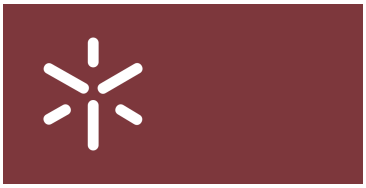


**Universidade do Minho**
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

## Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

# Monitorização de Rede

**Pedro Gomes** a104540
**André Ribeiro** a104436
**David Costa** a100003

Dezembro, 2024

# Índice

# 1. Introdução

## 1.1. Contextualização

O trabalho prático insere-se no contexto da gestão de redes de computadores, uma área fundamental em sistemas modernos que requer monitorização contínua para garantir a operação eficiente e a identificação precoce de problemas. A solução proposta visa responder a este desafio através de uma aplicação distribuída que utiliza um modelo cliente-servidor para recolher, monitorizar e relatar métricas críticas de dispositivos de rede.

## 1.2. Apresentação do Caso de Estudo

O caso de estudo envolve o desenvolvimento de um sistema de monitorização de redes (**Network Monitoring System - NMS**), constituído por dois componentes principais: o **"NMS_Agent"**, que recolhe métricas e envia relatórios, e o **"NMS_Server**"", que processa e apresenta os dados recebidos. Os protocolos aplicacionais **"NetTask"** (baseado em **UDP**) e **"AlertFlow"** (baseado em **TCP**) serão desenvolvidos para permitir uma comunicação eficiente e resiliente entre os dois componentes, garantindo a deteção e notificação de anomalias críticas.

## 1.3. Motivação e Objetivos

A motivação para este trabalho está na crescente complexidade e dependência de redes de computadores, onde a falha de componentes pode ter impactos severos. O trabalho propõe:

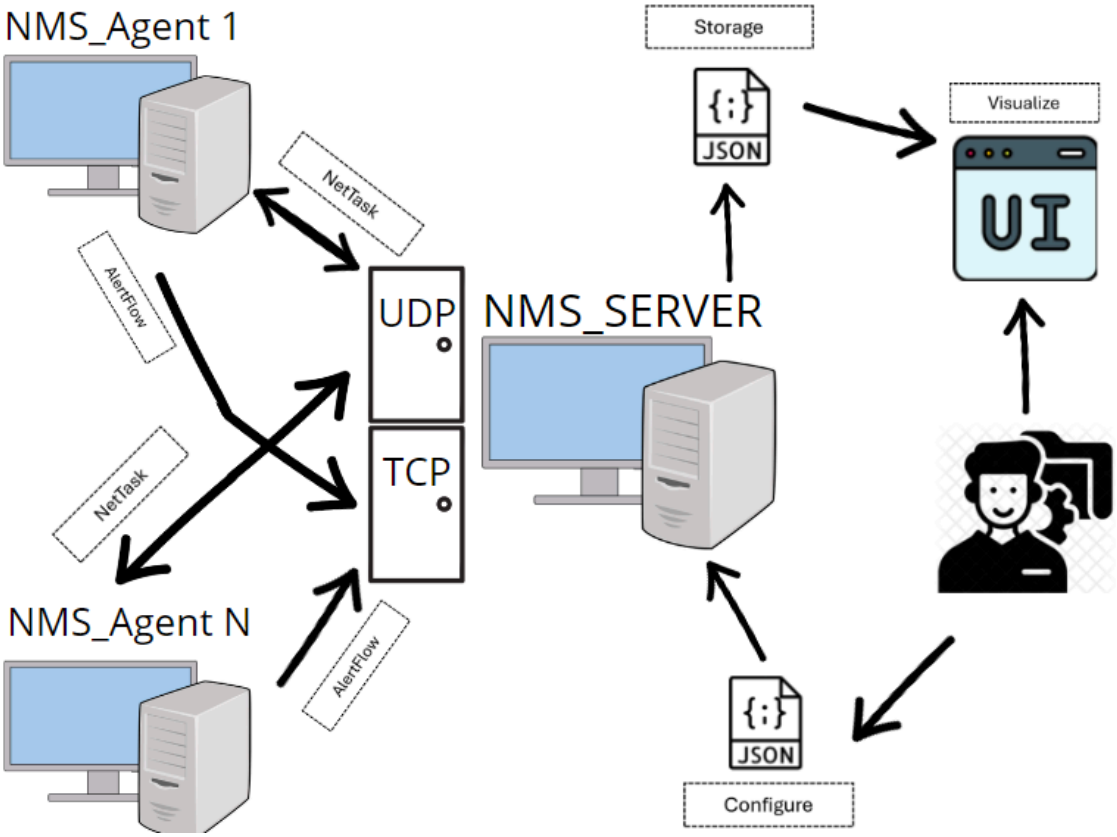
- Promover uma ferramenta robusta para monitorização contínua de redes.
- Permitir a deteção precoce de anomalias e eventos críticos.
- Explorar o design e a implementação de protocolos aplicacionais resilientes. Os objetivos específicos incluem:
  - Coletar métricas de rede de forma eficiente e apresentá-las de maneira estruturada.
  - Desenvolver protocolos para comunicação entre agentes e servidor, utilizando sockets **TCP** e **UDP**.
  - Implementar mecanismos opcionais de controlo de fluxo e retransmissão no protocolo **UDP**.
  - Simular cenários de teste em um ambiente controlado com o emulador **CORE**.

## 1.4. Estrutura do Relatório

O relatório técnico segue a estrutura proposta:

1. Introdução - Apresenta o contexto, motivação e objetivos do trabalho.
2. Arquitetura da Solução - Descreve o design geral do sistema e os seus componentes principais.
3. Especificação dos Protocolos - Detalha o formato das mensagens e a lógica de comunicação.
4. Implementação - Explica a abordagem de desenvolvimento, incluindo bibliotecas.
5. Testes e Resultados - Apresenta os cenários de teste, dados recolhidos e análises.
6. Conclusões e Trabalho Futuro - Resume os resultados e sugere melhorias para o sistema.

# 2. Arquitetura da solução




A arquitetura proposta consiste em uma solução distribuída onde o **"NMS_Server"** e os **"NMS_Agents"** se comunicam usando dois protocolos: **"NetTask" (UDP)** e **"AlertFlow" (TCP)**. O servidor é o gerenciador desta rede, apenas enviando tarefas e sendo o único que manipula a **"storage"**, enquanto os agentes executam essas mesmas tarefas e coletam as métricas de rede.

A **"storage"** do sistema é baseada em ficheiros **JSON**, que armazenam tanto as métricas coletadas por cada agente quanto os alertas associados a cada métrica. O **UI (User Interface)** é projetado para "consumir" esses ficheiros **JSON**, sem interação direta com o servidor ou outros sistemas, apenas exibindo as informações formatadas a partir desses arquivos. Isso simplifica a interface, mas requer que os dados estejam sempre atualizados e corretamente estruturados no formato **JSON** para garantir uma visualização adequada.

# 3. Especificação dos Protocolos

## 3.1. NetTask Protocol

O protocolo **“NetTask”** utiliza a camada de transporte UDP para realizar a comunicação eficiente e resiliente entre o servidor central (**“NMS_Server”**) e os agentes distribuídos (**“NMS_Agents”**) no sistema de monitorização de redes. Este protocolo é responsável pela gestão de tarefas de monitorização e pela transmissão contínua de métricas recolhidas pelos agentes, incorporando mecanismos como retransmissões para assegurar confiabilidade, em caso de perda de pacotes.

O **“NetTask”** opera através de quatro tipos de pacotes definidos no campo **“Packet Type”**:

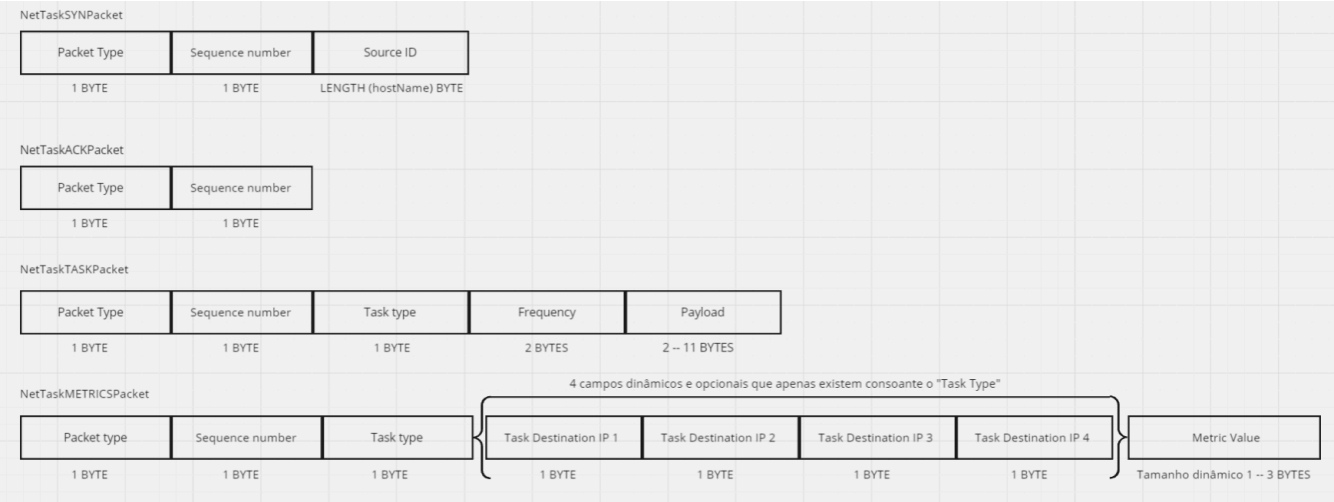
1. **SYN**: Este pacote é utilizado para o registo inicial de um agente no servidor, identificando-o de forma única e iniciando a comunicação entre ambos.
2. **ACK**: Pacote de reconhecimento, enviado como resposta a outros tipos de pacotes para confirmar a receção bem-sucedida das mensagens e garantir a sincronização na comunicação.
3. **TASK**: Enviado pelo servidor para os agentes, contém informações detalhadas sobre as tarefas a serem realizadas, incluindo métricas específicas a monitorizar, frequência de coleta e condições associadas.
4. **METRICS**: Pacote utilizado pelos agentes para enviar os resultados das tarefas de monitorização realizadas, reportando as métricas coletadas ao servidor para análise e armazenamento.

Além disso, o NetTask inclui mecanismos para retransmissão de pacotes caso a entrega falhe, reforçando a confiabilidade da comunicação. O propósito principal do protocolo é assegurar que as tarefas de monitorização sejam atribuídas e executadas de forma eficiente e que os dados relevantes sejam transmitidos periodicamente, mesmo em condições de rede adversas. O uso do UDP permite uma comunicação rápida e leve, enquanto mecanismos como números de sequência, ACKs e retransmissões garantem a entrega confiável das mensagens. Este protocolo suporta a coleta contínua de dados de desempenho da rede, servindo como base para decisões operacionais e manutenção proativa.

| Packet Type | Identificador |
|---|---|
| SYN | 000 |
| ACK | 001 |
| TASK | 010 |
| METRICS | 011 |

Cada tipo de pacote é distinguido através de identificadores numéricos de forma a obter um tamanho de pacote mais otimizado.

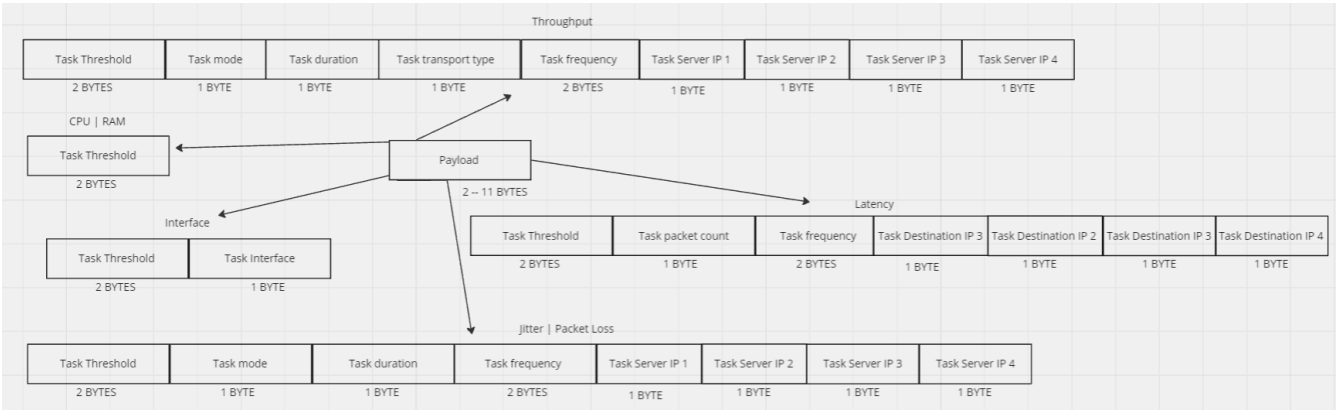
### 3.1.1. Especificação dos pacotes




#### 3.1.1.1. Pacote Task NetTask

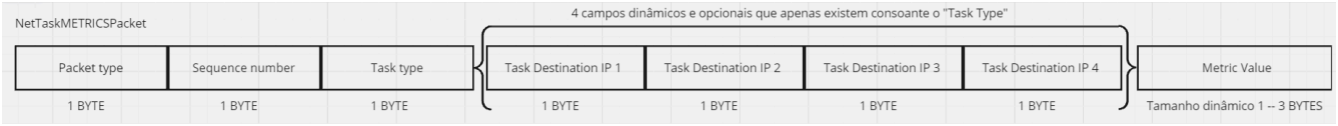
| Task Type | Identificador |
|---|---|
| CPU | 000 |
| RAM | 001 |
| Throughput | 010 |
| Latency | 011 |
| Interface | 100 |
| Jitter | 101 |
| Packet Loss | 110 |

Cada pacote do tipo **“Task”** é distinguido através de identificadores numéricos de forma a obter um tamanho de pacote mais otimizado.




De denotar que o campo **“Payload”** terá tamanhos variáveis de acordo com o tipo de **“Task”** que estiver a ser enviada.

### 3.1.1.2. Pacote Metrics NetTask




O pacote do tipo **“Metrics”** merece uma atenção especial pois varia bastante de acordo com o tipo de **“Task”** que o originou e de acordo o valor da métrica a ser enviada.

```python
class NetTaskMETRICSPacket:
    def __init__(self, packet_type: int, seq_num: int, task_type: int, task_server_ip_1: int, task_server_ip_2: int, task_server_ip_3: int, task_server_ip_4: int, metric_value: int):
        # Validate inputs
        if packet_type >= 8:
            raise ValueError("packet_type must be less than 8 (fits in 3 bits)")
        if seq_num >= 256:
            raise ValueError("seq_num must be less than 256 (fits in 8 bits)")
        if task_type >= 8:
            raise ValueError("task_type must be less than 8 (fits in 3 bits)")

        self.packet_type = packet_type
        self.seq_num = seq_num
        self.task_type = task_type
        self.task_server_ip_1 = task_server_ip_1
        self.task_server_ip_2 = task_server_ip_2
        self.task_server_ip_3 = task_server_ip_3
        self.task_server_ip_4 = task_server_ip_4
        self.metric_value = metric_value

    def to_bytes(self) -> bytes:
        self.seq_num += 1
        packet = struct.pack("!B", self.packet_type)
        packet += struct.pack("!B", self.seq_num)
        packet += struct.pack("!B", self.task_type)
        if (not (self.task_server_ip_1 == 0 and self.task_server_ip_2 == 0 and self.task_server_ip_3 == 0 and self.task_server_ip_4 == 0)):
            packet += struct.pack("!B", self.task_server_ip_1)
            packet += struct.pack("!B", self.task_server_ip_2)
            packet += struct.pack("!B", self.task_server_ip_3)
            packet += struct.pack("!B", self.task_server_ip_4)
        # Encode metric_value
        if self.metric_value <= 255:  # Fits in 1 bytes
            packet += struct.pack("!B", self.metric_value)
        elif self.metric_value <= 65535:  # Fits in 2 bytes
            packet += struct.pack("!H", self.metric_value)
        else:  # Requires 3 bytes
            packet += struct.pack("!I", self.metric_value)[1:]  # Take the last 3 bytes

        return packet
```

**“Task”**s que não necessitaram de qualquer IP de destino/IP de servidor, irão originar pacotes de **“Metrics”** que não incluirão os campos de endereço IP no método de **“encoding”**, (essa verificação é feita quando todos os atributos de endereço do servidor têm o valor 0). Para além disso, de acordo com o valor da métrica, o processo de **“encoding”** avalia a necessidade de usar 1, 2 ou 3 **“bytes”**.

```python
@classmethod
def from_bytes(cls, data: bytes):
    # Ensure the input data length is sufficient
    if len(data) < 4:  # Minimum required length for packet_type, seq_num, task_type, and metric_value
        raise ValueError("Insufficient data to decode NetTaskMETRICSPacket")
    # Decode fields in the same order as `to_bytes`
    offset = 0
    packet_type = struct.unpack("!B", data[offset:offset + 1])[0]
    offset += 1
    seq_num = struct.unpack("!B", data[offset:offset + 1])[0]
    offset += 1
    task_type = struct.unpack("!B", data[offset:offset + 1])[0]
    offset += 1

    remaining_length = len(data) - offset
    if remaining_length >= 5:
        task_server_ip_1 = struct.unpack("!B", data[offset:offset + 1])[0]
        offset += 1
        task_server_ip_2 = struct.unpack("!B", data[offset:offset + 1])[0]
        offset += 1
        task_server_ip_3 = struct.unpack("!B", data[offset:offset + 1])[0]
        offset += 1
        task_server_ip_4 = struct.unpack("!B", data[offset:offset + 1])[0]
        offset += 1

        # Decode metric_value
        remaining_length = len(data) - offset
        if remaining_length == 1:  # 1-byte metric_value
            metric_value = struct.unpack("!B", data[offset:offset + 1])[0]
            offset += 1
        elif remaining_length == 2:  # 2-byte metric_value
            metric_value = struct.unpack("!H", data[offset:offset + 2])[0]
            offset += 2
        elif remaining_length == 3:  # 3-byte metric_value
            metric_value = struct.unpack("!I", b'\x00' + data[offset:offset + 3])[0]
            offset += 3
        else:
            raise ValueError("Invalid metric_value length in data")
        # Return a new instance of NetTaskMETRICSPacket
        return cls(packet_type, seq_num, task_type, task_server_ip_1, task_server_ip_2, task_server_ip_3, task_server_ip_4, metric_value)
    else:
        # Decode metric_value
        remaining_length = len(data) - offset
        if remaining_length == 1:  # 1-byte metric_value
            metric_value = struct.unpack("!B", data[offset:offset + 1])[0]
            offset += 1
        elif remaining_length == 2:  # 2-byte metric_value
            metric_value = struct.unpack("!H", data[offset:offset + 2])[0]
            offset += 2
        elif remaining_length == 3:  # 3-byte metric_value
            metric_value = struct.unpack("!I", b'\x00' + data[offset:offset + 3])[0]
            offset += 3
        else:
            raise ValueError("Invalid metric_value length in data")
        # Return a new instance of NetTaskMETRICSPacket
        return cls(packet_type, seq_num, task_type, 0, 0, 0, 0, metric_value)
```

Já na chamada do método de **“decoding”**, tanto a existência de campos relativos ao endereço IP como o tamanho do campo do valor da métrica é averiguada através de quantos **“bytes”** restam para ler na sequência de **“bytes”** que está a ser **“decoded”**.

## 3.2. AlertFlow Protocol

O protocolo **“AlertFlow”** é utilizado para notificar alterações críticas nas métricas monitorizadas em uma rede. Ele opera sobre a camada de transporte **TCP**, garantindo a entrega confiável das mensagens.

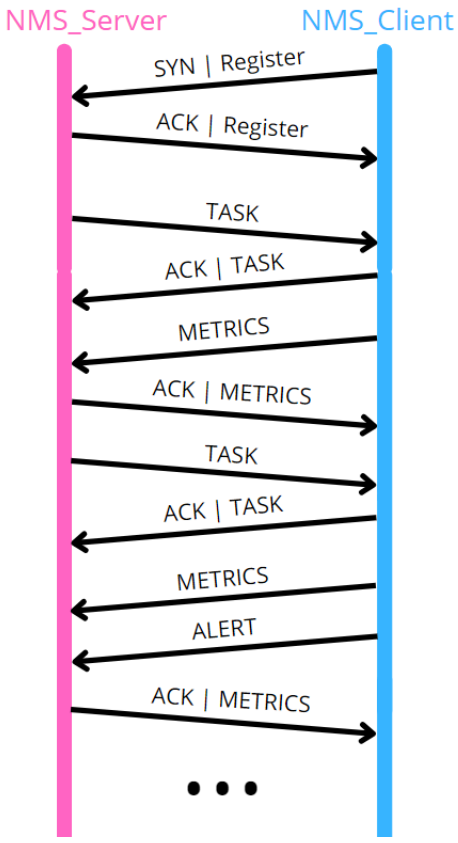
Seu propósito é assegurar que, ao identificar uma alteração significativa (como falhas em interfaces de rede, variações extremas de latência, perda de pacotes acima de limites predefinidos, ou alto uso de **CPU/RAM**), o agente responsável pela monitorização (**“NMS_Agent”**) envie uma notificação ao servidor central (**“NMS_Server”**) para que possam ser, posteriormente, tomadas as medidas necessárias para resolver e perceber o que está a afetar a rede nesse sentido. Assim, ele complementa o monitoramento contínuo realizado pelo protocolo **“NetTask”**, atuando em situações críticas.

AlertFlowPacket

| Packet type | Sequence number | Task type | Metric Value |
|---|---|---|---|
| 1 BYTE | 1 BYTE | 1 BYTE | Tamanho dinâmico 1 -- 3 BYTES |

Sobre o pacote que é enviado pelo protocolo, é seguida uma lógica idêntica de **“encoding”** e **“decoding”** relativamente ao pacote de **“Metrics”** do protocolo **“NetTask”**.

## 3.3. Comunicação - Diagrama temporal




O diagrama temporal vem confirmar de forma mais analítica o propósito dos protocolos. A grande maioria das mensagens são enviadas pelo protocolo **“NetTask”** sobre a camada de transporte **UDP** e apenas os Alertas são enviados pelo protocolo **“AlertFlow”** sobre a camada de transporte **TCP**. Vale relembrar que esta cronologia de mensagens foi feita para o que é esperado acontecer da forma que a comunicação está codificada e pode não acontecer, necessariamente, como descrito no diagrama.

# 4. Implementação

O projeto está dividido em dois componentes principais: **"NMS_Server"** e **"NMS_Agent"**, com funcionalidades específicas implementadas em **Python**.

## 4.1. NMS_Server:

- Coordena agentes distribuídos e armazena métricas em ficheiros **JSON**.
- Gera e envia tarefas aos agentes utilizando o protocolo **"NetTask"**.
- Recebe métricas e alertas de agentes através dos protocolos **"NetTask**" e **"AlertFlow"**, respetivamente.
- Implementa retransmissões para tarefas e pacotes críticos que não tenham sido confirmados por **ACKs**.
- Utiliza **threads** para lidar com múltiplos agentes simultaneamente.

## 4.2. NMS_Agent:

- Regista-se no servidor usando **"NetTask"**.
- Recebe tarefas, executa métricas (CPU, RAM, interfaces, latência, jitter, packet loss e throuput), e reporta resultados periodicamente.
- Gera alertas via **"AlertFlow"** quando condições críticas de métricas estabelecidas previamente no ficheiro de configuração de tarefas, são atingidas.
- Implementa retransmissões para pacotes de métricas caso o servidor não envie **ACKs**.

## 4.3. Parâmetros e Configurações

#### 4.3.1. NMS_Server:

- Porta **UDP** padrão: 6000.
- Porta **TCP** padrão: 5001.
- Diretoria para armazenamento de métricas: **JSON**, gerenciado por **ResultsDatabase.DatabaseHandler**.
- **Threads** dedicadas para comunicação **UDP** e **TCP**, além de retransmissões específicas para tarefas e pacotes não confirmados.

#### 4.3.2. NMS_Agent:

- Porta **UDP** de origem: 6969 (fixa).
- Portas do servidor configuráveis via argumentos (**UDP** e **TCP**).
- Execução de tarefas configurada com base nos pacotes recebidos, como frequência de coleta e limites de alerta.

#### 4.3.3. Bibliotecas Utilizadas

##### 4.3.3.1. Bibliotecas Padrão do Python:

- **socket**: Para comunicação via **TCP** e **UDP**.
- **threading**: Para execução de **threads** simultâneas no servidor e no agente.
- **time**: Para implementação de **timeouts** e retransmissões.
- **argparse**: Para interpretação de argumentos na inicialização do agente.
- **os** e **json**: Para gerenciamento de ficheiros e manipulação de dados **JSON**.

##### 4.3.3.2. Bibliotecas Externas:

- **psutil**: Para monitorização do uso de **CPU** e **RAM**.
- **utils**: Funções utilitárias para conversões de IP, execução de comandos de sistema (**ping**, **iperf**), e **manipulação de interfaces de rede**.

#### 4.3.4. Estrutura de Comunicação

Protocolo **"NetTask"**:

- **UDP**: Utilizado para troca de mensagens entre o servidor e os agentes.
- Pacotes implementados:
  - **SYN**: Registo do agente no servidor.
  - **TASK**: Envio de tarefas aos agentes.
  - **METRICS**: Envio de métricas coletadas pelos agentes.
  - **ACK**: Confirmação de recebimento de pacotes.

Protocolo **"AlertFlow"**:

- **TCP**: Utilizado para envio de alertas de condições críticas do agente para o servidor.
  - Implementado para garantir confiabilidade na entrega das mensagens.

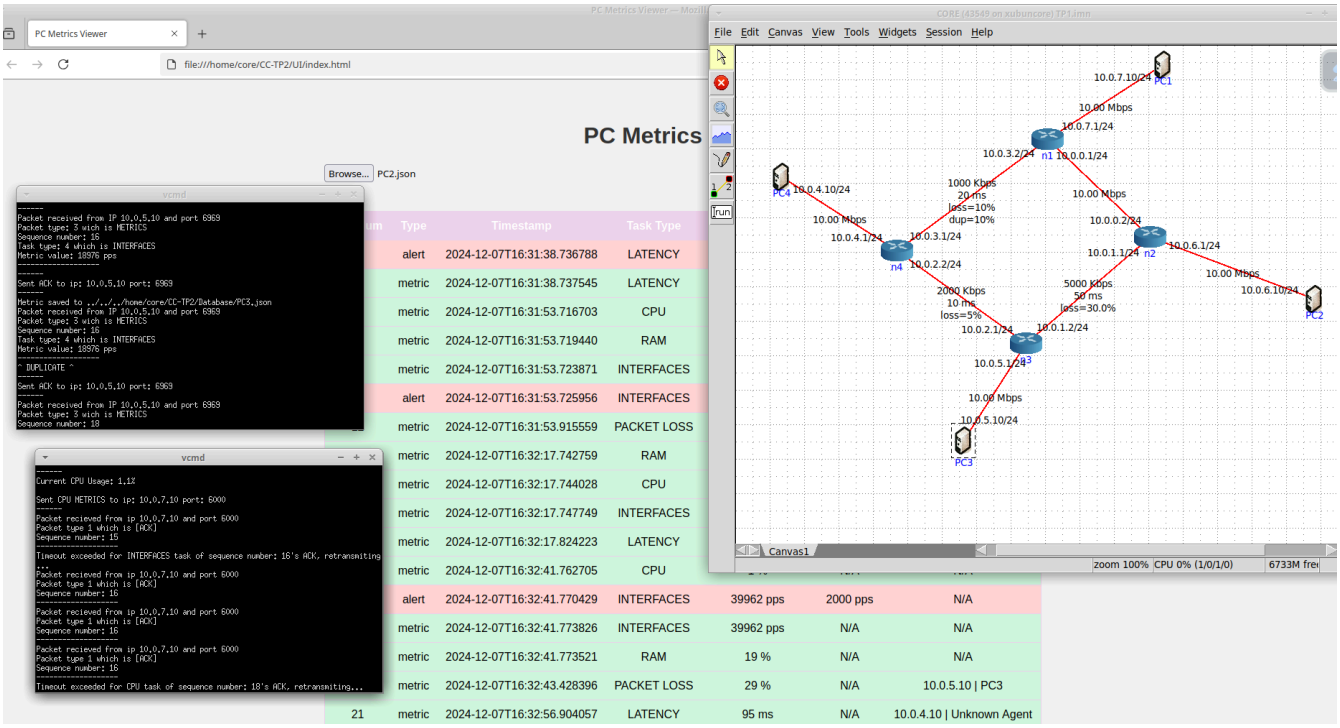
Detalhes da Implementação

- Servidor:
  - Implementa duas **threads** principais para comunicação:
    - **UDP Listener**: Para registo de agentes, envio de tarefas e receção de métricas.
    - **TCP Listener**: Para receção de alertas críticos.
  - Cada tarefa ou métrica é gerenciada por **threads** dedicadas, garantindo **paralelismo** e **escalabilidade**.
  - Implementa **retransmissões** com até 5 tentativas em caso de falha (**timeouts**).
- Agente:
  - Gerencia tarefas em paralelo utilizando **threads**.
  - Executa métricas específicas com funções dedicadas (CPU, RAM, latência, jitter, throughput, etc.).
  - Envia alertas em tempo real ao servidor caso as métricas ultrapassem os limites configurados.

Fluxo de Trabalho Resumido

- O agente regista-se no servidor utilizando o pacote **SYN** (**"NetTask"**).
- O servidor confirma o registo enviando um pacote **ACK**.
- O servidor envia tarefas para o agente utilizando o pacote **TASK**.
- O agente executa a tarefa e envia as métricas coletadas utilizando o pacote **METRICS**.
- Caso as métricas excedam os limites configurados, o agente envia um alerta utilizando o protocolo **"AlertFlow"**.
- **Retransmissões** ocorrem automaticamente em ambos os lados se os **ACKs** não forem recebidos.

# 5. Testes e resultados




Como podemos ver através dos “logs”, e devido às configurações da topologia, nomeadamente na ligação pela qual o PC2 e o PC3 enviam pacotes entre si, estão a acontecer retransmissões, seja por perda efetiva de pacotes ou por largos atrasos de rede (latência). Para além disso tanto o servidor como os agentes estão preparados para lidar com pacotes duplicados. O nosso UI mostra-nos também alguns resultados provenientes deste teste, onde se verifica a veracidade das configurações da rede, por exemplo na recolha de métricas de **“packet loss”**, resultante do **iperf** realizado no sentido PC2 -> PC3, (29%) ou os elevados valores de latência na direção PC2 -> PC4 registados (95ms).

# 6. Conclusões e Trabalho Futuro

## 6.1. Conclusões

O trabalho prático resultou no desenvolvimento bem-sucedido de um sistema de monitorização de redes (**Network Monitoring System - NMS**) robusto e eficiente, composto pelos componentes **"NMS_Server"** e **"NMS_Agent"**. A arquitetura distribuída e os protocolos aplicacionais implementados garantem comunicação confiável e eficiente, permitindo a deteção e notificação de anomalias críticas na rede.

**Pontos fortes do sistema:**

- Modularidade e Escalabilidade: A solução distribui responsabilidades entre o servidor e os agentes, possibilitando a integração de novos agentes com facilidade.
- Confiabilidade: A implementação de retransmissões e mecanismos para lidar com pacotes duplicados assegurou a entrega confiável de dados, mesmo em cenários de alta latência ou perdas de pacotes.
- Flexibilidade: O uso de ficheiros **JSON** para configuração e armazenamento de métricas simplifica a integração com o UI e permite adaptação a novos cenários de monitorização.
- Resiliência: A deteção e notificação de condições críticas em tempo real proporcionam uma base sólida para ações proativas na gestão de redes.

Apesar dos sucessos alcançados, algumas limitações foram identificadas, destacando oportunidades para melhoria.

Pontos a melhorar:

- Latência em redes congestionadas: Embora o sistema lide bem com retransmissões, em redes altamente congestionadas, pode haver atrasos significativos.
- Carga de processamento: O uso intensivo de **threads** para tarefas paralelas pode impactar o desempenho em ambientes com muitos agentes ativos.

## 6.2. Trabalho Futuro

**Extensões e Melhorias Técnicas:**

- Otimização do sistema de retransmissões: Implementar algoritmos adaptativos para ajustar dinamicamente o número de retransmissões e os intervalos de **timeout**, reduzindo impacto em redes congestionadas.
- Melhoria no uso de recursos: Explorar a utilização de reutilização de **threads** para otimizar o desempenho em cenários com alta carga de agentes.

## 6.3. Finalização

Em suma, o trabalho realizado representa um avanço significativo na monitorização de redes, entregando uma solução funcional e modular. As propostas de trabalho futuro visam ampliar a capacidade, resiliência e aplicabilidade do sistema, alinhando-o com as necessidades crescentes de redes modernas.