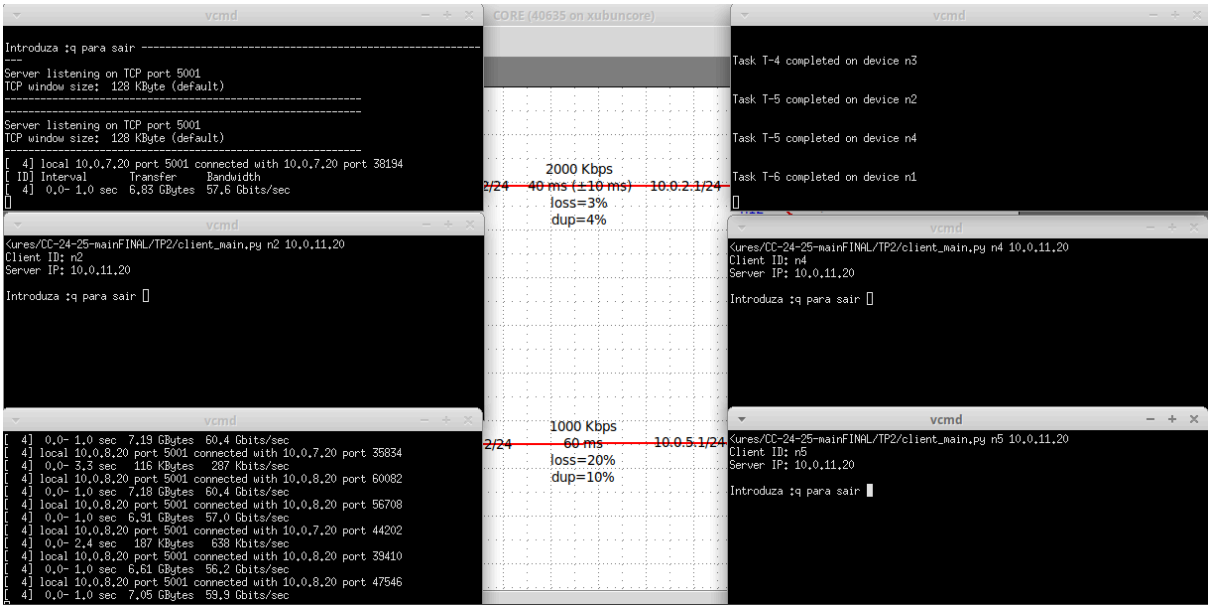

Figura 7. Resultados dos terminais de cada PC durante a execução (servidor no canto superior direito)

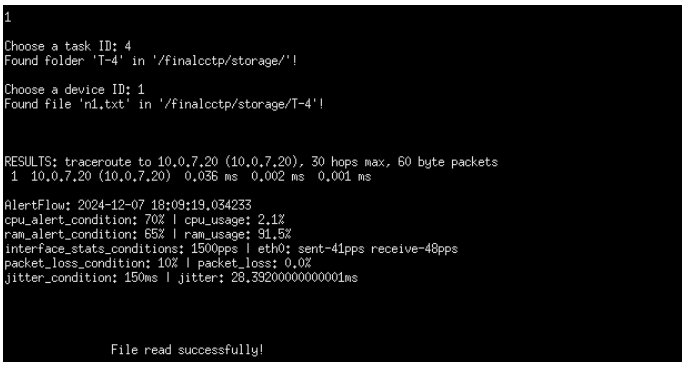

Figura 8. Consulta dos resultados das tasks em tempo real através do Menu

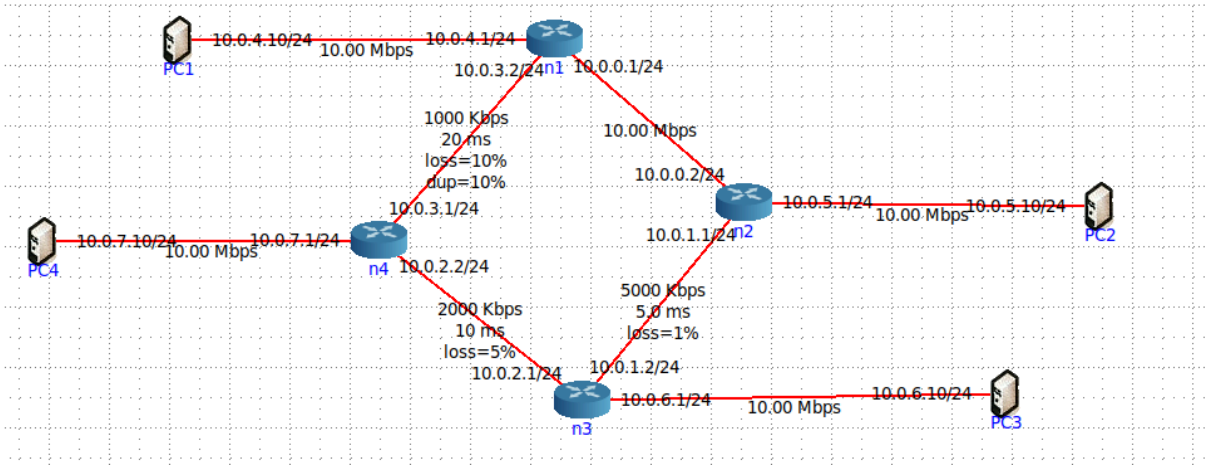

Figura 9. Topologia `CC-Topo-2024.imn`

PC1 corresponde a *n1*, *PC2* corresponde a *n2*, *PC3* corresponde a *n3* e *PC4* é aberto como servidor.

```
<C-24-25-mainFINAL/TP2/server_main.py /finalcctp2425
Storage_path ajustado: /finalcctp2425/storage/


                Welcome!

                [1]- Choose a task ID
                [0]- Quit


Choose an option:
Task T-1 completed on device n1


Task T-3 completed on device n3
```

Figura 10. Terminal do servidor após ter iniciado e tasks já estiverem sido feitas

PC4 é aberto como servidor e indica em tempo real o *id* da *task* e o respetivo *device* onde a task já foi realizada.

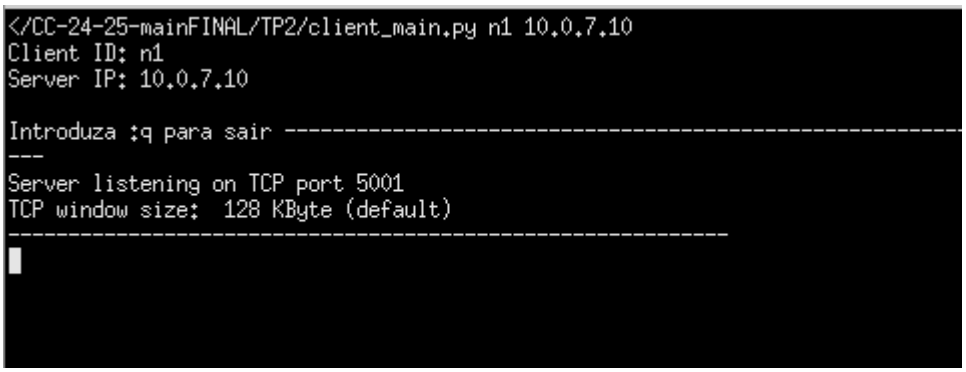

Figura 11. Terminal do lado do cliente (*PC1*)

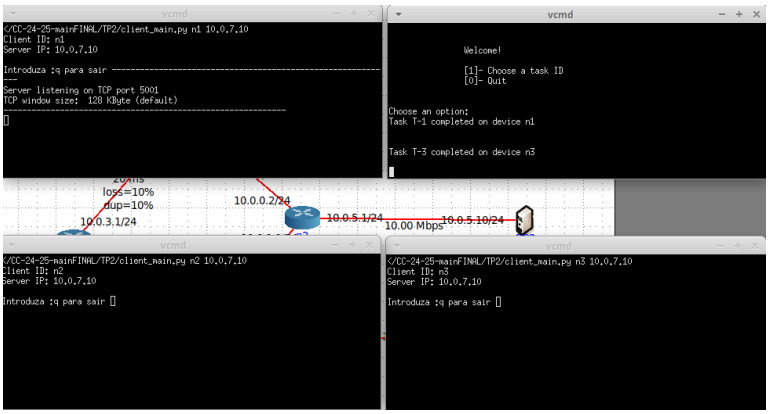

Figura 12. Terminais de cada um dos devices durante a execução (servidor no canto superior direito)

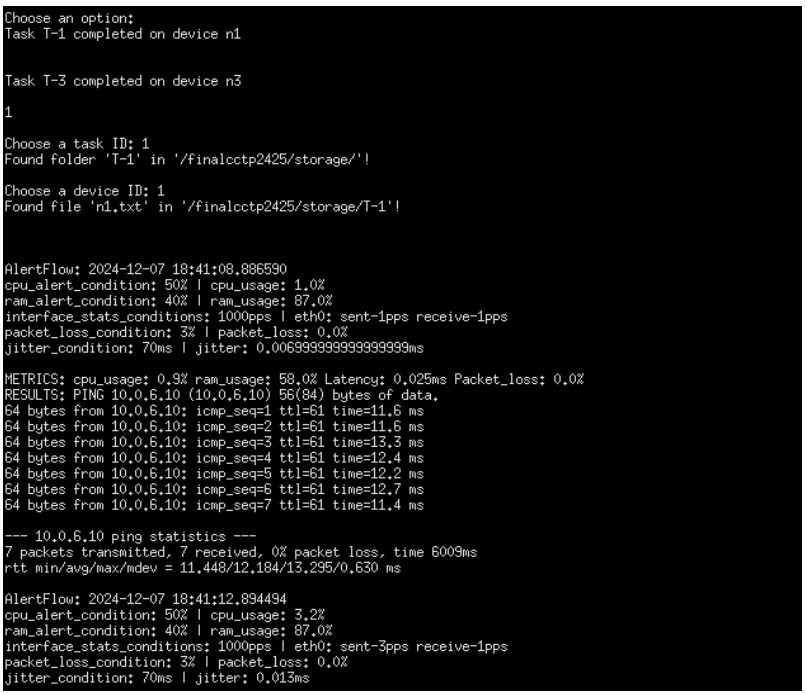

Figura 13. Consulta do resultado da *task* através das interações com o *Menu (métricas contidas)*

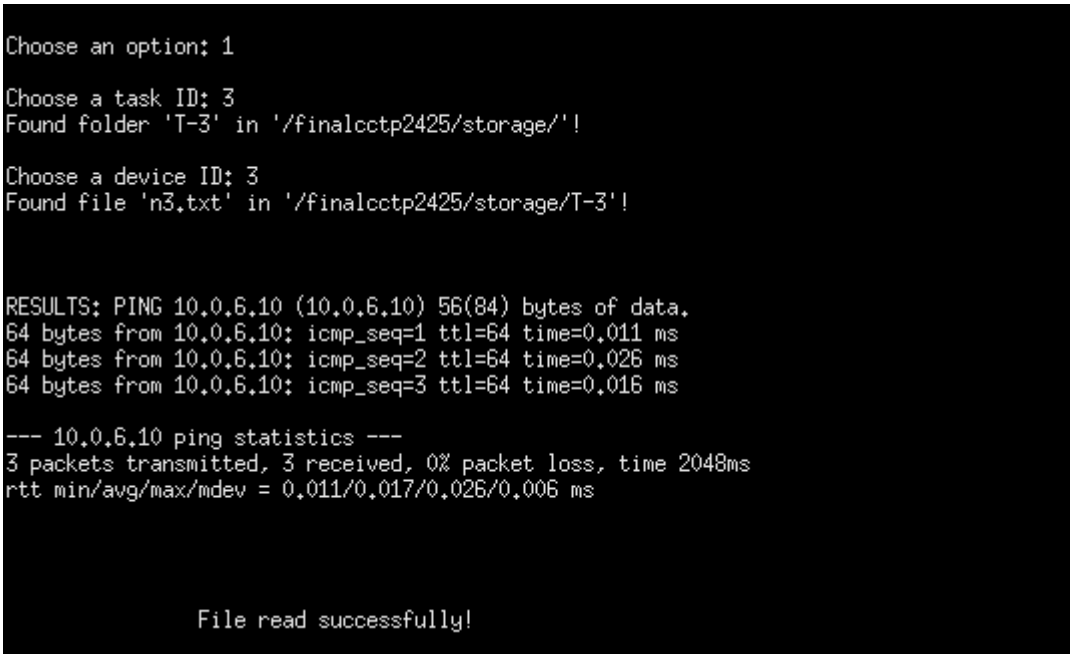

Figura 14. Consulta do resultado da *task* através das interações com o *Menu* (consulta do comando *ping*)

6 Conclusão e trabalho futuro

O desenvolvimento do sistema de monitorização de redes (*NMS*) foi bem-sucedido, com a implementação eficaz dos protocolos *NetTask* (*UDP*) e *AlertFlow* (*TCP*). O *NetTask* permitiu a coleta contínua de métricas de rede e *hardware* e o envio periódico dessas informações ao *NMS_Server*, enquanto o *AlertFlow* garantiu a notificação de momentos críticos. A utilização de *UDP* no protocolo *NetTask* exigiu a implementação de mecanismos de fragmentação e retransmissão para assegurar a entrega confiável dos dados em condições de rede com falhas. O sistema foi validado com sucesso através de variados testes realizados no emulador *CORE* com recurso a diferentes topologias, demonstrando a sua capacidade de detetar falhas de rede e disparar alertas adequados nos mais variados casos.

Futuramente, o sistema pode ser otimizado para suportar de maneira eficiente um número maior de *NMS_Agents*, especialmente em redes de maior escala, com a introdução de técnicas de balanceamento de carga, de modo a garantir que a qualidade e integridade não sejam afetadas. Na área técnica, achamos que seria pertinente, em fases futuras, adicionar controlo de fluxo ao protocolo *NetTask* e melhorar o processo de retransmissão, permitindo a retransmissão de um pacote específico. A análise de desempenho em tempo real e a validação em cenários de rede mais complexos também são áreas que podem ser exploradas para melhorar a escalabilidade e robustez do sistema.