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

## Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

# Monitorização Distribuída de Redes

Ana Pires(a96060) José Rodrigo Matos(a100612) Rúben Oliveira(a93625)

7 de dezembro de 2024

CC

| | |
|---|---|
| Data de Receção | |
| Responsável | |
| Avalição | |
| Observações | |

# Monitorização Distribuída de Redes

Ana Pires(a96060) José Rodrigo Matos(a100612) Rúben Oliveira(a93625)

7 de dezembro de 2024

# Resumo

Este relatório apresenta o desenvolvimento de uma aplicação modular e eficiente para monitorização de agentes numa rede, com comunicação entre agentes e servidor suportada por dois protocolos personalizados: **NetTask**, utilizando UDP, e **AlertFlow**, baseado em TCP. O objetivo principal foi criar uma solução capaz de lidar com falhas em rede e retransmissões de dados, ao mesmo tempo que fornece monitorização em tempo real de métricas importantes, como o uso de CPU e RAM, e geração de alertas.

A arquitetura da aplicação foi concebida de forma a separar claramente as camadas de cliente, servidor e protocolos, garantindo modularidade e reusabilidade. A implementação foi realizada em **Go**, escolhida pela sua eficiência em aplicações de rede e suporte nativo a programação concorrente.

# Índice

# Lista de Figuras

# 1 Introdução

Este trabalho prático visa o desenvolvimento de um NMS distribuído que, baseado no modelo cliente-servidor, deverá empregar duas aplicações principais: o NMS_Agent, responsável pela coleta de métricas, e o NMS_Server, que centraliza o processamento e análise dos dados recebidos. Para a comunicação entre estas aplicações, derverão ser desenvolvidos dois protocolos aplicacionais distintos:

NetTask: utilizando o protocolo UDP, para a coleta contínua de métricas e a execução de tarefas de monitorização.

AlertFlow: utilizando o protocolo TCP, para notificação imediata de alterações críticas no estado dos dispositivos.

O projeto visa proporcionar oportunidade do grupo se familiarizar com o desenvolvimento de protocolos aplicacionais e a utilização de sockets para comunicação em rede. Além disso, pretende-se explorar aspectos de resiliência e robustez em soluções distribuídas, estimulando a criação de uma abordagem própria e inovadora para a monitorização de redes, inspirada no protocolo NetFlow, mas com características únicas desenvolvidas pelo grupo.

# 2 Desenvolvimento

## 2.1 Arquitetura da solução

De modo a ilustrar a arquitetura do programa e facilitar o entendimento de sua estrutura e funcionamento, apresentam-se a seguir alguns diagramas que traduzem visualmente os principais componentes e suas interações. Na decisão da estrutura da solução, o grupo optou por separar as componentes do programa pelos seus respetivos módulos.

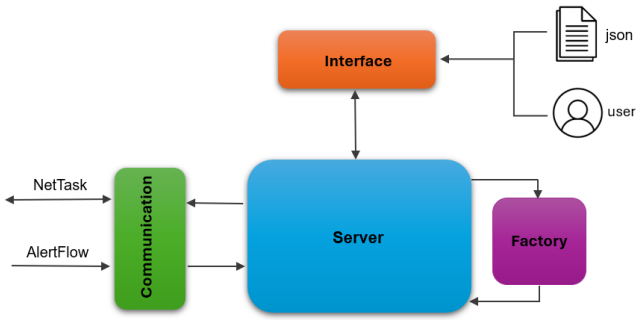



Figura 2.1: Arquitetura do servidor e as suas conexões

Neste primeiro diagrama, é possível perceber detalhadamente a arquitetura do servidor e as suas conexões com as demais componentes. Primeiramente, o utilizador fornece tarefas no formato JSON via interface. A *interface*, por sua vez, atua como o ponto de interação com o utilizador, pois comunica essas tarefas ao servidor. O *factory* é responsável por criar uma instância do servidor. Isto é importante, uma vez que permite que o servidor mantenha o seu estado durante a execução. A componente de *communication* é o meio do servidor interagir com agentes externos, empregando os protocolos apropriados, neste caso o NetTask e AlertFlow.

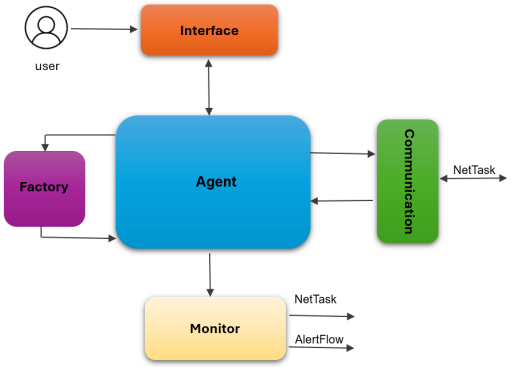



Figura 2.2: Arquitetura do agente e as suas conexões

No segundo diagrama, observa-se a mesma lógica que o anterior, no entanto, centraliza-se no agente. O utilizador interage, então, com a *interface*, que as envia para o agente. Mais uma vez, o *factory* atua, criando ,agora, uma instância do agente, permitindo que este mantenha o seu estado (como por exemplo, as várias tarefas em execução). Com a componente *communication*, os agentes comunicam com o servidor utilizando o protocolo NetTask. Por sua vez, o *monitor* supervisiona as operações em andamento, executando as tarefas e monitorizando a rede, enviando os resultados através dos protocolos NetTask e AlertFlow.

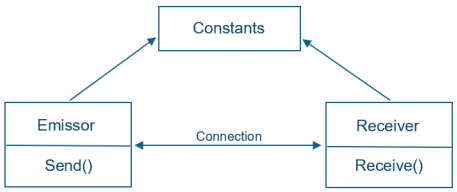



Figura 2.3: NetTask

Este terceiro esquema ilustra de forma mais específica como os emissores e os recetores (como o servidor e o agente) conectam entre si no protocolo NetTask. A sua conexão é bidirecional, uma vez que ambos entendem mensagens de controlo, um enviando-as (emitter) e outro recebendo-as (receiver). Por sua vez, ambos estão ligados a *Constants*, pois é lá que se encontram os parâmetros e definições importantes dos protocolos. Não é apresentado o AlertFlow, pois diferentemente do NetTask, não estabelece conexão. Este utiliza comunicação por TCP, apenas enviando mensagens de alertas em texto.

## 2.2 Protocolos

De modo a ser possível a comunicação entre um Servidor e um Agente, foram desenvolvidos, então, dois protocolos: AlertFlow e NetTask.

AlertFlow é um protocolo de comunicação baseado em TCP projetado para transmitir e gerir dados entre um emissor e um recetor com foco na simplicidade e eficiência. O protocolo segue um modelo claro de envio e recepção: o emissor é responsável por estabelecer uma conexão TCP com o recetor, enviar dados, e garantir que a conexão seja encerrada corretamente após a transmissão. Por sua vez, o recetor escuta conexões recebidas numa porta específica. Este protocolo inclui, também, um mecanismo de manipulação para processar conexões de entrada. Ele lê os dados enviados e encaminha tanto os dados, como possíveis erros, para canais específicos, garantindo que nenhuma falha passe despercebida.

NetTask é um protocolo de comunicação baseado em UDP que visa oferecer confiabilidade em redes não confiáveis, onde a perda de pacotes é uma preocupação. O protocolo implementa mecanismos de retransmissão e recuperação para garantir a entrega dos dados, mesmo em situações adversas. Neste caso, o emissor fornece funcionalidades para enviar dados via UDP, utilizando mecanismos de retransmissão de pacotes para lidar com falhas na entrega e recuperação de pacotes perdidos, garantindo que os dados alcancem o destino. Por sua vez, o recetor implementa funcionalidades para receber mensagens UDP, com suporte para retransmissão de pacotes solicitada ao emissor em caso de perda e mecanismos de recuperação, que reconstroem a integridade dos dados recebidos, mesmo em redes não confiáveis.

As restransmissões do protocolo NetTask são tratadas consoante o *timeout*. Na solução implementada, foi decidido que um *packet timeout* acontece quando a confirmação de receção de um pacote excede os 2 segundos. Assim sendo, por cada tempo excedido é efetuada a retransmissão dos pacotes. O grupo estipulou que o número máximo de retransmissões seguidas será 5. Após esse valor, acontece um *reset* da comunicação. No caso do recetor, este espera sempre os 2 segundos em qualquer transmissão, para considerar *timeout*. Porém, decidiu-se que há um caso específico em que o emissor espera mais 3 segundos extra, ou seja 5 segundos. Isto acontece apenas quando este espera a receção do último ACK (*Ack Complete*). Esta estratégia foi adotada para melhorar a garantia que a comunicação correu bem. Se o emissor mantivesse os 2 segundos, poderia acontecer deste fechar a comunicação, enquanto que acontecia um *timeout* por parte do recetor e fosse pedida uma retransmissão (que nunca chegaria, pois a conexão já estaria encerrada). Ao se incluir mais 3 segundos, o emissor dá tempo para perceber se algo não chega ao recetor, encerrando a comunicação apenas quando tudo ocorre com sucesso.

### 2.2.1 Formato das mensagens

Nesta secção serão aprensentadas as mensagens protocolares e próprias da aplicação.

No caso do protocolo AlertFlow, não é necessário apresentar o formato das mensagens. Como é baseado em TCP, ele opera de um modo simples e direto, onde o emissor apenas envia os dados, e o receptor tem ações predefinidas (aceitar/processar ou rejeitar). Não existe a necessidade de interpretar ou estruturar mensagens complexas.

O protocolo NetTask, nesta solução, tem os seguintes tipos de mensagens:

```
%d|{...}
RECOVERY|%d
FAST_RECOVERY|%d
AGREEMENT|%d
ACK_AGREEMENT|%d
ACK_COMPLETE
```

- **Sequence** - Mensagem que indica o número de sequência do pacote.
- **Recovery** - Mensagem para solicitar a recuperação de um pacote específico (indicado depois da barra).
- **Fast Recovery** - Mensagem para solicitar a recuperação rápida para retransmissão de pacotes ausentes (sendo o primeiro pacote especificado depois da barra).
- **Agreement** - Formato para um pacote de acordo (especificado depois da barra).
- **Ack Agreement**- Enviado para confirmar o acordo inicial para transmissão de pacotes.
- **Ack Complete** - Enviado para indicar a conclusão bem-sucedida da transmissão.

Para a aplicação, existem as mensagens:

```
REGISTER|" + a.ID + ":" + a.IPAddr
TASK|{...}
DATA|{...}
```

- **Register** - Mensagem que informa o servidor da presença (identificada pelo ID e endereço IP) de um novo agente.
- **Task** - Mensagem que informa que traz uma tarefa para o agente.
- **Data** - Mensagem que informa os outputs das tarefas de um agente.

### 2.2.2 Troca de mensagens

Nesta secção, apresentar-se-ão os diagramas de sequência ilustrativos das trocas de mensagens. Primeiramente, mostram-se exemplos de mensagens gerais que os protocolos permitem utilizar na comunicação entre um emissor e um recetor.

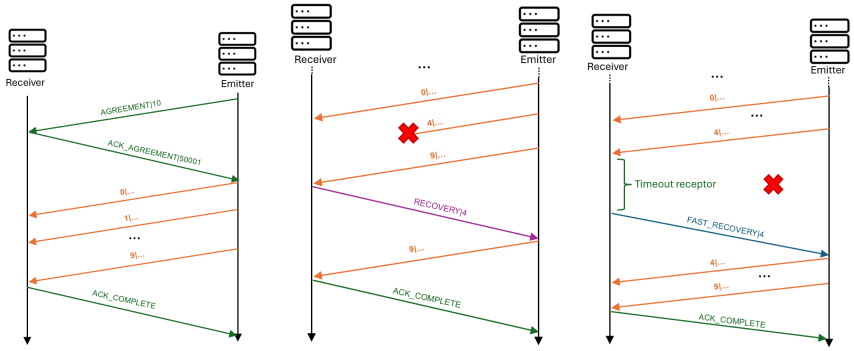



Figura 2.4: Trocas de mensagens possíveis com os protocolos

O primeiro diagrama corresponde a um início de comunicação e troca de mensagens com sucesso. O segundo diagrama exemplifica um caso da perda de um pacote, e onde foi pedida e efetuada uma retransmissão. Finalmente, o terceiro diz respeito a uma comunicação onde aconteceu um *timeout* no recetor, resultando num pedido de *Fast Recovery*.

Seguidamente, apresentam-se exemplos de mensagens específicas da aplicação.

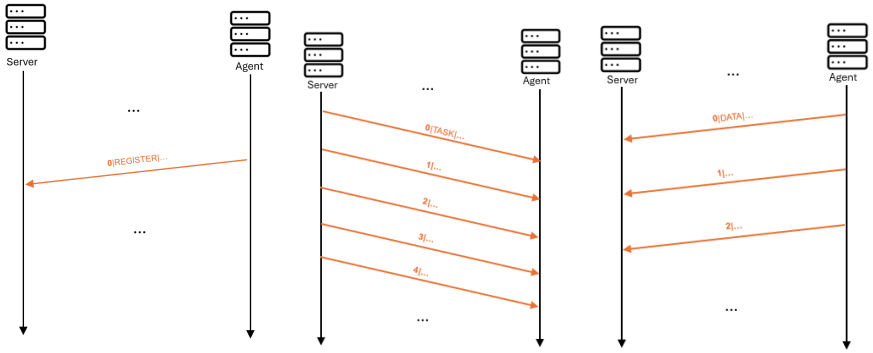



Figura 2.5: Exemplos de mensagens interiores à aplicação

O primeiro diagrama mostra uma mensagem em que houve um registo de um agente. Como falado na secção anterior, a mensagem trará consigo mais informação sobre o agente (como o ID e o seu endereço IP). O segundo diagrama apresenta o envio de mensagens de tarefas. Como se pode observar, apenas o primeiro pacote leva a especificação que os restantes são *Tasks*. Estas podem ser no máximo 5. Por fim, o último, exemplifica o envio de pacotes de *Data*, que podem ser no máximo 3.

## 2.3 Implementação

A implementação da aplicação seguiu rigorosamente a arquitetura definida e explicada no capítulo anterior.

A linguagem escolhida para o desenvolvimento foi **Go**, devido à sua versatilidade e especialização em aplicações que envolvem comunicações de rede. Esta escolha permite tirar partido de várias bibliotecas específicas para redes, bem como de **goroutines** — uma funcionalidade nativa do Go que facilita a programação concorrente, tornando-a mais eficiente e simples de implementar.

Para a comunicação, foi utilizada a biblioteca padrão do Go, **net**, que oferece as ferramentas necessárias para estabelecer conexões **UDP** e **TCP** de forma robusta. Adicionalmente, foi integrada a biblioteca externa **gopsutil**, que implementa as funcionalidades da conhecida biblioteca **psutil** em Go. Esta biblioteca foi crucial para monitorizar métricas importantes nos agentes, como o uso de CPU e memória RAM, dados fundamentais para os alertas.

As goroutines tiveram um papel central na implementação, uma vez que permitiram tratar cada conexão de forma concorrente, criando um ambiente de execução paralelo. Esta abordagem foi essencial para a aplicação, já que tanto os agentes quanto o servidor precisam lidar com múltiplas conexões e tarefas simultaneamente. Além disso, o uso das goroutines simplificou significativamente a implementação ao tirar proveito do modelo nativo de concorrência da linguagem Go.

Outro ponto importante na implementação foi a separação entre os protocolos desenvolvidos e a lógica aplicacional. Os protocolos **NetTask** e **AlertFlow** foram isolados no módulo **pkg**, tornando-os independentes do restante da aplicação. Esta abordagem modular permitiu que os protocolos fossem reutilizáveis em outras aplicações, aumentando a flexibilidade e escalabilidade do projeto.

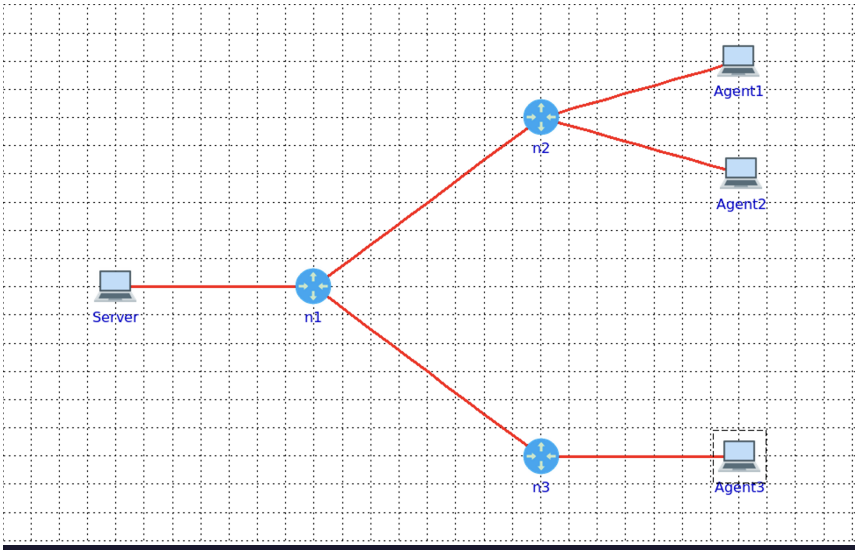
Adicionalmente, foi feita uma clara distinção entre a camada do servidor e a camada do cliente, implementados como programas independentes. Essa separação garante que ambos possam evoluir separadamente, desde que respeitem as regras de comunicação e mensagens definidas pelos protocolos. Como resultado, a aplicação apresenta uma estrutura modular e adaptável, na qual componentes podem ser ajustados ou substituídos para atender a diferentes necessidades.

## 2.4 Testes

Os testes realizados envolveram dois cenários distintos, cada um utilizando uma topologia definida no **CORE** (Common Open Research Emulator). Apesar de estruturalmente semelhantes, os cenários diferem nas configurações das ligações entre agentes e o servidor:

- **Cenário 1**: Representa uma rede ideal, sem falhas ou limitações nos links.
- **Cenário 2**: Simula a mesma rede, mas com falhas e restrições nas conexões.

No primeiro cenário, onde a rede opera sem qualquer tipo de falhas ou limitações, a aplicação mostrou um desempenho impecável. Todas as comunicações ocorreram de forma rápida e eficiente, como esperado devido à estabilidade do ambiente de rede e à arquitetura bem projetada da aplicação e dos protocolos. Este resultado demonstra que a solução é altamente eficaz em redes com condições ideais.




```
----- Jitter Report -----
Interval: 0.0000-10.0100, Transfer: 2.51 MBytes, Jitter: 0.004 ms

----- Packet Loss Report -----
Interval: 0.0000-10.0156, Transfer: 1.25 MBytes, Lost/Total: 0/895 (0%)

----- Ping Report -----
transmitted: 10
received: 10
packet loss: 0.00%
time elapsed: 9219ms
latency (min/avg/max/mdev): 0.025/0.031/0.065/0.011 ms

----- Bandwidth Report -----
Interval: 0.0000-10.0153, Transfer: 1.25 MBytes, Bandwidth: 1.05 Mbits/sec
```

Figura 2.6: Testes num cenário ótimo

No segundo cenário, foram introduzidas falhas e limitações nos links, como menor largura de banda e perda de pacotes. Estas condições revelaram fraquezas significativas na arquitetura e nos protocolos implementados. Em particular, as seguintes limitações foram identificadas:

- **Formato das mensagens**: O protocolo utiliza strings como base para os dados transmitidos, o que aumenta o overhead.

- **Fragmentação dos pacotes**: Cada pacote é dividido em partes muito pequenas (aproximadamente 512 bytes) e enviado simultaneamente, gerando congestionamento severo na rede.

Estes fatores combinados causam um tráfego excessivo, podendo congestionar a rede a tal ponto que a aplicação se torna ineficaz em redes com condições degradadas. Em casos extremos, o tráfego gerado pode até simular um ataque **Denial of Service (DoS)**, sobrecarregando os recursos da rede.

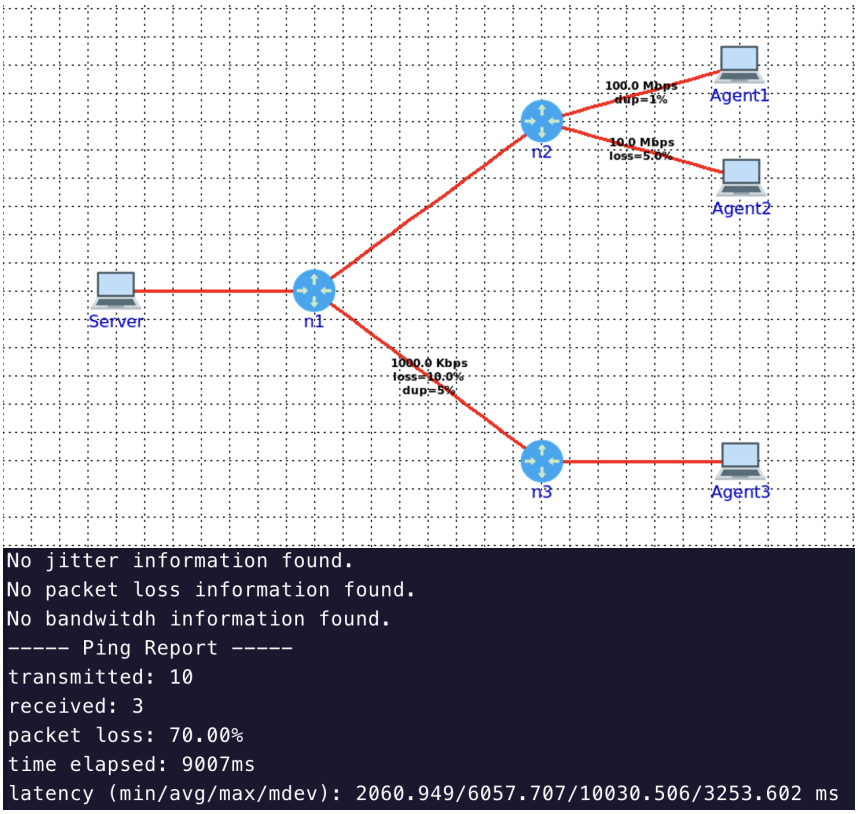



Figura 2.7: Testes num cenário frágil

Os testes evidenciaram que a aplicação apresenta ótimo desempenho em redes robustas, operando de forma rápida. No entanto, em redes frágeis, as limitações dos protocolos e da arquitetura tornam-se evidentes.

# 3 Conclusões e Trabalho Futuro

A aplicação desenvolvida cumpre os objetivos principais, demonstrando ser capaz de lidar com retransmissões de dados e falhas em rede, garantindo comunicação confiável entre agentes e o servidor. Além disso, permite monitorizar múltiplos agentes numa rede, recolher métricas essenciais, como uso de CPU e RAM, e reportar alertas ao servidor de forma eficiente.

No entanto, os testes evidenciaram limitações importantes em redes frágeis. O uso de strings como formato de mensagem e a abordagem atual do protocolo, que fragmenta pacotes em partes pequenas enviadas simultaneamente, geram um overhead elevado e congestionam a rede, podendo até comprometer o tráfego de forma destrutiva.

Para superar estas limitações, poderia-se implementar um formato de mensagem mais eficiente, como estruturas binárias, e a reformulação do protocolo para reduzir o congestionamento.

Apesar dos problemas identificados, a aplicação representa uma solução modular e funcional, com grande potencial para evoluir e atender a mais cenários no futuro.