



Universidade do Minho  
Escola de Engenharia  
Licenciatura em Engenharia Informática

## Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

# Monitorização de Redes

Rodrigo Granja (a104531)    Gonçalo Alves (a104079)    Duarte Faria (a95609)

6 de dezembro de 2024

CC

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Arquitetura da solução</b>	<b>2</b>
<b>3</b>	<b>Especificação dos protocolos propostos</b>	<b>3</b>
3.1	Formato das mensagens protocolares . . . . .	4
3.2	Diagrama de sequência . . . . .	5
<b>4</b>	<b>Implementação</b>	<b>6</b>
4.1	Bibliotecas . . . . .	6
4.2	Detalhes do servidor . . . . .	6
4.3	Detalhes dos Agentes . . . . .	8
4.4	Estrutura de mensagens . . . . .	10
4.5	Data Storage e Parser . . . . .	11
<b>5</b>	<b>Testes e Resultados</b>	<b>12</b>
5.1	Tarefa CPU . . . . .	12
5.2	Tarefa RAM . . . . .	12
5.3	Tarefa Latency . . . . .	12
5.4	Tarefa Bandwidth . . . . .	12
5.5	Tarefa Jitter . . . . .	13
5.6	Tarefa Packet Loss . . . . .	13
5.7	Tarefa Packet Loss . . . . .	13
5.8	Tarefa Interfaces/Packets Per Second . . . . .	13
<b>6</b>	<b>Conclusões e Trabalho Futuro</b>	<b>15</b>

# 1 Introdução

A crescente complexidade das redes modernas exige soluções eficientes para monitorizar e gerir métricas críticas em tempo real. Este trabalho apresenta o desenvolvimento de um Sistema de Monitorização de Redes (NMS), projetado para identificar anomalias e responder rapidamente a condições críticas. Baseado em uma arquitetura cliente-servidor distribuída, o sistema utiliza agentes distribuídos para monitorizar recursos e um servidor central para coordenação e armazenamento.

A comunicação no sistema é gerida por dois protocolos dedicados:

- **Nettask (UDP):** utilizado para envio de tarefas e coleta de métricas, priorizando eficiência e baixa latência.
- **Alertflow (TCP):** destinado ao envio de alertas críticos, garantindo maior confiabilidade na transmissão.

O sistema foi implementado e testado num ambiente emulado com o CORE, utilizando a topologia definida no primeiro trabalho prático. Este documento detalha a arquitetura da solução, os protocolos implementados e as interações entre os diferentes componentes.

## 2 Arquitetura da solução

A solução proposta apresenta uma arquitetura modular que separa claramente as funções do servidor e dos agentes. O servidor central é responsável pela coordenação de agentes, envio de tarefas, processamento de métricas e alertas, e armazenamento de dados. Os agentes são responsáveis por monitorizar recursos, executar tarefas e reportar dados ao servidor (métricas e alertas).

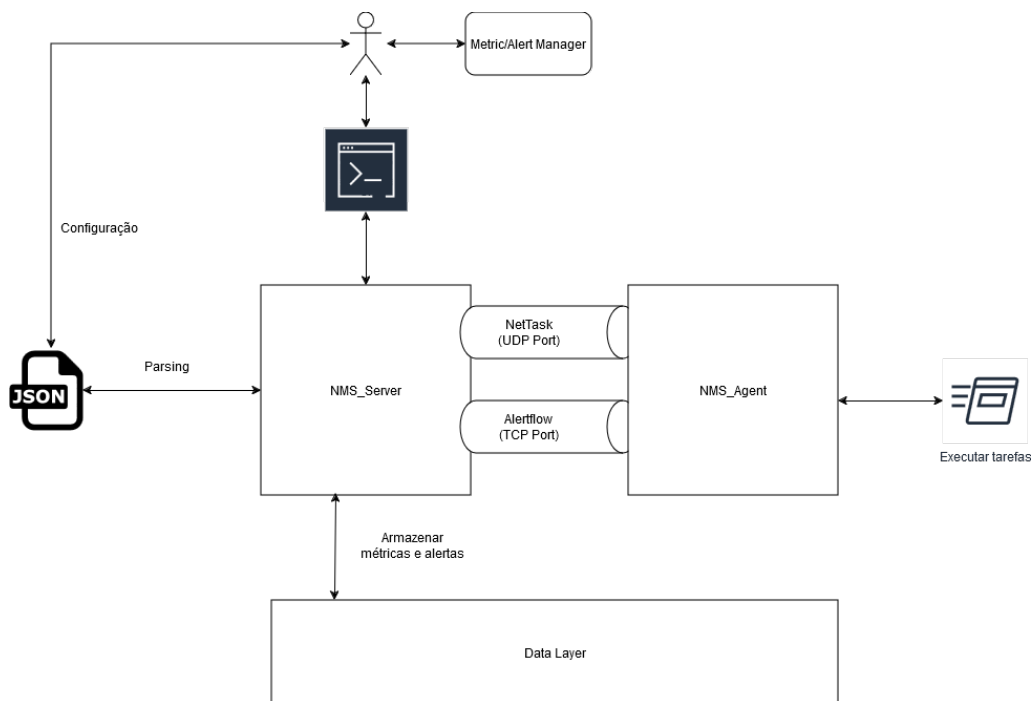


Figura 2.1: Arquitetura da solução

### 3 Especificação dos protocolos propostos

Os protocolos propostos são responsáveis por gerir a comunicação entre o servidor e os agentes, garantindo o registo, a execução de tarefas, a coleta de métricas e o envio de alertas. Eles utilizam mensagens compactas para minimizar a sobrecarga de rede, empregando números de sequência para garantir confiabilidade e evitar duplicações. A comunicação regular ocorre via UDP, enquanto mensagens críticas, como alertas, são transmitidas via TCP para maior confiabilidade. O protocolo suporta múltiplos tipos de tarefas e métricas, sendo flexível para atender diferentes cenários e capaz de detetar condições críticas definidas por *thresholds*.

### 3.1 Formato das mensagens protocolares

As mensagens seguem um formato binário otimizado, dividido em campos que identificam o tipo de mensagem, número de sequência e *payload* específico. Divide-se em cinco tipos principais de mensagens: Register, Ack, Task, Metric e Report. Cada tipo possui uma estrutura adaptada ao seu propósito. Por exemplo, mensagens de registo incluem o ID do agente, enquanto mensagens de tarefa levam informações detalhadas como tipo de tarefa, frequência de execução e valores de *threshold*. Métricas coletadas são enviadas em mensagens Metric, que incluem o tipo da métrica e o valor correspondente. Este formato permite eficiência na comunicação e compatibilidade entre servidores e agentes.

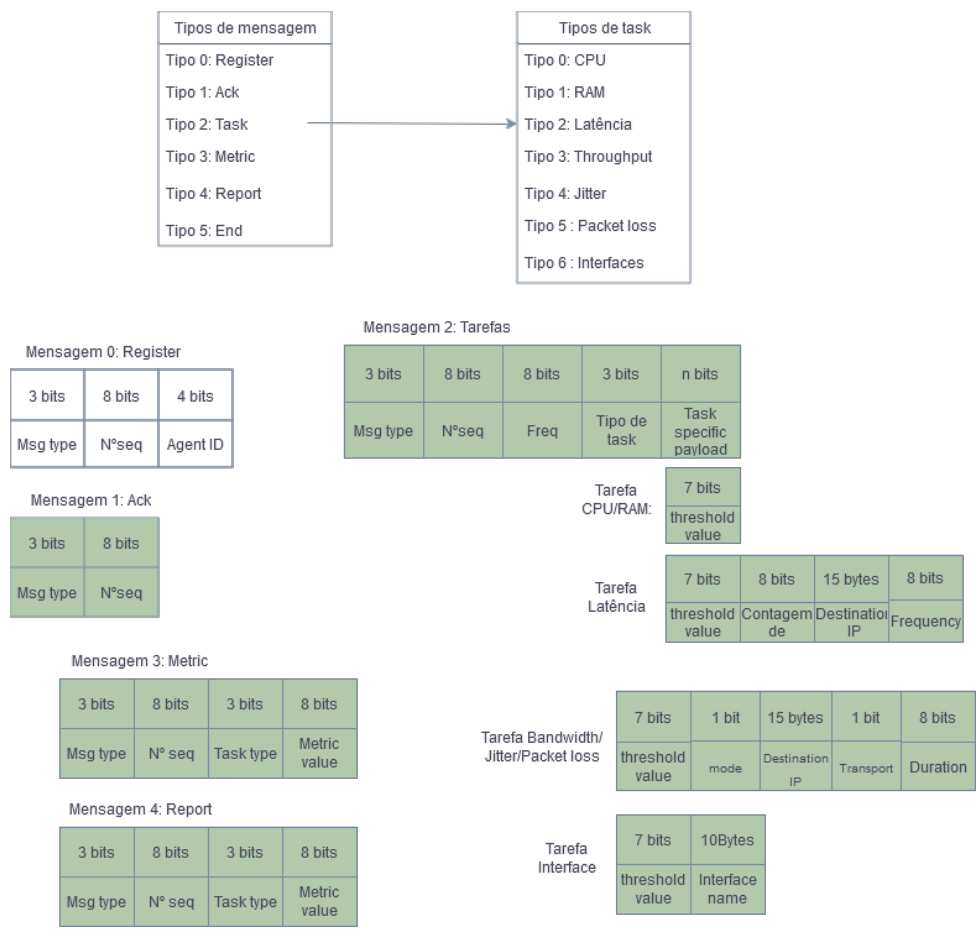


Figura 3.1: Formato das mensagens protocolares

## 3.2 Diagrama de sequência

O diagrama de sequência ilustra as interações entre os componentes principais do sistema: o servidor e os agentes. Ele detalha as etapas de registro do agente, envio de tarefas pelo servidor, coleta de métricas pelos agentes, e envio de alertas críticos para o servidor.

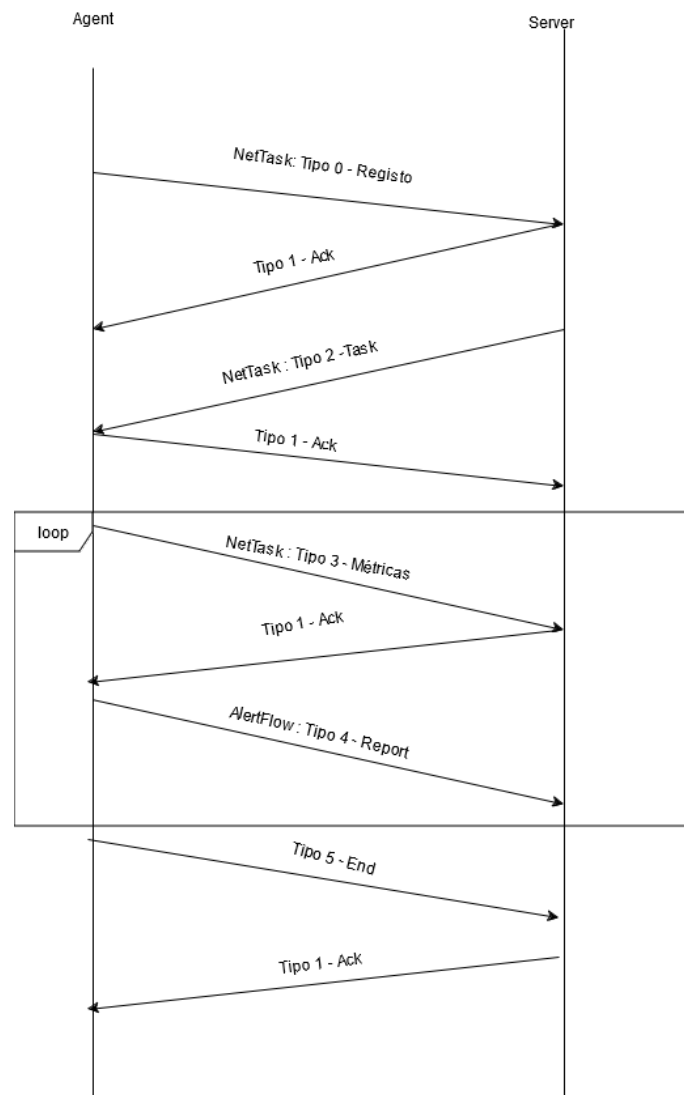


Figura 3.2: Diagrama de sequência das interações entre servidor e agentes

## 4 Implementação

A implementação do sistema foi concebida para garantir eficiência e modularidade, utilizando sockets para gerir a comunicação entre o servidor e os agentes. O servidor opera em duas portas distintas: uma porta UDP para o registo de agentes e envio de tarefas, e uma porta TCP para receber alertas críticos, aproveitando as características de cada protocolo. Sockets não bloqueantes são geridos com a função `select`, permitindo que o servidor monitorize múltiplas conexões simultaneamente. Do lado do agente, é utilizado um socket UDP para a comunicação regular com o servidor, incluindo o registo inicial, envio de ACKs e receção de tarefas. A implementação integra ainda threads para processar mensagens em paralelo, assegurando que a comunicação seja tratada de forma eficiente e responsiva.

### 4.1 Bibliotecas

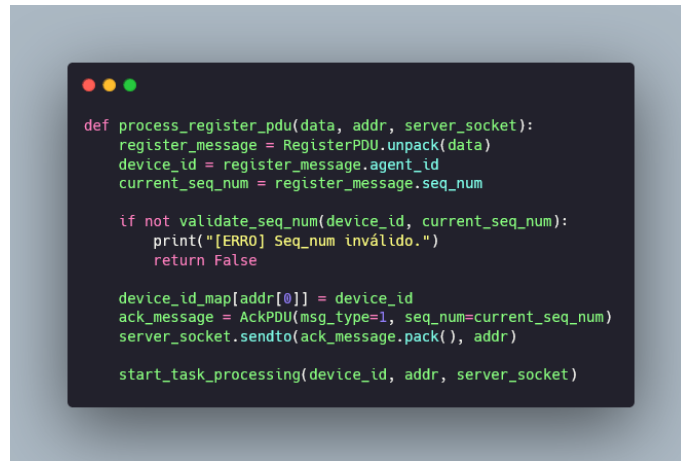
A implementação faz uso de diversas bibliotecas padrão e de terceiros, que fornecem suporte para comunicação de rede, manipulação de dados e execução de tarefas. Entre as principais bibliotecas estão:

- *socket*: Para implementar a comunicação via UDP e TCP entre o servidor e agentes.
- *struct*: Para empacotamento e desempacotamento de mensagens binárias.
- *subprocess*: Para executar comandos externos, como testes de rede (ping e iperf).
- *psutil*: Para coleta de métricas do sistema, como uso de CPU e memória.
- *threading*: Para gerir tarefas concorrentes, como a receção de mensagens e envio periódico de métricas.

### 4.2 Detalhes do servidor

O servidor foi implementado com foco na gestão eficiente de múltiplos agentes. Em relação ao registo de agentes: Os agentes enviam uma mensagem de registo com seu identificador único. O servidor valida o número de sequência e responde com um *ACK*.





```
def process_register_pdu(data, addr, server_socket):
    register_message = RegisterPDU.unpack(data)
    device_id = register_message.agent_id
    current_seq_num = register_message.seq_num

    if not validate_seq_num(device_id, current_seq_num):
        print("[ERRO] Seq_num inválido.")
        return False

    device_id_map[addr[0]] = device_id
    ack_message = AckPDU(msg_type=1, seq_num=current_seq_num)
    server_socket.sendto(ack_message.pack(), addr)

    start_task_processing(device_id, addr, server_socket)
```

Figura 4.1: pdu\_processor.py

A gestão de tarefas foi projetada para garantir que cada agente receba as tarefas designadas de forma eficiente e ordenada. Para isso, foi implementado um mecanismo baseado em filas (*queues*) para organizar e controlar as tarefas atribuídas a cada agente. Cada agente possui a sua própria fila de tarefas, armazenada no gestor de tarefas *task\_manager.py*. Essa abordagem permite que o servidor administre múltiplos agentes simultaneamente, assegurando que as tarefas sejam enviadas na ordem correta e respeitando os intervalos de execução configurados. Além disso, cada tarefa é acompanhada por um número de sequência único, que ajuda a rastrear respostas e evitar duplicações e ainda processa todo o tipo de métricas recebidas.

```

def queue_task(self, device_id, task_pdu):
    """Adiciona uma tarefa à fila do dispositivo"""
    with self._lock:
        if device_id not in self._task_queues:
            self._task_queues[device_id] = Queue()
            self._task_queues[device_id].put(task_pdu)

def send_single_task(self, task_pdu, device_id, agent_addr, server_socket):
    task_pdu.seq_num = get_next_seq_num(device_id, 1)
    packed_task = task_pdu.pack()

    ack_event = threading.Event()
    metric_event = threading.Event()

    try:
        for attempt in range(3):
            server_socket.sendto(packed_task, agent_addr)

            # Aguarda ACK por 5 segundos
            if ack_event.wait(5.0):
                with self._lock:
                    key = (device_id, task_pdu.seq_num)
                    if key in self._pending_metrics:
                        print("[DUP] Pacote Duplicado\n")
                        return True
                if metric_event.wait(30.0):
                    return True
            else:
                print(f"[TIMEOUT] Não recebeu métrica para a tarefa {task_pdu.seq_num}\n")
                return False

            print(f"[TIMEOUT] Tentativa {attempt + 1} de envio da tarefa {task_pdu.seq_num}\n")
            time.sleep(1)

        return False

    finally:
        # Limpa os eventos pendentes
        with self._lock:
            self._pending_tasks.pop((device_id, task_pdu.seq_num), None)
            self._pending_metrics.pop((device_id, task_pdu.seq_num), None)

```

Figura 4.2: tasks\_manager.py

## 4.3 Detalhes dos Agentes

Os agentes coletam as métricas periodicamente. Para isto, utilizam a biblioteca *psutil* (como referido acima) para métricas de CPU, memória e interfaces, além de executar comandos externos para medições de latência (ping), *throughput*, *packet loss* e *jitter* (iperf).

```

def _collect_metric(self, task_pdu):
    """Coleta a métrica específica da tarefa"""
    task_type = task_pdu.task_type
    metric_value = 0

    if task_type == 0: #CPU
        metric_value = int(psutil.cpu_percent())
    elif task_type == 1: #RAM
        metric_value = int(psutil.virtual_memory().percent)
    elif task_type == 2: # Latencia
        metric_value = self._execute_ping_task(task_pdu.payload)
    elif task_type in [3,4,5]:
        if task_pdu.payload.transport == "TCP" and task_type in [4,5]:
            print("[IPERF] As tarefas 4 e 5 não executam numa conexão TCP \n")
            return metric_value
        metrics = self._execute_iperf_metrics(task_pdu.payload)
        if task_type == 3:
            metric_value = metrics[0]
        if task_type == 4:
            metric_value = metrics[1]
        if task_type == 5:
            metric_value = metrics[2]
    elif task_type == 6:
        metric_value = self._execute_pps_task(task_pdu.payload)

```

Figura 4.3: tasks\_executor.py

Após coletar as métricas, o agente envia os dados ao servidor utilizando mensagens UDP. Quando valores críticos são detetados, alertas são enviados via TCP (já referido anteriormente).

```

def _send_metric(self, task_type, metric_value, seq_num):
    """Envia métrica para o servidor com tentativas de retransmissão"""
    try:
        # Arredonda o valor da métrica
        rounded_value = round(metric_value, 3)

        # Cria a PDU da métrica
        metric_pdu = MetricPDU(
            msg_type=3,
            seq_num=seq_num,
            task_type=task_type,
            metric_value=rounded_value
        )

        # Envia a métrica usando o socket
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as metric_socket:
            attempt = 0
            max_attempts = 3
            while attempt < max_attempts:
                try:
                    metric_socket.settimeout(5)

                    metric_socket.sendto(metric_pdu.pack(), (self.server_ip, self.task_port))

                    ack_data, _ = metric_socket.recvfrom(1024)
                    ack_message = AckPDU.unpack(ack_data)
                    if ack_message.seq_num == metric_pdu.seq_num:
                        return True

                except socket.timeout:
                    print(f"[TIMEOUT] Tentativa {attempt + 1} de envio da métrica")
                    attempt += 1

            print("[ERROR] Falha ao receber confirmação da métrica após várias tentativas")
            return False

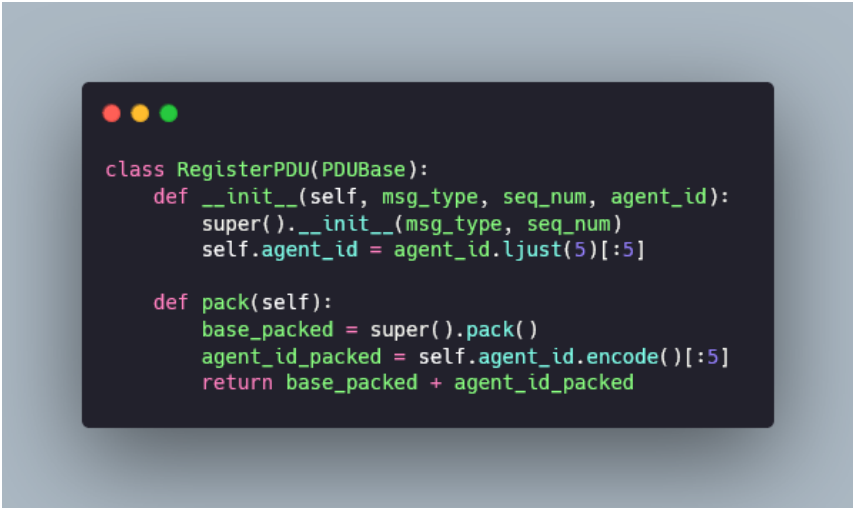
    except Exception as e:
        print(f"[ERROR] Falha ao enviar métrica: {e}")

```

Figura 4.4: Envio de métricas

## 4.4 Estrutura de mensagens

As mensagens seguem o formato binário definido na especificação dos protocolos. O empacotamento e desempacotamento utilizam a biblioteca *struct* para garantir eficiência na transmissão. Cada mensagem contém campos fixos, como tipo de mensagem e número de sequência, e campos variáveis dependendo do tipo. Observamos ainda que todas as mensagens iriam ter os campos *'seq\_num'* bem como o tipo de mensagem *'msg\_type'* e então ficou como super classe em que todos as outras mensagens já iriam herdar os mesmos fazendo *super()* como se verifica abaixo:

A screenshot of a code editor window with a dark background and light-colored text. The code defines a class `RegisterPDU` that inherits from `PDUBase`. It includes an `__init__` method that calls `super().__init__` and sets `self.agent_id` to the last 5 bytes of `agent_id`. It also includes a `pack` method that calls `super().pack()` and appends the encoded `agent_id` to the packed data.

```
class RegisterPDU(PDUBase):
    def __init__(self, msg_type, seq_num, agent_id):
        super().__init__(msg_type, seq_num)
        self.agent_id = agent_id.ljust(5)[:5]


    def pack(self):
        base_packed = super().pack()
        agent_id_packed = self.agent_id.encode()[:5]
        return base_packed + agent_id_packed
```

Figura 4.5: Register PDU

## 4.5 Data Storage e Parser

O armazenamento de tarefas e métricas foi implementado de forma eficiente. Utilizamos arquivos JSON.

O parser é responsável por interpretar as tarefas do arquivo JSON para objetos do tipo *NetTaskPDU*, utilizados pelo sistema. Para suportar diferentes tipos de tarefa, o parser foi modularizado, delegando a validação e o processamento do *payload* específico a classes dedicadas, como o *CPUPayload* por exemplo. Isso permite fácil adição de novos tipos de métricas sem modificar o parser central.



```
for task in tasks_data['tasks']:
    for device in task['devices']:
        device_id = device['device_id']
        freq = task['frequency']

        # Processa tarefas de CPU
        if "cpu_usage" in device['device_metrics'] and device['device_metrics']['cpu_usage'] ==
True:
            threshold = device['alertflow_conditions'].get('cpu_usage', 50)
            task_pdu = NetTaskPDU(
                msg_type=2,
                seq_num=0, # seq_num sera atualizado no task_manager
                freq=freq,
                task_type=0,
                payload=CPUPayload(threshold_value=threshold)
            )
            tasks.append((device_id, task_pdu))
```

Figura 4.6: Exemplo CPU Task Parser

## 5 Testes e Resultados

Para uma melhor interpretação das métricas e dos alertas, desenvolvemos um *website*, onde carregámos os ficheiros *json* de outputs para uma melhor visualização e análise.

### 5.1 Tarefa CPU

Na tarefa do CPU, medimos a percentagem de CPU utilizada pelo host. Os resultados obtidos nesta tarefa para a topologia usada, estiveram entre 1% e 15%, valores que se encontram bastante abaixo dos valores de threshold.

### 5.2 Tarefa RAM

Na tarefa da RAM, medimos a percentagem de RAM utilizada pelo host. Foi possível ver que à medida que se vão adicionando hosts, vai aumentando a percentagem de RAM utilizada, sendo que para a topologia utilizada, obtemos valores entre os 70% e os 80% valores que se encontram bastante próximos dos valores de threshold definidos.

### 5.3 Tarefa Latency

Na tarefa da Latency, medimos a latência entre dois endereços de IP, sendo que um deles é o do Agente no qual esta tarefa é realizada. No PC4, onde executamos a tarefa para o PC3, obtivemos resultados entre os 21 e os 22 ms. Estes valores não variam muito numa mesma rota, pois estes dependem maioritariamente da rota percorrida.

### 5.4 Tarefa Bandwidth

Na tarefa da Bandwidth, medimos a largura de banda entre um cliente e um servidor. No PC4, onde executamos a tarefa como cliente para o PC2, obtivemos o valor 1.05 Mbps.

## 5.5 Tarefa Jitter

Na tarefa da Jitter, medimos o jitter entre um cliente e um servidor. No PC4, onde executamos a tarefa como cliente para o PC2, obtivemos resultados entre os 0 e 3 ms, valores considerados bons e bastante abaixo do threshold definido.

## 5.6 Tarefa Packet Loss

Na tarefa do Packet Loss, medimos a percentagem de packet loss entre um cliente e um servidor. No PC4, onde executamos a tarefa como cliente para o PC2, temos uma rota com 10% de loss e obtivemos resultados entre os 5% e os 24%, valores considerados preocupantes e que são característicos de uma rede praticamente inutilizável. Para além disso, estes valores excedem bastante o valor definido de threshold.

## 5.7 Tarefa Packet Loss

Na tarefa do Packet Loss, medimos a percentagem de packet loss entre um cliente e um servidor. No PC4, onde executamos a tarefa como cliente para o PC2, temos uma rota com 10% de loss e obtivemos resultados entre os 5% e os 24%, valores considerados preocupantes e que são característicos de uma rede praticamente inutilizável. Para além disso, estes valores excedem bastante o valor definido de threshold.

## 5.8 Tarefa Interfaces/Packets Per Second

Na tarefa interfaces, medimos a quantidade de pacotes recebida por uma interface de um Agente. No PC2, onde executamos um servidor do iperf, foi onde houve um valor maior nesta tarefa. Os valores obtidos encontram-se entre os 0 pps e os 57 pps, sendo que estes valores são bastante aceitáveis e encontram-se bastante abaixo do threshold definido.

PC	Seq Num	Timestamp	Task Type	Metric Value
[PC4]	9	2024-12-06T15:43:24.790738	CPU	3.00 %
[PC4]	10	2024-12-06T15:43:24.836427	RAM	76.00 %
[PC4]	11	2024-12-06T15:43:25.009193	Latency	22.00 ms
[PC4]	12	2024-12-06T15:43:28.735011	Bandwidth	1.05 Mbps
[PC4]	13	2024-12-06T15:43:32.422606	Jitter	2.46 ms
[PC4]	14	2024-12-06T15:43:39.230803	Packet Loss	23.16 %
[PC4]	15	2024-12-06T15:43:49.285255	Packets Per Second	1.00 pps

Figura 5.1: Métricas recolhidas pelo PC4

PC	Seq Num	Timestamp	Task Type	Metric Value
[PC2]	9	2024-12-06T15:43:18.464857	CPU	1.00 %
[PC2]	10	2024-12-06T15:43:18.469136	RAM	76.00 %
[PC2]	11	2024-12-06T15:43:18.647693	Latency	40.00 ms
[PC2]	15	2024-12-06T15:43:38.677089	Packets Per Second	42.00 pps

Figura 5.2: Métricas recolhidas pelo PC2

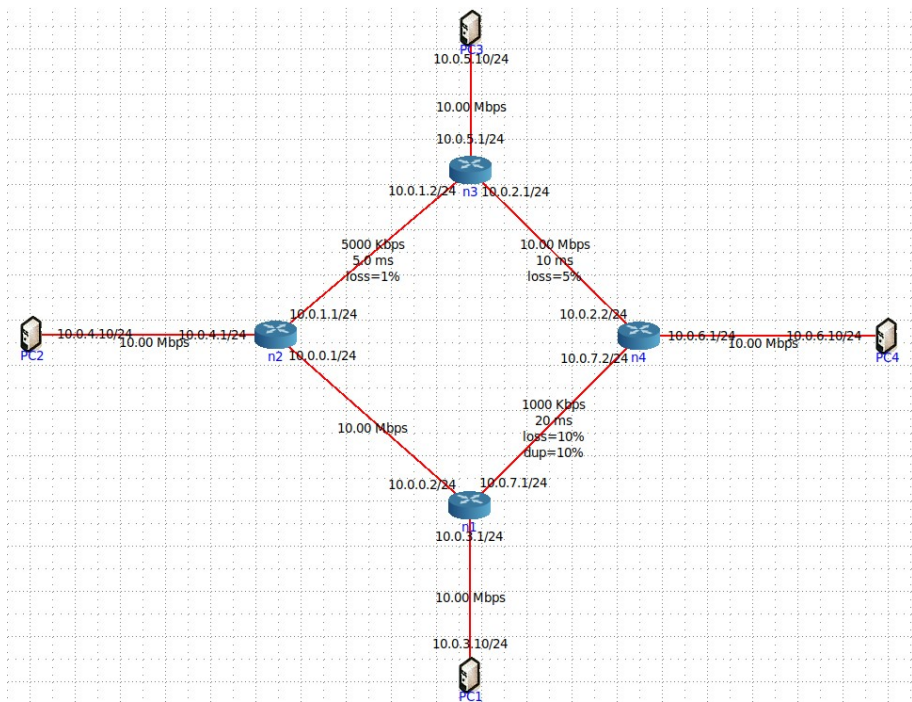


Figura 5.3: Topologia utilizada para testar a solução



## 6 Conclusões e Trabalho Futuro

O trabalho prático demonstrou a viabilidade de implementar um sistema distribuído de monitorização de redes, utilizando uma arquitetura cliente-servidor. Através dos protocolos NetTask (UDP) e AlertFlow (TCP), foi possível garantir a coleta eficiente de métricas e a notificação confiável de eventos críticos.

Os testes realizados validaram a funcionalidade do sistema, incluindo o registo de agentes, o envio e a receção de tarefas, a coleta de métricas e a emissão de alertas em condições críticas. O sistema também demonstrou resiliência em cenários com perda de pacotes, assegurando a continuidade das operações através de mecanismos de retransmissão.

Embora o sistema tenha atendido aos objetivos definidos, melhorias futuras podem incluir a integração de métricas adicionais e a implementação de uma interface gráfica para facilitar a visualização em tempo real dos dados monitorados. Este projeto contribuiu significativamente para o aprendizado de conceitos como desenvolvimento de protocolos aplicacionais e resiliência em redes distribuídas.