Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

Network Monitoring System

Sara Azevedo Lopes Diogo do Rego Neto Rui Filipe Mesquita Amaral

7 de dezembro de 2024

CC

Data de Receção	
Responsável	
Avalição	
Observações	

Network Monitoring System

Sara Azevedo Lopes Diogo do Rego Neto Rui Filipe Mesquita Amaral

7 de dezembro de 2024

Resumo

Este trabalho apresenta o desenvolvimento de um sistema de monitorização de redes distribuído, composto por dois módulos principais: o *Server*, responsável pela gestão centralizada, e os *Agents*, que recolhem métricas e notificam sobre anomalias sentidas na rede. A comunicação entre os módulos é garantida por dois protocolos aplicacionais: o *NetTask*, baseado em *UDP*, para troca eficiente de tarefas e métricas e o *AlertFlow*, baseado em *TCP*, para notificações críticas.

O *Server* processa ficheiros *JSON* contendo as tarefas a atribuir, coordenando a recolha de métricas, armazenamento e consulta de dados. Por sua vez, os *Agents* executam comandos de sistema, como testes de largura de banda e latência, monitorizando parâmetros como utilização de *CPU*, *RAM* e estatísticas de *interfaces*. Os dados recolhidos são enviados periodicamente ao servidor, enquanto alertas são disparados sempre que valores críticos são excedidos.

Durante o desenvolvimento, foram implementados mecanismos adicionais para assegurar a resiliência e fiabilidade, incluindo retransmissão por *timeout* e números de sequência, adaptando características do *TCP* ao *UDP*. A validação foi realizada em cenários simulados com perdas de pacotes e latências elevadas, usando o *emulador CORE*, demonstrando a eficácia da solução.

O sistema final é funcional, permitindo uma monitorização robusta e eficiente, cujos resultados podem ser visualizados através de uma *interface* gráfica. Possui também potencial para extensões futuras, como integração de mecanismos de segurança, maior potencial de paralelização.

Área de Aplicação: Monitorização e Gestão de Redes de Computadores, com foco no desenvolvimento de protocolos aplicacionais para comunicação distribuída.

Palavras-Chave: Redes de Computadores, Monitorização de Redes, Protocolos Aplicacionais, *UDP*, *TCP*, Programação de *Sockets*, Resiliência em Redes, *CORE Emulator*, *NetTask*, *AlertFlow*, *Interface*, Página *Web*, *HTTP*, *DNS*

Índice

Lista de Figuras

1 Introdução

A monitorização de redes é uma componente fundamental para assegurar o correto funcionamento de infraestruturas de comunicação, sendo essencial em contextos onde a fiabilidade e o desempenho das ligações são críticos.

Para lidar com os desafios impostos por redes sujeitas a perdas de pacotes, latência elevada e outras condições adversas, este trabalho prático propõe o desenvolvimento de uma solução distribuída composta por dois módulos principais: o *NMS_Server* e os *NMS_Agents*.

O projeto baseia-se na implementação de dois protocolos aplicacionais: *NetTask*, que utiliza a camada de transporte *UDP* para comunicação eficiente e resiliente ao implementar alguns dos mecanismos do *TCP*, e *AlertFlow*, que utiliza a camada de transporte *TCP* para garantir a entrega fiável de notificações críticas. Ambos os protocolos serão projetados para lidar com cenários de redes suscetíveis a falhas, garantindo a recolha, transmissão e apresentação de métricas importantes para a gestão da rede.

1.1 Contextualização

O trabalho será desenvolvido no emulador de redes *CORE 7.5*, utilizando a topologia *CC-Topo-2024.imn*, que simula redes com diferentes níveis de perda de pacotes.

Neste cenário, o *NMS_Server* atua como um gestor central, atribuindo tarefas aos *NMS_Agents* com base num ficheiro de configuração *JSON*. Estes agentes são responsáveis por recolher métricas de dispositivos de rede, como uso de *CPU*, *RAM*, estatísticas de *interfaces*, largura de banda, latência e *jitter*.

Adicionalmente, os *NMS_Agents* enviam notificações ao servidor sempre que limites críticos são excedidos. Este mecanismo assegura a deteção atempada de problemas e permite uma gestão eficiente e proativa da rede.

1.2 Apresentação do Caso de Estudo

O caso de estudo centra-se na simulação de um ambiente de rede utilizando o emulador *CORE 7.5* e a topologia *CC-Topo-2024.imn*, que reproduz cenários reais com diferentes níveis de

perda de pacotes e condições adversas.

O *NMS_Server* desempenha o papel de gestor central, atribuindo tarefas de monitorização aos *NMS_Agents* com base em configurações detalhadas fornecidas num ficheiro *JSON*. Estes agentes, distribuídos por diferentes localizações na rede, recolhem métricas fundamentais como uso de *CPU*, *RAM*, estatísticas de *interfaces* de rede, largura de banda, latência e *jitter*.

Além disso, o sistema foi concebido para lidar com falhas na rede. Os *NMS_Agents* são capazes de detetar alterações críticas nas métricas monitorizadas e notificar o *NMS_Server* através do protocolo fiável *AlertFlow*, garantindo a gestão proativa de problemas.

1.3 Motivação e Objectivos

O rápido crescimento das redes de comunicação e a crescente complexidade das suas infraestruturas criam a necessidade de soluções robustas para a monitorização e gestão. Este trabalho surge da motivação de desenvolver um sistema que não só permita a monitorização em tempo real, mas também reaja de forma eficaz a eventos críticos.

Os principais objetivos deste trabalho incluem:

- Desenvolver dois protocolos aplicacionais: o *NetTask*, para comunicação eficiente utilizando *UDP*, e o *AlertFlow*, para notificações críticas utilizando *TCP*.
- Implementar o *NMS_Server*, capacitando-o a interpretar ficheiros *JSON*, atribuir tarefas e centralizar a recolha de métricas/alertas.
- Construir os *NMS_Agents* para recolher e transmitir métricas em cenários de redes com perdas.
- Garantir que o sistema é resiliente e eficiente, mesmo sob condições adversas.
- Validar a solução através de testes no emulador *CORE 7.5*.

1.4 Estrutura do Relatório

Este relatório está organizado em capítulos que abordam todas as fases do projeto, desde a sua conceção até à validação final. A estrutura detalhada é apresentada a seguir:

- **Introdução:** Apresentação do contexto, caso de estudo, objetivos do trabalho e organização do relatório.
- **Desenvolvimento:** Este capítulo agrupa as principais etapas do projeto, desde o *design* até à implementação e inclui decisões de *design* técnico.

  - **Arquitetura da Solução:** Descrição da estrutura geral do sistema, incluindo os módulos *Server* e *Agent*, e a forma como interagem entre si.
  - **Estrutura do PDU NetTask:** Descreve as estruturas dos *PDUs* do protocolo NetTask, detalhando campos.
  - **Estrutura do PDU AlertFlow:** Descreve as estruturas dos *PDUs* do protocolo *AlertFlow*, detalhando campos.
  - **Interações no Sistema:** Documenta as interações entre os Agentes e o Servidor que estão contempladas no sistema, dando significado aos *PDUs* elaborados.
  - **Mecanismos de Retransmissão no protocolo NetTask:** Explica mecanismos de retransmissão no *NetTask*, usando timeout e 3 *ACKs* duplicados para garantir entrega confiável, adaptando funcionalidades do *TCP* ao *UDP*.
  - ***DNS*:** Descreve a implementação de *DNS* que foi efetuada, e como esta é utilizada para aumentar a robustez do sistema desenvolvido.
  - **Interface Gráfica:** Documenta o desenvolvimento da aplicação *web* responsável pela recolha das métricas/alertas do servidor e caracteriza a disponibilização gráfica das mesmas.
  - ***JSON*:** Indica a organização das tarefas de cada agente.
  - **Logging:** Apresenta o registo de eventos importantes em ficheiros e no terminal, incluindo timestamps e níveis de severidade.
- **Testes e Resultados:** Cenários de teste realizados no emulador *CORE 7.5*, demonstrando todas as funcionalidades previamente documentadas.
- **Conclusões e Trabalho Futuro:** Reflexão sobre os resultados alcançados, limitações da solução implementada e propostas de melhorias ou extensões futuras.

2 Desenvolvimento

Esta secção foca-se em detalhar as funcionalidades implementadas.

2.1 Arquitetura da Solução

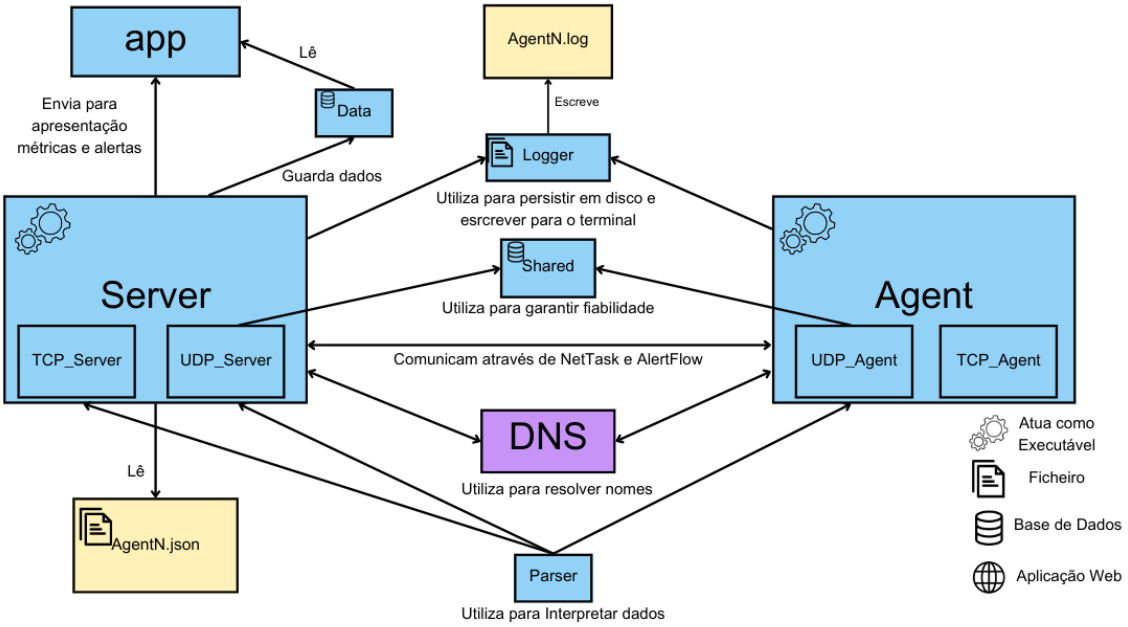

Figura 2.1: Arquitetura do sistema

De acordo com a imagem a cima, foram desenvolvidos os seguintes módulos:

Server: Conjunto de módulos que compõe o sistema centralizado que gere a sessão de todos os agentes, envia corretamente as suas tarefas, e ao recolher as suas métricas e alertas é capaz de dar um significado global às interações que ocorrem na rede. Além de comunicar através dos protocolos *NetTask* e *AlertFlow* com todos os agentes, é ainda responsável por recolher os dados para apresentação na *webapp*.

Agent: Representa um agente individual, encarregado de gerir tarefas, registar *logs* e co-

municar com o servidor utilizando os protocolos *NetTask* e *AlertFlow*. Ele executa tarefas específicas, monitoriza thresholds e dispara alertas quando necessário, enviando mensagens binárias por *UDP* com as métricas e por *TCP* com os alertas. Dentro do módulo, o *TCP_Agent* lida com a comunicação via *sockets TCP*, enviando alertas formatados e processando a fila de notificações. O *UDP_Agent* cuida da comunicação em tempo real via *sockets UDP*, processando mensagens do servidor, executando medições e testes de desempenho, e assegurando a entrega confiável com retransmissões. Estas partes funcionam de forma integrada para que o agente cumpra o seu papel no sistema distribuído.

Parser: Analisa mensagens recebidas em formato binário ou interpretar ficheiros *JSON*, extraindo informações relevantes para a comunicação entre agentes e servidor.

Shared: Mecanismo de sincronização e partilha de estado entre diferentes *threads*. É crucial para gerir a comunicação, controlar duplicações de *ACKs*, monitorizar pacotes recebidos, tratar *timeouts* e coordenar ações entre *threads*.

Data: Guarda em memória a informação de métricas e alertas, sendo consumido pela *webapp* para disponibilizar a informação visualmente

Logger: Registo de *logs*, com diferentes níveis de gravidade, permitindo guardar informação em ficheiros e apresentá-la no terminal.

App: Aplicação *web* para visualização dos dados de agentes do sistema, inclui as suas tarefas, métricas e alertas registados e quando estes ocorreram.

DNS: Resolução de nomes na topologia, permitindo traduzir nomes em endereços e vice-versa.

AgentN.log: Conjunto de tarefas que o servidor deve enviar ao agente N após o seu registo.

2.2 Estrutura do PDU NetTask

Estrutura do PDU para Registo, Ack e End

Campo	Agent Identifier	Message Type	Sequence Number
Espaço (bits)	5	3	8

Figura 2.2: Descrição da PDU - Registo, Ack e End.

Os campos apresentados na estrutura *PDU* têm as seguintes funções:

- **Agent Identifier:** Identifica de forma única o agente que enviou/recebeu a mensagem, permitindo distinguir diferentes agentes na rede. Também possibilita verificar se a mensagem é destinada ao agente correto.
- **Message Type:** Especifica o tipo de mensagem enviada/recebida. Indica o tipo de mensagem (no caso acima, um Registo, *Ack* ou *End*). Apenas precisamos de 3 *bits*, uma vez que só há 5 tipos de mensagem:

Valor Binário	O que representa	Número do tipo de mensagem
0b000	Registo	0
0b001	Ack	1
0b010	Task	2
0b011	Metric	3
0b100	End	4

Figura 2.3: Descrição do campo Message Type.

- **Sequence Number:** Número de sequência da mensagem, utilizado para rastrear a ordem das mensagens e detetar perdas na comunicação.

Estrutura do PDU para Tarefa

Se Task Type = 0, 1, 2 ou 4:

Campo	Agent Identifier	Message Type	Sequence Number	Task Identifier	Task Type	Frequency	Threshold
Espaço (bits)	5	3	8	5	3	8	8

Figura 2.4: Descrição PDU - Tarefas para CPU, RAM, Latência e Interface.

Os campos apresentados na estrutura *PDU* têm as seguintes funções:

- **Agent Identifier:** Explicado anteriormente.
- **Message Type:** Indica o tipo de mensagem (no caso acima, uma tarefa).
- **Sequence Number:** Explicado anteriormente.
- **Task Indentifier:** Indica de forma única a tarefa a ser executada.
- **Task Type:** Especifica o tipo de tarefa enviada/recebida. Apenas precisamos de 3 *bits*, uma vez que só há 5 tipos de mensagem:

Valor binário	O que representa	Número da métrica
0b000	CPU	0
0b001	RAM	1
0b010	Latência	2
0b011	Throughput	3
0b100	Interface	4

Figura 2.5: Descrição do campo Task Type.

- **Frequency:** Representa a frequência com que se vai realizar a tarefa.
- **Threshold:** Define o limite que o resultado da tarefa não deveria exceder. Para *CPU* e *RAM* este limite é expresso como uma percentagem e no caso da latência é expresso em milissegundos. Caso este seja excedido, é acionado um alerta.

Se Task Type = 3:

Campo	Agent Identifier	Message Type	Sequence Number	Task Identifier	Task Type	Frequency	Throughput Threshold	Jitter Threshold	Losses Threshold
Espaço (bits)	5	3	8	5	3	8	16	16	8

Figura 2.6: Descrição PDU - Tarefas para IPERF.

Os campos descritos na estrutura *PDU* têm as seguintes funções:

- **Agent Identifier:** Explicado anteriormente.
- **Message Type:** Indica o tipo de mensagem (no caso a cima, uma tarefa).
- **Sequence Number:** Explicado anteriormente.
- **Task Identifier:** Explicado anteriormente.
- **Task Type:** Explicado anteriormente.
- **Frequency:** Explicado anteriormente.
- **Throughput Threshold:** Define o limite máximo permitido para variação da largura de banda, expresso em *Mbps*. Caso seja excedido, é acionado um alerta.
- **Jitter Threshold:** Define o limite máximo permitido para variação do *Jitter*, expresso em milissegundos. Caso seja excedido, é acionado um alerta.
- **Losses Threshold:** Define o limite máximo permitido de perdas de pacotes, expresso em percentagem. Caso seja excedido, é acionado um alerta.

Estrutura do PDU Métrica

Se Task Type = 0, 1, 2 ou 4:

Campo	Agent Identifier	Message Type	Sequence Number	Task Identifier	Task Type	Metric Value
Espaço (bits)	5	3	8	5	3	16

Figura 2.7: Descrição PDU - Métricas para CPU, RAM, Latência e Interface.

Os campos descritos na estrutura *PDU* têm as seguintes funções:

- **Agent Identifier:** Explicado anteriormente.
- **Message Type:** Explicado anteriormente.
- **Sequence Number:** Explicado anteriormente.

- **Task Identifier:** Explicado anteriormente.
- **Task Type:** Explicado anteriormente.
- **Metric Value:** Representa o valor numérico medido para a tarefa correspondente.

Se Task Type = 3:

Campo	Agent Identifier	Message Type	Sequence Number	Task Identifier	Task Type	Throughput	Throughput Unit	Jitter	Losses
Espaço (bits)	5	3	8	5	3	16	8	16	8

Figura 2.8: Descrição PDU - Métricas para IPERF.

Os campos descritos na estrutura *PDU* têm as seguintes funções:

- **Agent Identifier:** Explicado anteriormente.
- **Message Type:** Explicado anteriormente.
- **Sequence Number:** Explicado anteriormente.
- **Task Identifier:** Explicado anteriormente.
- **Task Type:** Explicado anteriormente.
- **Throughput:** Representa o valor medido da largura de banda utilizada.
- **Throughput Unit:** Define a unidade de medição do throughput. Esta é necessária, porque observamos medições em *Kbps* e *Mbps*.
- **Jitter:** Representa a variação medida na latência da rede.
- **Losses:** Representa a percentagem de pacotes perdidos na comunicação.

2.3 Estrutura do PDU AlertFlow

Os campos descritos na estrutura do *PDU* do *AlertFlow* seguem a mesma lógica que os campos do *PDU* do *NetTask*, com a exceção do campo de *Padding*, que serve para que todos os segmentos tenham exatamente 8 *bytes*, uma vez que o *TCP* não contempla segmentação.

Campo	Agent Identifier	Task Type	Task Identifier	Metric	Padding
Espaço (bits)	5	3	8	16	32

Figura 2.9: PDU AlertFlow para alertas de CPU, RAM e Latência

Campo	Agent Identifier	Task Type	Task Identifier	Throughput	Throughput Unit	Jitter	Losses
Espaço (bits)	5	3	8	16	8	16	8

Figura 2.10: PDU AlertFlow para alertas de IPERF

Campo	Agent Identifier	Task Type	Task Identifier	Interface State	Padding
Espaço (bits)	5	3	8	8	40

Figura 2.11: PDU AlertFlow para alertas de estado de interface

2.4 Interações no Sistema

Após a explicação detalhada dos cabeçalhos utilizados nos protocolos implementados, podemos agora particularizar as interações que estes nos permitem ter, e que levam a que o sistema possa suprir as suas necessidades.

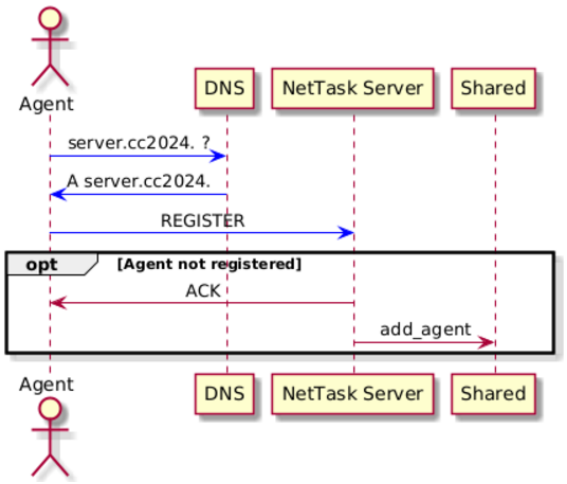

Figura 2.12: Diagrama de sequência para o registo dos agentes

No registo, o agente começa por resolver o nome "server"(no domínio cc2024) através de uma *query* enviada ao nosso servidor de *DNS*. Após obter o endereço do servidor, envia um pacote de registo com o cabeçalho descrito nas secções anteriores. Caso o agente não estivesse previamente registado, o *NetTask Server* envia-lhe um *ACK* confirmando a receção do pacote, e adiciona uma entrada no objeto partilhado do tipo *Shared*. Este último será posteriormente utilizado para as funcionalidades do *TCP* que foram implementadas sobre *UDP*. Na eventualidade de duplicação de um pacote ou tentativas de registo quando o agente já se encontra registado, o servidor irá ignorar o pedido de registo.

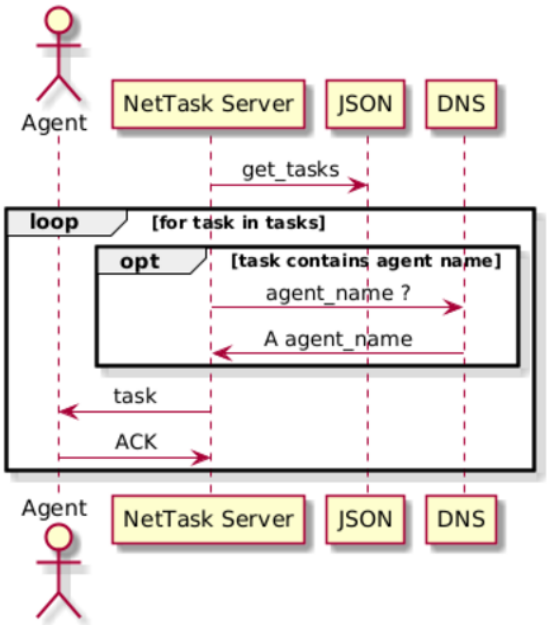

Figura 2.13: Diagrama de sequência para o envio das tasks aos agentes

Uma vez concluído o registo de um agente, o servidor vai aceder ao ficheiro *JSON* correspondente às tarefas deste agente. Algumas delas contêm nomes para os quais a tarefa deverá apontar, pelo que o servidor faz a resolução do nome e inclui apenas o endereço *IPv4* correspondente no pacote que contém a tarefa a atribuir.

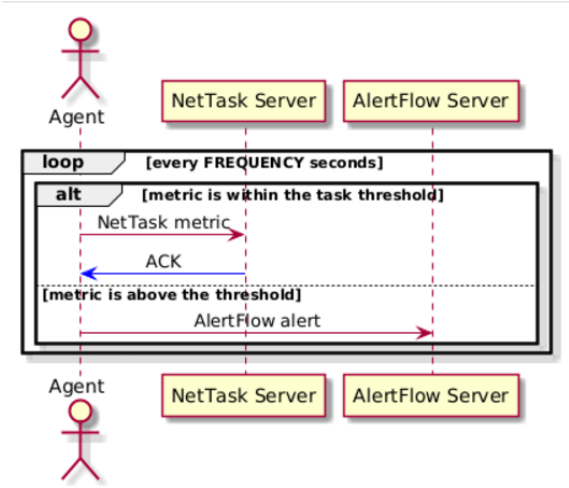

Figura 2.14: Diagrama de sequência para o envio das metricas/alertas ao servidor

Após a receção das tarefas, o agente começará a desempenhá-las e a cada período de tempo estipulado pela frequência associada à tarefa, envia a métrica recolhida por *UDP* ao *NetTask Server*, que confirmará a entrega do pacote.

No caso da métrica recolhida exceder o *Threshold* definido para a tarefa, é então enviado um alerta através do protocolo *AlertFlow*, que atua sobre *TCP*.

No caso do servidor ser encerrado, este vai notificar todos os agentes para que não continuem a enviar métricas e alertas:

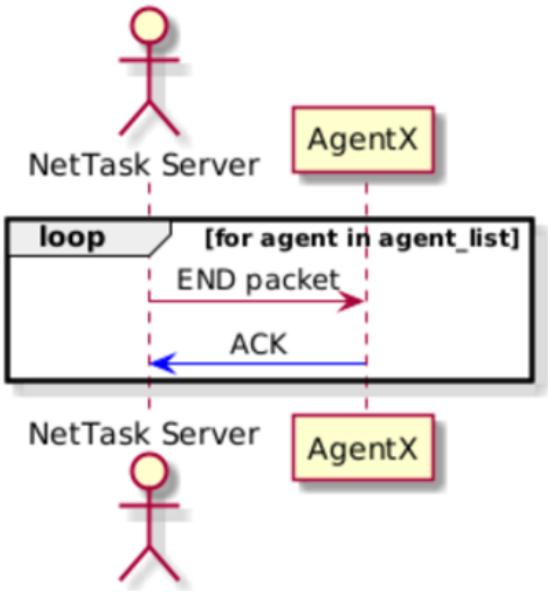

Figura 2.15: Diagrama de sequência para o envio da mensagem de END a partir do Servidor

Esta funcionalidade é implementada através de um handler para o sinal *SIGINT* enviado quando o utilizador sai do programa utilizando *CTRL+C*.

Deste modo, garante-se que não existe tráfego adicional gerado quando o servidor abandona o sistema, uma vez que não haveria um coletor centralizado para as recolher e seriam portanto inúteis.

O comportamento inverso também é contemplado no nosso sistema, em que ao sair o Agente notifica o Servidor para que não mantenha em memória a informação de um Agente que já foi desligado:

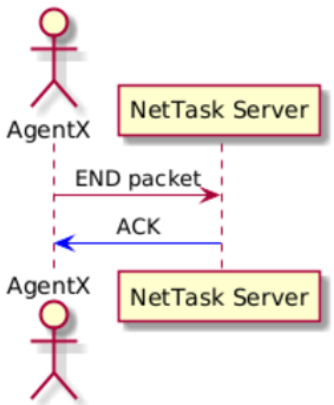

Figura 2.16: Diagrama de sequência para o envio da mensagem de END a partir do Agente

Neste caso é enviada apenas uma mensagem do Agente para o Servidor, que é corretamente confirmada pelo Servidor antes do Agente ser efetivamente desligado.

2.5 Mecanismos de Retransmissão no protocolo NetTask

Uma vez que o protocolo *NetTask* atua sobre o protocolo *UDP*, que é extremamente simples, existe a necessidade de se implementarem alguns dos mecanismos do *TCP* para que se garanta a fiabilidade na entrega do nosso serviço. Para tal, foram contemplados dois casos em que a retransmissão será necessária: retransmissão em caso de *timeout*, e retransmissão na receção de 3 *ACKs* duplicados.

Para que estas funcionalidades possam ser implementadas, existe a necessidade de termos informação para determinar se um pacote foi bem recebido. Para tal, utilizam-se números de sequência e mensagens de *Acknowledgement(ACK)* para confirmação da receção correta dos pacotes.

Cada pacote possui um número de sequência, e dependendo do tamanho do pacote, o interveniente enviará um pacote de confirmação designando qual é o *byte* a partir do qual a mensagem que espera começa. Como exemplo:

Agente20 envia uma mensagem com SEQ=20 de tamanho 7 *bytes*. Caso esta seja bem recebida, o Servidor responderá com *ACK* 27, confirmando que recebeu corretamente o pacote e espera agora um pacote que começa no *byte* 27. Deste modo, é possível ter noção se um pacote foi perdido e se deverá ser retransmitido.

Para que se dê a primeira situação a de *timeout*, ao enviar um pacote, estabeleceu-se um período de 2 segundos para a receção de um *ACK* igual ou superior ao número de sequência deste pacote mais o seu tamanho. Se não houver a receção do mesmo, temos uma situação de *timeout* e o pacote é retransmitido nesse momento.

Porém, o grupo decidiu ir mais longe e implementar mecânicas mais eficientes para a deteção e subsequente retransmissão de pacotes perdidos. O outro mecanismo que foi implementado foi o dos 3 *ACKs* duplicados, como no exemplo:

O Servidor envia um pacote de SEQ=10 com 10 *bytes*, seguido de SEQ=20 com 10 *bytes*, ao qual recebe uma resposta de ACK 10. Caso o servidor envie mais dois pacotes e receba na mesma *ACK* 10, ao chegar ao terceiro *ACK*, ele apercebe-se que o pacote de SEQ=10 não foi entregue corretamente e reenvia o mesmo imediatamente, sem esperar pelo tempo de *timeout*.

Por modo a garantir o mínimo de tráfego gerado, não é utilizado necessariamente um *ACK* para cada uma das mensagens recebidas, podendo ser utilizado um *ACK* cumulativo que confirma a receção bem sucedida de vários pacotes. Se forem enviados vários pacotes, seguidos de um pacote de SEQ=40 e com 6 *bytes*, o servidor ou o agente pode apenas enviar um *ACK*

46, confirmando tanto este pacote, como todos os anteriores. Os nossos dispositivos estão formatados para determinar o *ACK* mais alto que podem enviar a cada 200ms, garantindo um equilíbrio entre confirmações expeditas e minimização do tráfego gerado.

2.6 *DNS*

Dada a elevada importância que o *DNS* tem no mundo da *internet*, decidimos incluir esta componente no projeto, de modo a ter uma melhor oportunidade de explorar o protocolo. Para tal, foi adicionado um novo elemento à topologia, o nosso servidor de *DNS*. Este foi configurado como o servidor de *DNS* por defeito para todos os dispositivos na topologia, e está implementado recorrendo ao named. O named é um daemon bem conhecido para o efeito, sendo a sua configuração relativamente sucinta, tornando-se assim a escolha que o grupo fez para implementar *DNS* no projeto.

A sua configuração é feita através de ficheiros de zona, nos quais incluímos um ficheiro para a zona cc2024:

```
$TTL 86400  ; Default Time to Live
@       IN      SOA     ns1.cc2024. admin.cc2024. (
                        2023112101  ; Serial number (YYYYMMDDXX)
                        3600        ; Refresh time
                        1800        ; Retry time
                        1209600     ; Expire time
                        86400       ; Minimum TTL
)

; Nameservers
@       IN      NS      ns1.cc2024.

; A Records for Nameservers
ns1     IN      A       10.0.4.10
; Server Record
server  IN      A       10.0.8.10
agent20 IN      A       10.0.5.10
agent31 IN      A       10.0.6.10
agent4  IN      A       10.0.7.10
```

Figura 2.17: Zona cc2024

Nesta, além dos parâmetros de *DNS* que queremos adoptar para o protocolo, definem-se também os records do tipo A (endereço *IPv4*) para o nameserver (o nosso servidor *DNS*), assim como o servidor e os 3 agentes.

Assim, é possível fazer a resolução destes nomes em qualquer um dos dispositivos da topologia e obter o endereço correspondente. Esta funcionalidade é utilizada para que não seja necessário introduzir endereços *hardcoded* nas tarefas dos agentes, podendo em vez disso ser utilizados nomes.

Adicionalmente, foi também implementada a resolução reversa, em que queremos encontrar qual é o nome correspondente a um endereço *IPv4*. Para tal foi criada uma zona para cada subrede em que temos dispositivos que desejamos ser descobertos (ficheiros 10.0.x.rev, correspondendo às subredes desejadas).

Como exemplo de uma das subredes definidas:

```
$TTL 86400 ; 24 hours TTL (default)
@       IN      SOA     ns1.cc2024. admin.cc2024. (
                        2023112101 ; Serial (YYYYMMDDXX format)
                        3600       ; Refresh
                        1800       ; Retry
                        1209600    ; Expire
                        86400      ; Minimum TTL
)

; Nameservers
@       IN      NS      ns1.cc2024.

; Reverse PTR Records
10      IN      PTR     agent20.cc2024.
```

Figura 2.18: zona para *DNS* reverso

Neste exemplo temos a zona do agent20, em que o ficheiro é denominado de 10.0.5.rev, para se referir à subrede 10.0.5.0/24 em que o agent20 reside. Ao identifica-lo com o 10, temos o endereço completo do agente, 10.0.5.10, pelo que um *lookup* por 10.0.5.10 devolverá o identificador deste agente.

Esta funcionalidade é utilizada para que os agentes possam ser instanciados sem qualquer tipo de configuração dada pelo utilizador ou *hardcoded* no seu código. Assim, os agentes ao inicializar descobrem o seu próprio endereço e fazem a resolução reversa do mesmo, descobrindo o seu *ID*, que será utilizado para todas as interações com o servidor.

2.7 Interface Gráfica

A interface gráfica é implementada através de um servidor *web* baseado em *Flask*. Esta tecnologia foi escolhida por ser relativamente simples de implementar um servidor que através de *html*, *css* e algum *javascript* nos disponibiliza uma *interface* capaz de disponibilizar informação de quais agentes estão ligados, que tarefas lhes foram atribuídas, assim como as métricas e alertas recolhidos para cada. Tudo isto é feito em tempo real através de *javascript*, fazendo atualizações à página a cada dois segundos sem que o utilizador tenha de fazer *refresh*.

Esta página é naturalmente acedida por *HTTP*, podendo usar-se o *firefox* dentro do *CORE* para visualizar o sistema em tempo real.

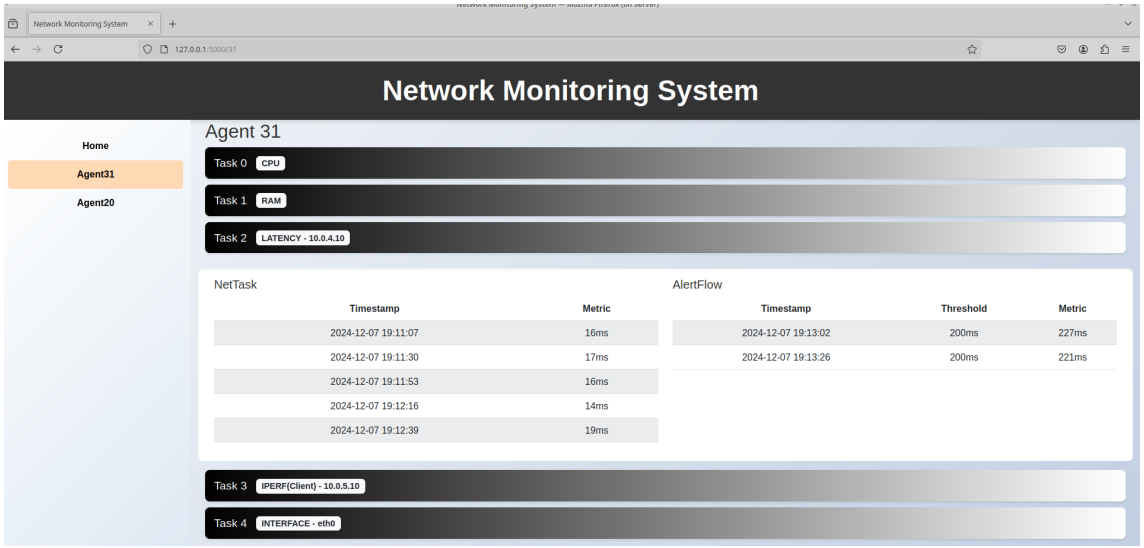

Figura 2.19: Interface Gráfica

A imagem acima mostra a *interface* gráfica desenvolvida, cujo nome é **Network Monitoring System**.

Na coluna lateral esquerda da *interface*, encontra-se uma barra de navegação que organiza os elementos disponíveis no sistema. O primeiro, denominado *Home*, leva o utilizador página inicial do sistema. Abaixo, é apresentada uma lista de agentes ativos, identificados como Agent31, Agent20, por exemplo.

Selecionando um agente, na imagem está selecionado o agente 31, é possível observar as tarefas que ele tem de realizar e o tipo dessa tarefa. Por exemplo, o agente 31 tem de realizar a sua tarefa 0, que é uma medição de CPU. Dentro de cada tarefa é possível observar do lado esquerdo as métricas registadas e quando foram registadas., por baixo do campo *Metric* e *Timestamp* em *NetTask*, respetivamente. Do lado direito de cada tarefa, é possível ver as métricas que excederam o *Threshold*, o valor do *Threshold* e quando este valor foi excedido, nos campos *Metric*, *Threshold* e *Timestamp* a baixo de *AlertFlow*.

2.8 JSON

Optámos por armazenar as tarefas de cada agente num ficheiro dedicado exclusivamente a ele. Esta abordagem foi escolhida para melhorar a **organização**, facilitando a identificação e consulta das tarefas de cada agente e para **otimizar o desempenho** de forma simples, uma vez que permite acesso simultâneo sem conflitos, melhorando a concorrência no sistema. Além disso, a **manutenção tornou-se mais simples**, pois o isolamento dos dados de cada agente facilita a identificação e resolução de problemas.

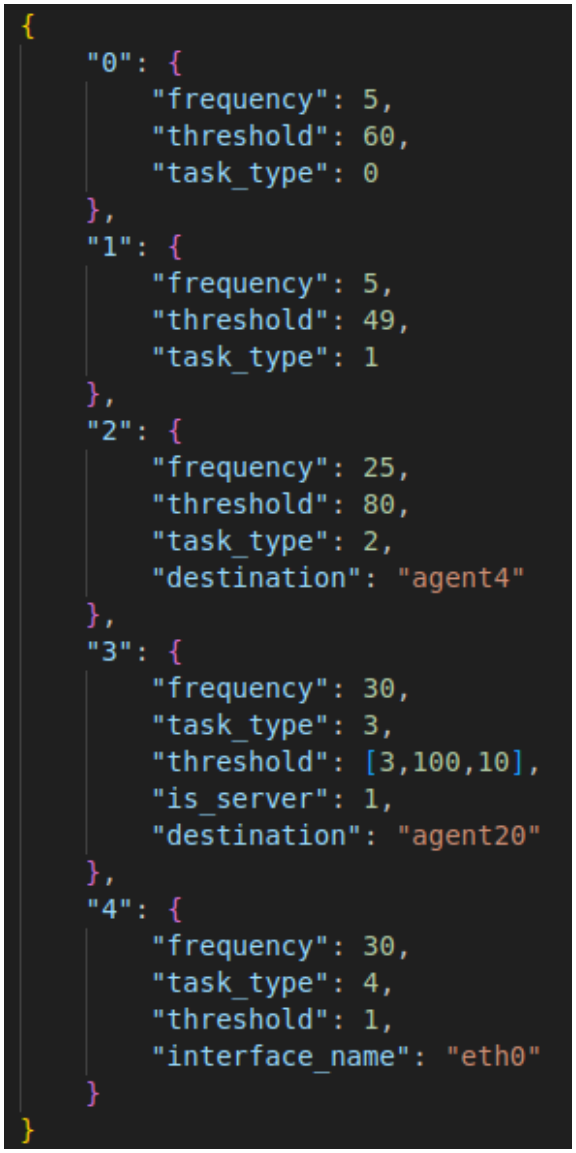

Figura 2.20: Ficheiro *JSON* que representa as tarefas de um agente.

O nome do ficheiro, onde está presente o conteúdo da imagem acima, é agent20.json. Através do nome do ficheiro, é possível identificar o agente associado (neste caso, 20), permitindo que o servidor envie as tarefas ao agente correto.

No ficheiro *JSON* apresentado, os valores entre aspas na coluna mais à esquerda representam o campo *Task Identifier* (Identificador da Tarefa). Cada identificador é único dentro do ficheiro e está associado a uma tarefa específica. Por exemplo, "0": indica que o identificador da tarefa é 0.

Os detalhes de cada tarefa estão representados na estrutura *JSON*:

- Tarefa com identificador 0: medição da utilização de *CPU* ("task_type": 0), de acordo

com a Figura 2.4. Esta métrica será avaliada a cada 5 segundos ("frequency": 5) e não deverá exceder o limite de 60% ("threshold": 60), caso contrário, será acionado um alerta.

- Tarefa com identificador 1: medição da utilização de *RAM* ("task_type": 1). Esta medição será realizada a cada 5 segundos ("frequency": 5) e deve permanecer abaixo de 49% ("threshold": 49).

- Tarefa com identificador 2: medição da latência ("task_type": 2). A métrica será avaliada a cada 25 segundos ("frequency": 25), com um limite de 80 milissegundos ("threshold": 80). Esta medição será feita no agente agent4 ("destination": "agent4"). Note-se que o campo "destination"não contém um endereço IP, mas sim o nome do agente, que será resolvido através de uma consulta ao servidor *DNS*.

- Tarefa com identificador 3: medição de JITTER ("task_type": 3). Inclui:

  - Limite máximo de *throughput*: 100 Mbps ("threshold"[1]: 100);

  - Limite máximo de *jitter*: 10 ms ("threshold"[2]: 10);

  - Limite máximo de perdas de pacotes: 3% ("threshold"[0]: 3).

  Esta tarefa será executada a cada 30 segundos ("frequency": 30) no agente agent20 ("destination": "agent20") e está marcada como uma tarefa de servidor ("is_server": 1).

- Tarefa com identificador 4: monitorização de uma *interface* de rede ("task_type": 4). Esta monitorização será realizada a cada 30 segundos ("frequency": 30) e avaliada para a *interface* eth0 ("interface_name": "eth0"). Pretende-se verificar se esta se encontra ativa e receber um alerta, caso deixe de estar, ("threshold": 1). Caso se quisesse receber um alerta, quando esta ficasse ativa, o valor do campo do *threshold* deveria ser 0, ("threshold": 0).

2.9 Logging

Uma das componentes mais importantes de qualquer projeto/aplicação é a capacidade da mesma nos dar as informações corretas para que seja possível efetuar qualquer tipo de *troubleshooting*. O conteúdo que é impresso no terminal é volátil e rapidamente se torna impossível de acompanhar.

Dada esta situação, implementou-se um mecanismo de *logging* paralelizado em que todos os eventos importantes são escritos para o terminal assim como persistidos em disco. O servidor escreve um ficheiro de *log* por cada agente que se ligue (agentX.log), e os agentes escrevem o seu próprio ficheiro. Cada entrada nos ficheiros é etiquetada com um *timestamp* e com um nível de severidade (*DEBUG*,*INFO*,*WARNING*,*CRITICAL*, etc) mediante a informação contida.

Deste modo, é possível averiguar o que poderá ter acontecido no caso de um *crash* da aplicação ou comportamento anómalo, mesmo que esta já não esteja a correr. Da mesma maneira, podemos rapidamente pesquisar por certos eventos, auxiliando bastante o *troubleshooting* de qualquer utilizador. Abaixo segue-se um exemplo de um ficheiro de *log* produzido com este formato:

```
2024-12-07 19:07:35,165 - WARNING - TIMEOUT while sending packet with seq=28
2024-12-07 19:07:35,697 - INFO - RECEIVED INTERFACE METRIC from AGENT20 SEQ=207 LENGTH=5 TASK_ID=4 METRIC=U
2024-12-07 19:07:35,706 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=212 LENGTH=6 TASK_ID=0 METRIC=11%
2024-12-07 19:07:36,006 - DEBUG - SENDING ACK SEQ=218
2024-12-07 19:07:37,183 - WARNING - TIMEOUT while sending packet with seq=28
2024-12-07 19:07:37,266 - DEBUG - RECEIVED ACK SEQ=39
2024-12-07 19:07:38,167 - INFO - RECEIVED CPU METRIC from AGENT4 SEQ=39 LENGTH=6 TASK_ID=0 METRIC=9%
2024-12-07 19:07:38,167 - INFO - RECEIVED RAM METRIC from AGENT4 SEQ=45 LENGTH=6 TASK_ID=1 METRIC=54%
2024-12-07 19:07:38,188 - DEBUG - SENDING ACK SEQ=51
2024-12-07 19:07:40,713 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=218 LENGTH=6 TASK_ID=0 METRIC=2%
2024-12-07 19:07:41,025 - DEBUG - SENDING ACK SEQ=224
```

Figura 2.21: Exemplo de um ficheiro de log

Por último, ao inicializar o servidor e os agentes, é possível inserir como argumento o nível de *logging* a partir do qual desejamos ver informações, indo desde o nível de *DEBUG* para vermos todas as informações que são geradas, até ao *FATAL* em que apenas vemos este tipo de mensagens.

3 Testes e Resultados

Nesta secção serão detalhados os testes efetuados para que se possa comprovar o correto funcionamento das funcionalidades que foram descritas no capítulo de Desenvolvimento.

3.1 Registo, receção tarefas e envio de métricas/alertas

O primeiro teste efetuado pretende demonstrar o *log* gerado numa situação na qual o Agent20 se liga, recebe as suas tarefas do servidor, e envia as métricas recolhidas / algum alerta necessário.

```
1   15:48:38,509 - INFO - Agent20 started.
2   15:48:38,511 - INFO - SENDING SEQ=0, MESSAGE_TYPE=0, LENGTH=3 bytes
3   15:48:38,514 - DEBUG - RECEIVED ACK SEQ=3
4   15:48:38,515 - INFO - RECEIVED TASK SEQ=3 LENGTH=7 ID=0 TASK_TYPE=CPU FREQUENCY=5 THRESHOLD=60
5   15:48:38,516 - INFO - RECEIVED TASK SEQ=10 LENGTH=7 ID=1 TASK_TYPE=RAM FREQUENCY=5 THRESHOLD=49
6   15:48:38,519 - INFO - RECEIVED TASK SEQ=17 LENGTH=15 ID=3 TASK_TYPE=IPERF IP=10.0.5.10 FREQUENCY=30 THRESHOLD=[BANDWIDTH:3Mbps, JITTER:100ms,
7   15:48:38,521 - INFO - RECEIVED TASK SEQ=32 LENGTH=11 ID=2 TASK_TYPE=LATENCY DESTINATION=10.0.7.10 FREQUENCY=25 THRESHOLD=80
8   15:48:38,523 - INFO - RECEIVED TASK SEQ=43 LENGTH=17 ID=4 TASK_TYPE=INTERFACE INTERFACE=eth0 FREQUENCY=30
9   15:48:38,612 - DEBUG - SENDING ACK SEQ=60
10  15:48:43,526 - WARNING - RAM threshold surpassed for task 1, measured 53.6% with threshold 49%
11  15:48:43,526 - INFO - CPU task 0 measured cpu usage of 0.8%
12  15:48:43,530 - INFO - SENDING SEQ=60, MESSAGE_TYPE=3, LENGTH=6 bytes
13  15:48:43,534 - INFO - SENDING SEQ=66, MESSAGE_TYPE=3, LENGTH=5 bytes
14  15:48:43,545 - DEBUG - RECEIVED ACK SEQ=71
15  15:48:47,555 - INFO - LATENCY task 2 measured a latency of 46ms to 10.0.7.10
16  15:48:47,556 - INFO - SENDING SEQ=71, MESSAGE_TYPE=3, LENGTH=6 bytes
17  15:48:47,563 - DEBUG - RECEIVED ACK SEQ=77
18  15:48:48,529 - WARNING - RAM threshold surpassed for task 1, measured 53.7% with threshold 49%
19  15:48:48,536 - INFO - CPU task 0 measured cpu usage of 0.9%
20  15:48:48,537 - INFO - SENDING SEQ=77, MESSAGE_TYPE=3, LENGTH=6 bytes
21  15:48:48,565 - DEBUG - RECEIVED ACK SEQ=83
22  15:48:53,537 - WARNING - RAM threshold surpassed for task 1, measured 53.7% with threshold 49%
23  15:48:53,538 - CRITICAL - 3 consecutive fails on RAM task 1
```

Figura 3.1: Teste 1 - Exemplo de tarefas recebidas sem perdas no lado do agente

Na linha 2 vemos o primeiro pacote enviado pelo agent20, correspondendo ao seu registo e com um tamanho de 3 *bytes*. Este é confirmado pelo agente com um *ACK* 3.

De seguida, o servidor envia as *tasks* que estão associadas a este agente. Podemos confirmar que os números de sequência são atualizados mediante o tamanho de cada um dos pacotes enviados. Adicionalmente, também se constata que a funcionalidade de envio de *ACKs* cumulativos está funcional, uma vez que o agente ao receber ordenadamente todos os pacotes de tasks apenas envia um *ACK* 60, confirmando o pacote de seq=43 com 17 *bytes* de dados, assim como todos os recebidos anteriormente.

Começa assim a correr as suas *tasks*, tendo métricas dentro do esperado para o *CPU* (linha 11) e latência (linha 15), mas gerando um alerta para a tarefa de medição de *RAM*, que mediu 53.6% com um *threshold* de 49%.

Confirma-se também que a funcionalidade de paragem de tarefas quando existem 3 falhas consecutivas está funcional, observando-se uma entrada de gravidade *CRITICAL* na linha 23 para a task de medição de *RAM*. Neste caso, além do alerta enviado por *AlertFlow*, a tarefa é também terminada.

```
1   2024-12-07 15:48:38,512 - INFO - Agent 20 logged in.
2   2024-12-07 15:48:43,543 - DEBUG - SENDING ACK SEQ=3
3   2024-12-07 15:48:38,514 - INFO - SENDING SEQ=3, MESSAGE_TYPE=2, LENGTH=7 bytes
4   2024-12-07 15:48:38,514 - INFO - SENDING SEQ=10, MESSAGE_TYPE=2, LENGTH=7 bytes
5   2024-12-07 15:48:38,516 - INFO - SENDING SEQ=17, MESSAGE_TYPE=2, LENGTH=15 bytes
6   2024-12-07 15:48:38,519 - INFO - SENDING SEQ=43, MESSAGE_TYPE=2, LENGTH=17 bytes
7   2024-12-07 15:48:38,517 - INFO - SENDING SEQ=32, MESSAGE_TYPE=2, LENGTH=11 bytes
8   2024-12-07 15:48:38,615 - DEBUG - RECEIVED ACK SEQ=60
9   2024-12-07 15:48:43,528 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=60 LENGTH=6 TASK_ID=0 METRIC=1%
10  2024-12-07 15:48:43,535 - INFO - RECEIVED INTERFACE METRIC from AGENT20 SEQ=66 LENGTH=5 TASK_ID=4 METRIC=UP
11  2024-12-07 15:48:43,543 - DEBUG - SENDING ACK SEQ=71
12  2024-12-07 15:48:43,572 - WARNING - ALERT received for task 1 from Agent20, TASK_TYPE=1, METRIC=54%
13  2024-12-07 15:48:47,558 - INFO - RECEIVED LATENCY METRIC from AGENT20 SEQ=71 LENGTH=6 TASK_ID=2 METRIC=46ms
14  2024-12-07 15:48:47,561 - DEBUG - SENDING ACK SEQ=77
15  2024-12-07 15:48:48,538 - WARNING - ALERT received for task 1 from Agent20, TASK_TYPE=1, METRIC=54%
16  2024-12-07 15:48:48,538 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=77 LENGTH=6 TASK_ID=0 METRIC=1%
17  2024-12-07 15:48:48,564 - DEBUG - SENDING ACK SEQ=83
18  2024-12-07 15:48:53,545 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=83 LENGTH=6 TASK_ID=0 METRIC=0%
19  2024-12-07 15:48:53,583 - WARNING - ALERT received for task 1 from Agent20, TASK_TYPE=1, METRIC=54%
20  2024-12-07 15:48:53,601 - DEBUG - SENDING ACK SEQ=89
21  2024-12-07 15:48:58,550 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=89 LENGTH=6 TASK_ID=0 METRIC=1%
22  2024-12-07 15:48:58,618 - DEBUG - SENDING ACK SEQ=95
23  2024-12-07 15:49:03,557 - INFO - RECEIVED CPU METRIC from AGENT20 SEQ=95 LENGTH=6 TASK_ID=0 METRIC=1%
24  2024-12-07 15:49:03,637 - DEBUG - SENDING ACK SEQ=101
```

Figura 3.2: Teste 1 - Exemplo de tarefas recebidas sem perdas no lado do servidor

A interação do lado do servidor é a esperada, refletindo o comportamento que foi verificado no agente. Demonstra-se aqui a sinergia existente entre os dois protocolos, com os alertas a detalhar as falhas sentidas na tarefa 1, ao passar o *threshold* imposto para a percentagem de utilização de *RAM*. Como os envios de métricas são mais esporádicos que o envio das tarefas, existem menos situações de *ACKs* cumulativos, mas sendo possível observar as linhas 9 e 10, que são confirmadas com um único *ACK* 71 na linha 11.

3.2 Retransmissão por Timeout

O segundo teste pretende exemplificar como é que o Agent4 se comporta quando o seu pacote inicial de registo é perdido. Pode-se ver abaixo o *log* relativo a esta interação:

```
1   2024-12-07 17:27:44,064 - INFO - Agent4 started.
2   2024-12-07 17:27:44,067 - INFO - SENDING SEQ=0, MESSAGE_TYPE=0, LENGTH=3 bytes
3   2024-12-07 17:27:46,076 - WARNING - TIMEOUT while sending packet with seq=0
4   2024-12-07 17:27:46,135 - DEBUG - RECEIVED ACK SEQ=3
5   2024-12-07 17:27:46,136 - INFO - RECEIVED TASK SEQ=3 LENGTH=7 ID=0 TASK_TYPE=CPU FREQUENCY=5 THRESHOLD=70
6   2024-12-07 17:27:46,136 - INFO - RECEIVED TASK SEQ=10 LENGTH=7 ID=1 TASK_TYPE=RAM FREQUENCY=5 THRESHOLD=60
7   2024-12-07 17:27:46,138 - INFO - RECEIVED TASK SEQ=17 LENGTH=11 ID=2 TASK_TYPE=LATENCY DESTINATION=10.0.5.10 FREQUENCY=
```

Figura 3.3: Teste 2 - Pacote de registo inicial perdido no Agent4

O pacote de registo enviado pelo Agent4 na linha 2 é perdido, como se pode ver pelo aviso de *timeout* na linha 3. Uma vez que é o primeiro pacote a ser enviado, não existe outra maneira do servidor avisar o agente desta perda. Ao atingir o *timeout*, o agente retransmite o pacote, que é corretamente recebido e confirmado pelo servidor abaixo:

```
1   2024-12-07 17:27:46,112 - INFO - Agent 4 logged in.
2   2024-12-07 17:27:46,113 - DEBUG - SENDING ACK SEQ=3
3   2024-12-07 17:27:46,114 - INFO - SENDING SEQ=3, MESSAGE_TYPE=2, LENGTH=7 bytes
4   2024-12-07 17:27:46,116 - INFO - SENDING SEQ=10, MESSAGE_TYPE=2, LENGTH=7 bytes
5   2024-12-07 17:27:46,117 - INFO - SENDING SEQ=17, MESSAGE_TYPE=2, LENGTH=11 bytes
6   2024-12-07 17:27:46,118 - INFO - SENDING SEQ=28, MESSAGE_TYPE=2, LENGTH=11 bytes
7   2024-12-07 17:27:46,216 - DEBUG - RECEIVED ACK SEQ=28
```

Figura 3.4: Teste 2 - Receção e envio de ACK pelo Servidor

3.3 Retransmissão por 3 ACKs duplicados

Numa situação com mais tráfego, é possível observar que existe também a possibilidade do servidor perceber que houve a perda de uma tarefa ao receber 3 *ACKs* duplicados quando envia as tarefas seguintes.

```
1   2024-12-07 17:39:40,735 - INFO - Agent4 started.
2   2024-12-07 17:39:40,736 - INFO - SENDING SEQ=0, MESSAGE_TYPE=0, LENGTH=3 bytes
3   2024-12-07 17:39:40,792 - DEBUG - RECEIVED ACK SEQ=3
4   2024-12-07 17:39:40,805 - INFO - RECEIVED TASK SEQ=10 LENGTH=7 ID=1 TASK_TYPE=RAM FREQUENCY=5 THRESHOLD=60
5   2024-12-07 17:39:40,875 - DEBUG - SENDING ACK SEQ=3
6   2024-12-07 17:39:40,902 - INFO - RECEIVED TASK SEQ=17 LENGTH=11 ID=2 TASK_TYPE=LATENCY DESTINATION=10.0.5.10 FREQUENC
7   2024-12-07 17:39:40,905 - DEBUG - SENDING ACK SEQ=3
8   2024-12-07 17:39:40,931 - INFO - RECEIVED TASK SEQ=28 LENGTH=11 ID=3 TASK_TYPE=LATENCY DESTINATION=10.0.8.10 FREQUENC
9   2024-12-07 17:39:40,940 - DEBUG - SENDING ACK SEQ=3
10  2024-12-07 17:39:40,960 - INFO - RECEIVED TASK SEQ=3 LENGTH=7 ID=0 TASK_TYPE=CPU FREQUENCY=5 THRESHOLD=70
11  2024-12-07 17:39:40,972 - DEBUG - SENDING ACK SEQ=39
```

Figura 3.5: Teste 3 - Exemplo de 3 ACKs duplicados no lado do Agent4

Neste caso, o Agent4 não recebeu corretamente a tarefa 0 de SEQ=3, pelo que assim que recebe a tarefa de SEQ=10, envia um ACK=3 indicando esta perda. Seguidamente recebe as tarefas de SEQ=17 e SEQ=28, enviando também ACK=3 para cada uma, dado que ainda não recebeu a primeira tarefa. Ao enviar o terceiro *ACK* duplicado para o servidor, este percebe que a tarefa de SEQ=3 foi perdida, e retransmite-a. Como já tinha recebido as 3 tarefas seguintes e as manteve em *cache*, o Agent4 responde com ACK=39.

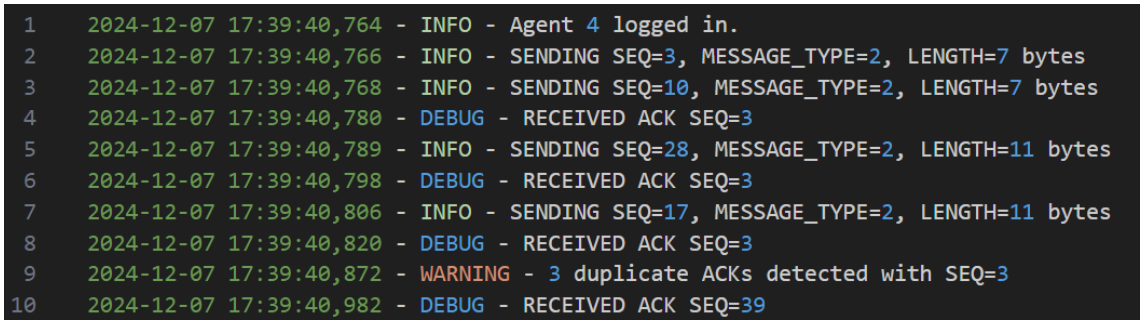

Figura 3.6: Teste 3 - Exemplo de 3 ACKs duplicados no lado do servidor

No lado o servidor temos o comportamento esperado, com uma entrada de gravidade *WARNING* quando são detetados os 3 *ACKs* duplicados. Depois desta retransmissão, é recebido o ACK=39 e todas as entregas confirmadas.

3.4 Agente é desligado

Quando um Agente é desligado, o servidor deve ser notificado, para que o possa remover do sistema. Temos abaixo a situação em que o Agent20 é desligado:

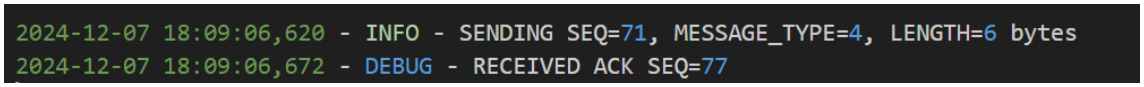

Figura 3.7: Teste 4 - Agent20 desliga-se

O Agent20 corretamente envia o pacote do tipo END para o Servidor.

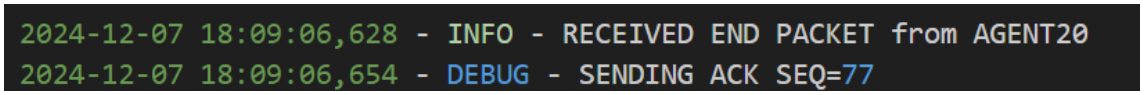

Figura 3.8: Teste 4 - Servidor recebe o END do Agent20

O servidor ao receber o *END* do Agent20 apaga a informação mantida em memória sobre o mesmo e confirma a receção com um *ACK* 77.

3.5 Servidor é desligado

Quando o servidor é desligado, este encarrega-se de avisar todos os agentes para não continuarem a enviar mais métricas e informações do estado da rede. Abaixo está o *log* referente à saída do servidor:

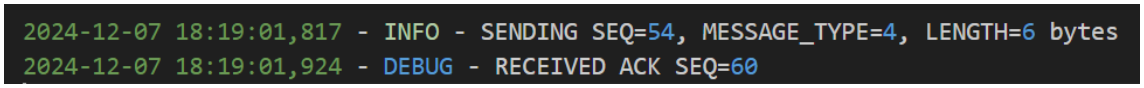

Figura 3.9: Teste 5 - Servidor envia END a Agent20

Como se pode constatar, o Servidor corretamente envia um pacote do tipo *END* ao Agent20 quando é desligado, garantindo assim que não há tráfego adicional gerado quando não existe servidor para o recolher.

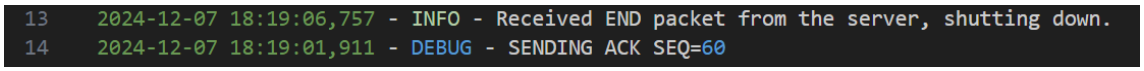

Figura 3.10: Teste 5 - Agent20 recebe END do servidor

Ao receber o *END* do Servidor, o Agent20 confirma a sua receção e desliga-se também.

4 Conclusões e Trabalho Futuro

Este projeto focou-se na implementação de um sistema de monitorização de redes distribuído, utilizando protocolos aplicacionais desenvolvidos especificamente para este propósito. O sistema foi projetado para operar em condições adversas, simuladas pelo *emulador CORE*, onde foi essencial garantir a fiabilidade na comunicação entre os agentes e o servidor. Para tal, implementaram-se dois protocolos: *NetTask*, que utiliza *UDP* com mecanismos adicionais de retransmissão e gestão de sequência, e *AlertFlow*, baseado em *TCP* para notificações críticas.

Ao longo do desenvolvimento, foram implementados diversos componentes para operacionalizar este sistema de monitoria de rede, incluindo o processamento de mensagens binárias, gestão de tarefas de monitorização, reação a eventos críticos na rede. O sistema demonstrou capacidade para recolher métricas como latência, utilização de *CPU* e largura de banda, assegurando a entrega de mensagens e a deteção de falhas através da implementação de alguns dos mecanismos do protocolo *TCP*. A utilização de *threads* e mecanismos de sincronização, como *locks* e condições, permitiu implementar uma arquitetura concorrente eficiente, mas evidenciou desafios na gestão de concorrência e na integração de diferentes componentes.

A maior contribuição para o nosso conhecimento foi a implementação de um cabeçalho corretamente, assim como os mecanismos do *TCP* que tivemos de adequar às nossas necessidades para conseguir construir um protocolo fiável sobre uma rede inerentemente não fiável. Através da construção destes mecanismos temos agora uma visão muito mais clara das diferenciações entre o *TCP* e o *UDP*.

Como adições ao estipulado no enunciado, foi implementado um sistema de resolução de nomes (*DNS*) incluindo a resolução reversa, assim como um servidor web baseado em Flask, para que o projeto incorporasse tanto a parte de *DNS* como a parte de *HTTP*.

Não obstante, existem algumas lacunas no trabalho entregue, das quais o nosso grupo está ciente, mas que não teve oportunidade de aperfeiçoar. Quando o servidor é parado e envia mensagem aos agentes para também pararem, estes ainda continuam a enviar as suas métricas e alertas enquanto o processo de paragem está a decorrer, pelo que deveriam cessar imediatamente de enviar tráfego.

Como trabalho futuro, sugere-se melhorar a escalabilidade e otimizar a gestão de retransmissões em cenários de elevada carga. Aliado a estas melhorias, também poderia ser implementada segurança nos pacotes trocados, que não foi efetuado por falta de tempo. A ideia seria a geração de chaves assimétricas para encriptação dos dados e potencial garantia de que estavamos efetivamente a falar com o dispositivo correto. Adicionalmente, ainda se poderia

implementar a *interface web* com *HTTPS*, ainda que com um certificado self-signed. Este projeto proporcionou uma experiência prática valiosa, consolidando conhecimentos sobre redes distribuídas, programação de *sockets* e sistemas resilientes.