



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

Network Monitoring System

Edgar Ferreira
a99890

Eduardo Faria
a104353

Nuno Silva
a104089

Dezembro, 2024



Data da Receção	
Responsável	
Avaliação	
Observações	

Network Monitoring System

Edgar Ferreira

a99890

Eduardo Faria

a104353

Nuno Silva

a104089

Dezembro, 2024

Resumo

Este trabalho, desenvolvido no âmbito da unidade curricular de [Comunicações por Computador](#), teve como objetivo criar um sistema distribuído de monitorização de redes. O sistema, composto por um servidor central e agentes distribuídos, utiliza dois protocolos distintos para comunicação: o protocolo *NetTask*, baseado em UDP, e o protocolo *AlertFlow*, baseado em TCP.

O protocolo *NetTask* foi projetado para o registo dos agentes, distribuição de tarefas e recolha de métricas de desempenho dos agentes, permitindo um fluxo eficiente e contínuo de dados. Apesar das limitações do UDP, como a inexistência de garantia de entrega, foram implementados mecanismos de confiabilidade, como retransmissões e confirmações (*acknowledgments*). O protocolo *AlertFlow*, por sua vez, é responsável pelo envio confiável de alertas em situações críticas, utilizando as vantagens do TCP para garantir integridade e entrega das mensagens.

Os agentes, ao receberem tarefas, executam comandos como *ping* e *iperf* e monitorizam métricas do sistema e da conexão. Sempre que os limites predefinidos são excedidos, alertas detalhados são enviados ao servidor, que os processa e regista para análise futura. Além disso, o sistema implementa funções de serialização e desserialização de mensagens, garantindo compactação e compatibilidade entre diferentes dispositivos.

Os testes realizados demonstraram uma melhoria na eficiência do sistema na monitorização de redes distribuídas, com uma redução significativa no uso de largura de banda devido à otimização da serialização. A solução proposta é escalável e pode ser aplicada em redes de grande porte, oferecendo uma gestão eficiente de tarefas e alertas críticos.

Área de Aplicação: Monitorização de Redes e Sistemas Distribuídos.

Palavras-Chave: Sistemas Distribuídos, Protocolos de Comunicação, Socket, UDP, TCP, Concorrência, Monitorização de Redes, Servidor.

Índice

1. Introdução	1
2. Arquitetura da solução	2
2.1. Servidor (<i>NMS_Server</i>)	2
2.2. Agente (<i>NMS_Agent</i>)	2
2.3. Comunicação Cliente-Servidor	3
3. Especificação dos protocolos propostos	4
3.1. <i>NetTask</i>	4
3.2. <i>AlertFlow</i>	4
3.3. Mecanismos de Fiabilidade	5
3.4. Fluxo do Sistema	5
3.5. Formato das Mensagens Protocolares	6
3.5.1. Tipos de Dados	6
3.5.2. Tipos das Mensagens	6
3.5.3. Identificação dos pacotes	7
3.5.4. Serialização e Desserialização dos pacotes	7
3.5.5. <i>Checksum</i>	8
3.5.6. Exemplo da estrutura completa de um pacote	8
4. Implementação	10
4.1. Detalhes	10
4.1.1. Modularização e Organização do código	10
4.1.2. Escalonamento	10
4.2. Parâmetros	10
4.3. Bibliotecas	11
5. Testes e Resultados	12
6. Conclusões e Trabalho Futuro	14
Lista de Siglas e Acrónimos	15

Lista de Figuras

Figura 1: Arquitetura geral do sistema.	2
Figura 2: Diagrama de sequência representativos da comunicação entre um agente e o servidor, através dos protocolos NetTask e AlertFlow.	6
Figura 3: Tipos de pacote.	7
Figura 4: Exemplo do pacote de métricas.	7
Figura 5: Exemplo do pacote Ack a serializar.	8
Figura 6: Exemplo do pacote Ack serializado.	8
Figura 7: Exemplo da estrutura completa do pacote de alerta.	9
Figura 8: Tipos de alerta.	9
Figura 9: Estrutura de um pacote Ack.	12
Figura 10: Implementação da serialização através da biblioteca Gob.	12
Figura 11: Resultado do uso do comando tcpdump — primeira iteração.	12
Figura 12: Resultado do uso do comando tcpdump — segunda iteração.	13

1. Introdução

Neste relatório, detalhamos o desenvolvimento do trabalho no âmbito da unidade curricular de [Comunicações por Computador](#). Foi proposto implementar um sistema de redes, denominado de *Network Monitoring System* (NMS), capaz de monitorizar, controlar e recolher informação, de forma distribuída, o estado dos *links* e dispositivos numa rede.

Tendo em conta a liberdade cedida pela equipa docente, a linguagem escolhida para o desenvolvimento do projeto foi o *Go* (*Golang*), devido às seguintes características:

- **Concorrência facilitada:** Oferece uma forma eficiente de lidar com múltiplas conexões, garantindo uma elevada escalabilidade do sistema.
- **Desempenho:** Conta com um elevado desempenho, permitindo latências baixas entre comunicações.
- **Controlo de erros:** Proporciona uma gestão prudente de erros, essencial num contexto onde falhas de redes são comuns.

2. Arquitetura da solução

Tal como requerido, utilizamos uma arquitetura baseada no modelo cliente-servidor, permitindo uma coordenação centralizada pelo servidor. Este comunica com vários agentes conectados numa dada rede e atribui-lhes tarefas para executarem. Através do uso de diferentes protocolos, são enviadas tarefas, recolhidas métricas e enviados alertas, sendo estes dados armazenados no servidor.

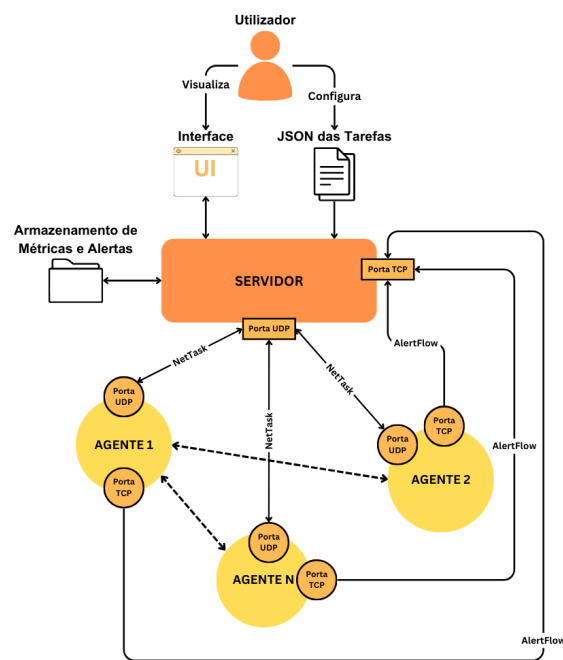


Figura 1: Arquitetura geral do sistema.

2.1. Servidor (*NMS_Server*)

O servidor atua como entidade central do sistema, sendo responsável por coordenar os agentes e verificar o seu estado. É responsável por:

- Gerir o registo dos agentes.
- Enviar tarefas configuradas em ficheiros JSON para os agentes.
- Receber e processar dados enviados periodicamente pelos agentes.
- Receber alertas em caso de violação de condições críticas definidas pelas tarefas.

2.2. Agente (*NMS_Agent*)

Cada agente atua num dispositivo da rede e pode realizar as seguintes funções:

- Registar-se no servidor.
- Receber tarefas configuradas pelo servidor.

- Recolher métricas locais relativas à execução das tarefas.
- Enviar relatórios periódicos ao servidor.
- Enviar alertas ao servidor, em caso de violação de condições definidas pelas tarefas.

2.3. Comunicação Cliente-Servidor

Na comunicação entre o servidor e os agentes, implementamos as funcionalidades especificadas através de dois protocolos: *NetTask* e *AlertFlow*.

Para gerir a comunicação de forma eficiente e independente, o servidor inicia, paralelamente, ambos os protocolos. Esta abordagem concorrente garante que a gestão do envio de tarefas, recolha de métricas e receção de alertas críticos possam ser processados de forma **simultânea**.

O agente inicializa apenas o protocolo *NetTask* para se registar e receber tarefas, e faz uso do *AlertFlow* só em caso de violação de condições para enviar alertas.

3. Especificação dos protocolos propostos

O desenvolvimento dos protocolos *NetTask* e *AlertFlow* está organizado de forma modular para atender às funcionalidades definidas no projeto. A separação das responsabilidades entre os dois protocolos foi feita tanto no servidor quanto nos agentes.

3.1. *NetTask*

O protocolo *NetTask* utiliza a camada de transporte UDP e conta também com algumas características normalmente associadas ao protocolo TCP. Este dinamiza o registo dos agentes no servidor, a distribuição de tarefas configuradas e a recolha periódica de métricas, tendo em atenção o envio e receção de *acknowledgements* e a retransmissão de pacotes perdidos.

Cada agente implementa a lógica necessária para se registar no servidor, receber tarefas configuradas e enviar relatórios periódicos com as métricas recolhidas na execução das mesmas. Para a gestão eficiente das conexões UDP, tanto no agente quanto no servidor, desenvolvemos funções responsáveis por configurar o envio e receção de mensagens, de forma a sustentar a comunicação fluida entre os dois lados do sistema.

No servidor, é aberta uma conexão de escuta na porta designada, que permite a receção de mensagens de registo enviadas pelos vários agentes. Posteriormente, o servidor utiliza essa mesma conexão para obter as mensagens das métricas recolhidas, subsequentes da execução das tarefas por parte dos agentes. No agente, a lógica é semelhante, mas este já procede à abertura de uma conexão de escuta unicamente dedicada à receção de mensagens do servidor, referentes às tarefas que lhe são atribuídas.

Por outro lado, tanto o servidor quanto os agentes estabelecem conexões para o envio de mensagens específicas. No servidor, é estabelecida uma conexão para cada um dos agentes, onde o único objetivo desta conexão resume-se ao envio das tarefas para cada agente respetivo. Já no agente, é estabelecida uma conexão para envio do seu registo inicial e outras para o envio dos relatórios periódicos de métricas.

3.2. *AlertFlow*

De forma a assegurar que o envio de alertas por parte dos agentes é sempre recebido pelo servidor e tratado com a maior das prioridades, desenvolveu-se o protocolo *AlertFlow*, que utiliza a camada de transporte TCP.

Para que o servidor seja capaz de receber e processar os alertas de maneira atómica e eficiente, a implementação do *AlertFlow* foi concretizada de modo a que a sua gestão seja independente do protocolo *NetTask*, garantindo a alta disponibilidade e paralelismo do protocolo.

Desta forma, os agentes estabelecem uma comunicação unidirecional e confiável com o servidor para transmitir os alertas no momento em que são detetadas condições críticas ou *timeouts*, caso não consigam enviar as métricas na frequência estabelecida.

3.3. Mecanismos de Fiabilidade

Para assegurar uma comunicação fiável no protocolo *NetTask*, foram implementados mecanismos que superam as limitações do UDP, como a ausência de garantia de entrega. Estes mecanismos incluem:

- **Confirmações (ACKs):** Cada pacote enviado deve ser confirmado pelo destinatário. Caso a confirmação não seja recebida dentro de um intervalo de tempo definido, o pacote é retransmitido até ao limite máximo de tentativas permitido.
- **Gestão de Retransmissões e Mapeamento de Estado:** O estado dos pacotes pendentes de confirmação é monitorizado continuamente através de uma *goroutine* dedicada. Um mapa partilhado rastreia os pacotes, permitindo reenviar automaticamente aqueles que não foram confirmados dentro do tempo limite. Este processo assegura fiabilidade, evitando sobrecarga na comunicação e garantindo consistência mesmo com múltiplas *threads*.
- **Controlo de Pacotes Duplicados:** Cada pacote possui um identificador único, o que permite ao sistema descartar duplicados e evitar processamentos redundantes.
- **Validação de Pacotes:** Ao receber um pacote, o destinatário verifica a sua integridade e, caso seja válido, envia um ACK. Pacotes inválidos resultam num NOACK.

Estes mecanismos garantem que apenas mensagens legítimas são processadas, proporcionando uma comunicação robusta e eficiente, mesmo em redes instáveis.

3.4. Fluxo do Sistema

A comunicação entre o servidor e os agentes segue uma lógica estruturada para assegurar a execução eficiente das tarefas e o envio de alertas críticos. O diagrama de sequência apresentado na Figura 2 ilustra as principais interações, destacando os seguintes passos:

1. **Inicialização do Servidor:** Configura dois processos concorrentes:
 - Um processo TCP dedicado ao tratamento de alertas críticos.
 - Um processo UDP para o registo de agentes e distribuição de tarefas a partir de ficheiros JSON.
2. **Registo de Agentes:** Cada agente estabelece uma conexão UDP e envia os seus dados de identificação ao servidor, que valida e regista os agentes disponíveis.
3. **Distribuição de Tarefas:** O servidor atribui tarefas específicas aos agentes, utilizando pacotes estruturados enviados pelo protocolo *NetTask*.
4. **Execução e Monitorização:** Os agentes processam as tarefas recebidas, executam comandos (*ping* e/ou *iperf*) e monitorizam métricas definidas, como *jitter*, *packet loss*, CPU e RAM.
5. **Envio de Métricas e Alertas:** Métricas periódicas são enviadas via UDP, enquanto alertas críticos são transmitidos de forma confiável via TCP através do AlertFlow.

O diagrama de sequência fornece uma visão clara da coordenação entre os diferentes processos, ilustrando como os pacotes de dados fluem entre os agentes e o servidor para assegurar a monitorização contínua e confiável da rede.

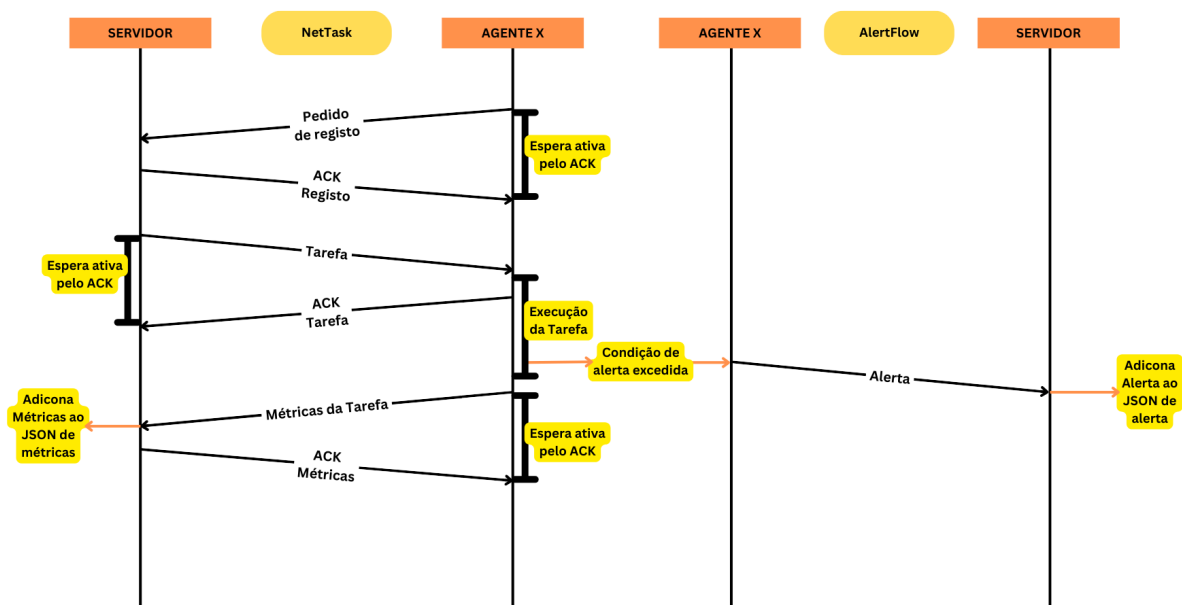


Figura 2: Diagrama de sequência representativos da comunicação entre um agente e o servidor, através dos protocolos NetTask e AlertFlow.

3.5. Formato das Mensagens Protocolares

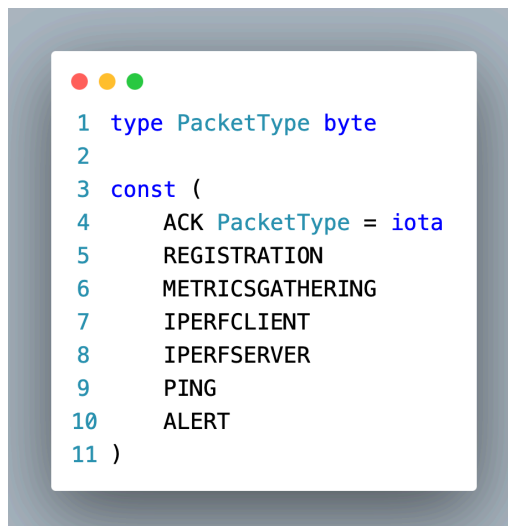
A estrutura das mensagens do sistema foi projetada para proporcionar uma comunicação leve e rápida entre o servidor e os agentes. Cada mensagem segue um formato padronizado que facilita a transmissão, receção, e interpretação dos dados em diversas funcionalidades do sistema.

3.5.1. Tipos de Dados

O uso de tipos de dados **compactos**, como *byte*, *uint16* e *enums*, foi essencial para otimizar o desempenho e reduzir o consumo de recursos, especialmente no contexto do projeto, onde grandes volumes de dados precisam de ser processados **rapidamente**.

3.5.2. Tipos das Mensagens

Em cada mensagem incluímos o parâmetro *PacketType* que especifica o seu tipo, diferenciando o propósito do pacote. Esta abordagem modular facilita a expansão do sistema, além de simplificar a distinção de pacotes e execução das diferentes funcionalidades.



```

1 type PacketType byte
2
3 const (
4     ACK PacketType = iota
5     REGISTRATION
6     METRICSGATHERING
7     IPERFCLIENT
8     IPERFSERVER
9     PING
10    ALERT
11 )

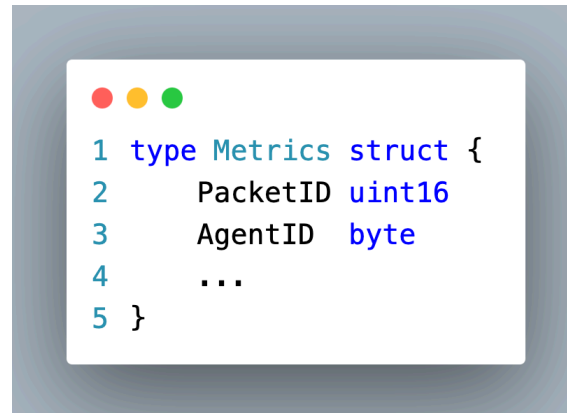
```

Figura 3: Tipos de pacote.

3.5.3. Identificação dos pacotes

Para possibilitar o rastreamento e controlo adequado do fluxo de dados, a grande maioria das mensagens incluem dois campos essenciais:

- **PacketID**: Identifica de forma **única** o pacote em relação ao dispositivo que o enviou ou recebeu. Funciona como um número de sequência local ao dispositivo, garantindo que cada pacote enviado pelo mesmo possa ser identificado e, caso necessário, confirmado.
- **AgentID**: Representa o ID único de cada agente dentro do sistema. Corresponde ao ID de registo do agente, assegurando que todas as mensagens trocadas estejam associadas ao dispositivo correto.



```

1 type Metrics struct {
2     PacketID uint16
3     AgentID  byte
4     ...
5 }

```

Figura 4: Exemplo do pacote de métricas.

Estes campos, quando combinados, formam um identificador exclusivo dos pacotes de um agente, viabilizando a rastreabilidade de pacotes entre múltiplos agentes e o servidor.

3.5.4. Serialização e Desserialização dos pacotes

Para garantir que as mensagens sejam transmitidas de forma compacta e uniforme, cada mensagem é serializada antes do envio. Esse processo transforma os dados num formato de *bytes* eficiente para transmissão. Do lado da receção, os dados são desserializados, garantindo que os pacotes sejam reconstruídos exatamente como foram enviados. Este processo é importante pois minimiza o risco de erros de interpretação, além de diminuir drasticamente o tamanho dos pacotes transmitidos.

Suplementarmente, a serialização e desserialização estão **isentas** do uso de bibliotecas de codificação automática (ex. [encoding/gob](https://golang.org/pkg/encoding/gob/)). Desta forma, o envio e receção de pacotes manipulam apenas os *bytes* referentes aos parâmetros da estrutura do pacote em questão, sem portar os **bytes extra** associados à

codificação do pacote através da biblioteca. Tal fundamentação pode ser vista mais detalhadamente no [capítulo 5](#). Para além de uma comunicação mais leve, esta implementação assegura que, por exemplo, o recetor dos *bytes* do pacote consegue decodificá-los através de qualquer linguagem de programação, não requerendo que o mesmo utilize a biblioteca usada para a codificação.

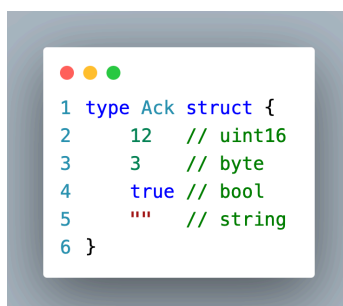


Figura 5: Exemplo do pacote Ack a serializar.

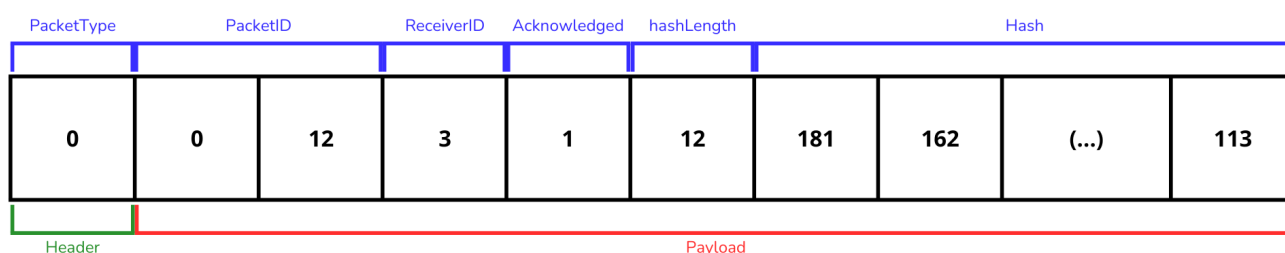


Figura 6: Exemplo do pacote Ack serializado.

Assegurando que os dados sejam interpretados corretamente em diferentes plataformas, optamos por usar um formato único *Big Endian* para a codificação e decodificação dos *bytes*. Desta forma, os erros de compatibilidade são minimizados e a preservação da consistência dos dados durante a comunicação é assegurada. É de notar que o formato *Little Endian* seria igualmente válido, desde que fosse usado consistentemente por todas as partes envolvidas na comunicação.

3.5.5. Checksum

Devido às possíveis instabilidades das ligações na rede a ser testada, foi incorporado um campo de *hash* em todas as mensagens transmitidas pelo protocolo *NetTask*. Após a parametrização de um pacote, o mesmo é serializado, e a lista de *bytes* resultante é usada como *input* pelo algoritmo **SHA-256**. A chave consequente colocada no pacote a enviar é truncada a 12 *bytes*, para minimizar o tamanho ocupado pelo pacote.

Na receção de pacotes, o *hash* é recalculado e comparado com o valor recebido. Como a *hash* permite a distinção de **2⁹⁶ pacotes**, qualquer incongruência no pacote é detetada e a mensagem é rejeitada, protegendo o sistema contra possíveis corrupções ou modificações durante o transporte.

3.5.6. Exemplo da estrutura completa de um pacote

Segue, como exemplo, a estrutura do pacote de alerta enviado pelo *AlertFlow* contendo métricas recolhidas por um agente após a execução de tarefas:

```
1 type Alert struct {
2     PacketID uint16
3     AgentID   byte
4     TaskID    uint16
5     AlertType AlertType
6     Exceeded  float32
7     Time      string
8 }
```

Figura 7: Exemplo da estrutura completa do pacote de alerta.

```
1 type AlertType byte
2
3 const (
4     CPU AlertType = iota
5     RAM
6     JITTER
7     PACKETLOSS
8     TIMEOUT
9     INTERFACESTATS
10    ERROR
11 )
```

Figura 8: Tipos de alerta.

- **PacketID**: Identifica o pacote único do respectivo dispositivo;
- **AgentID**: Indica o agente responsável pelo envio do pacote;
- **TaskID**: Identifica a tarefa a partir da qual o alerta é originado;
- **AlertType**: Identifica o tipo de alerta;
- **Exceeded**: Quantifica o valor que despertou o alerta;
- **Time**: Regista o momento exato em que o alerta foi despertado.

4. Implementação

4.1. Detalhes

4.1.1. Modularização e Organização do código

Com o objetivo de facilitar o reaproveitamento de funcionalidades e diminuir algumas **redundâncias** no código desenvolvido, optamos por generalizar as funcionalidades fundamentais do nosso sistema, tendo por base as principais necessidades tanto do servidor como dos agentes. Desta forma, é possível reduzir a reescrita de código em futuras atualizações e a necessidade de modificação do modelo atual torna-se praticamente inexistente.

Adicionalmente, tendo como base uma **forte modularização** e organização de código, a localização e correção de possíveis erros torna-se, eventualmente, mais simples e dinâmica, apontando para um maior sucesso no desenvolvimento de futuras funcionalidades.

Todo este processo pôde ser traduzido pela implementação de **funções genéricas**, capazes de abstrair ao máximo as diferentes funcionalidades, visto que muitas das funções do servidor são espelhadas nos diferentes agentes (ou vice-versa), desde o envio e recepção de pacotes, codificação e decodificação de mensagens até ao estabelecimento de conexões.

4.1.2. Escalonamento

De forma a lidar eficientemente com os diversos processos do sistema, projetamos um escalonamento que aproveita as capacidades de **paralelismo** da linguagem *Go*. Tanto no servidor quanto no agente, utilizamos *goroutines* e métodos de sincronização para executar concorrentemente diversas funcionalidades.

Na comunicação entre o servidor e os agentes, é possível visualizar o uso dessa concorrência ao gerir múltiplas conexões simultâneas, que permite que o sistema lide com os diversos pedidos e conexões de forma paralela.

No entanto, nem todos os processos do sistema exigem execução concorrente, como, por exemplo, o processo de registo dos agentes no servidor, no qual adotamos um modelo sequencial (FIFO). Esse tipo de escalonamento sequencial é crucial para assegurar que as mensagens de registo sejam tratadas de forma ordenada, evitando conflitos na atualização do sistema com métodos de exclusão mútua (*mutex*).

Para garantir que as tarefas sejam executadas de forma eficiente, utilizamos uma execução concorrente de tarefas, permitindo que múltiplas operações de teste da rede ocorram simultaneamente. Contudo, certos comandos a executar concorrentemente dentro do mesmo dispositivo podem interferir com outras tarefas, como o caso da execução dos comandos *iperf* e *ping* em simultâneo. Sendo assim, o sistema de escalonamento do agente é responsável por controlar a execução dos comandos e priorizar as tarefas de *iperf*, não permitindo a execução de outras tarefas enquanto um *iperf* está ativo.

4.2. Parâmetros

Durante a implementação definimos uma série de parâmetros no código que desempenham um papel crucial para configurar o comportamento do sistema e garantir a sua funcionalidade. Estes valores foram

ajustados para otimizar a performance e a confiabilidade do sistema, ao mesmo tempo que simplificam ajustes futuros. Abaixo, detalhamos algumas constantes utilizadas:

- **TIMEOUT**: Este parâmetro define o intervalo de tempo (em milissegundos) que o sistema aguardará antes de considerar uma mensagem perdida e retransmiti-la.
- **BUFFERSIZE**: Define o tamanho máximo do *buffer* utilizado para leitura e escrita de dados, tanto para *NetTask* quanto para *AlertFlow*.
- **SERVERID**: Representa o identificador único atribuído ao servidor no sistema.
- **HASHSIZE**: Determina o tamanho da *hash* utilizada para diminuir o tamanho total dos pacotes a serem enviados via *NetTask*.
- **MAXRETRANSMISSIONS**: Define o número máximo de tentativas de retransmissão para mensagens não confirmadas via *NetTask*.

Estes parâmetros são cuidadosamente escolhidos, podendo ser ajustados para otimizar o desempenho em diferentes ambientes de rede.

4.3. Bibliotecas

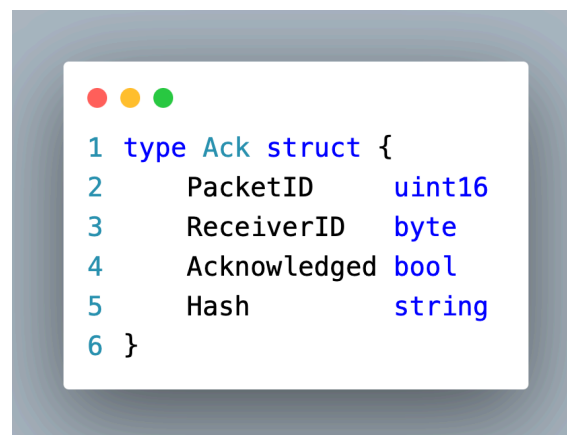
Durante a realização do projeto, recorremos a várias bibliotecas padrão da linguagem utilizada (*Go*) para atender aos requisitos do sistema. Abaixo, destacamos as mais importantes e seus respectivos objetivos:

- **net** - A biblioteca padrão *net* fornece as ferramentas necessárias para implementação de sockets TCP e UDP, fundamentais para os protocolos *NetTask* e *AlertFlow*.
- **sync** - A biblioteca *sync* é utilizada para gerir a concorrência e garantir a integridade dos dados compartilhados entre *goroutines* (*threads*), através de, por exemplo, *mutex* e seus *locks*.

Estas e outras bibliotecas importadas desempenham papéis fundamentais na implementação, oferecendo-nos funcionalidades de alto nível que facilitaram o desenvolvimento deste projeto.

5. Testes e Resultados

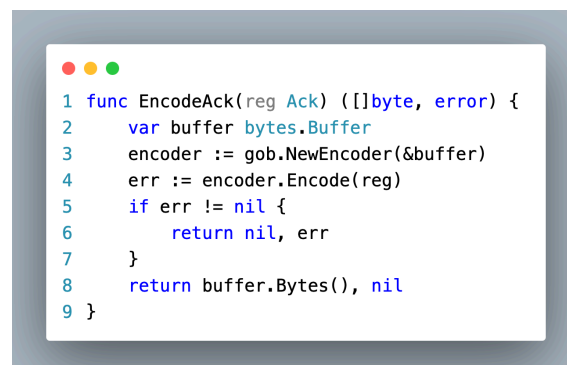
O uso de uma serialização e desserialização manual dos parâmetros das estruturas dos pacotes revelou-se ser um ponto fulcral para manter uma comunicação **leve e eficiente** entre os dispositivos do sistema. Olhemos para a estrutura de um pacote *Ack*:



```
1 type Ack struct {
2     PacketID    uint16
3     ReceiverID   byte
4     Acknowledged bool
5     Hash         string
6 }
```

Figura 9: Estrutura de um pacote Ack.

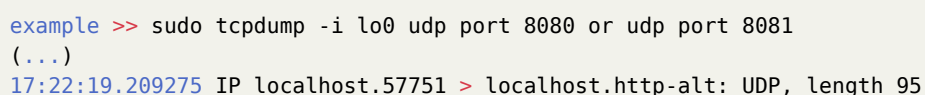
Numa fase inicial, o grupo fez uso da biblioteca [encoding/gob](https://golang.org/pkg/encoding/gob/) para a codificação e decodificação dos nossos pacotes. A sua implementação tomou a seguinte forma:



```
1 func EncodeAck(reg Ack) ([]byte, error) {
2     var buffer bytes.Buffer
3     encoder := gob.NewEncoder(&buffer)
4     err := encoder.Encode(reg)
5     if err != nil {
6         return nil, err
7     }
8     return buffer.Bytes(), nil
9 }
```

Figura 10: Implementação da serialização através da biblioteca Gob.

Concebeu-se um simples programa que envia um pacote *Ack* codificado usando a implementação referida acima. Deste modo, observando o resultado do uso do comando *tcpdump* para rastrear a comunicação entre processos, é possível destacar que a serialização dos pacotes resulta num *array* de *bytes* que, neste caso, ocupa **95 bytes** de espaço.



```
example >> sudo tcpdump -i lo0 udp port 8080 or udp port 8081
(...)
17:22:19.209275 IP localhost.57751 > localhost.http-alt: UDP, length 95
```

Figura 11: Resultado do uso do comando *tcpdump* — primeira iteração.

Analogamente, desenvolveu-se um programa que envia um pacote *Ack* codificado com a corrente implementação. Observando novamente o resultado do comando *tcpdump*, vemos que a serialização atual resulta agora num *array* de apenas **18 bytes** de tamanho.

```
example >> sudo tcpdump -i lo0 udp port 8080 or udp port 8081  
(...)  
17:21:24.511111 IP localhost.62037 > localhost.sunproxyadmin: UDP, length 18
```

Figura 12: Resultado do uso do comando *tcpdump* — segunda iteração.

Conclui-se que esta decisão de design trouxe um impacto positivo para o programa, reduzindo significativamente o uso de espaço pelas diversas conexões, neste caso, em **81%**.

6. Conclusões e Trabalho Futuro

Este projeto permitiu o desenvolvimento de um sistema distribuído de monitorização de redes, demonstrando a viabilidade de protocolos personalizados, como o NetTask e o AlertFlow, para comunicação eficiente e confiável. A modularidade e o uso da linguagem *Go* foram fundamentais para alcançar escalabilidade e desempenho, enquanto a serialização manual mostrou-se eficaz na redução do consumo de largura de banda.

No futuro, o sistema pode ser aprimorado com a inclusão de mecanismos de segurança, ferramentas de análise avançada de dados e testes mais extensivos em redes reais. Essas melhorias visam aumentar a aplicabilidade e robustez da solução em cenários complexos e reais.

Lista de Siglas e Acrónimos

TCP Transmission Control Protocol

UDP User Datagram Protocol

FIFO First In, First Out

SHA Secure Hash Algorithm

CPU Central Processing Unit

RAM Random Access Memory

JSON JavaScript Object Notation

ACK Acknowledgment