



Universidade do Minho  
Escola de Engenharia  
Licenciatura em Engenharia Informática

## Trabalho Prático 2

Ano Letivo de 2025/2026

## Comunicações por Computador - PL504

**João Delgado**

A106836

**Nelson Mendes**

A106884

**Simão Mendes**

A106928

December 07, 2025

CC

# Índice

<b>1.</b>	<b>Preparação e Ambiente .....</b>	<b>1</b>
<b>2.</b>	<b>Design dos Protocolos .....</b>	<b>2</b>
2.1.	<i>MissionLink</i> (ML) .....	2
2.1.1.	Arquitetura Global e Filosofia de <i>Design</i> .....	2
2.1.1.1.	<i>MissionLinkReceiver</i> (Nave-Mãe) .....	3
2.1.1.2.	<i>MissionLinkSender</i> ( <i>Rover</i> ) .....	3
2.1.2.	Estrutura e Semântica das Mensagens .....	3
2.1.2.1.	Estrutura Comum (Cabeçalho - 17 bytes) .....	3
2.1.2.2.	Estrutura <i>MissionRequest</i> .....	4
2.1.2.3.	Estrutura <i>MissionAssignment</i> .....	4
2.1.2.4.	Estrutura <i>MissionProgress</i> .....	5
2.1.2.5.	Estrutura <i>MissionAck</i> .....	5
2.1.3.	Mecanismos de Fiabilidade e Controlo .....	6
2.2.	<i>TelemetryStream</i> (TS) .....	6
2.2.1.	Arquitetura e Transporte .....	6
2.2.2.	Formato da Mensagem .....	6
2.2.3.	Controlo e Tolerância a Falhas .....	7
2.3.	API de Observação .....	7
<b>3.</b>	<b>Implementação do <i>MissionLink</i> (ML) .....</b>	<b>9</b>
3.1.	Transmissor ( <i>Rover</i> ) .....	9
3.2.	Recetor (Nave-Mãe) .....	9
3.3.	Serialização do ML .....	10
<b>4.</b>	<b>Implementação do <i>TelemetryStream</i> (TS) .....</b>	<b>11</b>
4.1.	Transmissor ( <i>Rover</i> ) .....	11
4.2.	Recetor (Nave-Mãe) .....	11
4.3.	Serialização de Alta Performance .....	12
<b>5.</b>	<b>Implementação da API de Observação e <i>Ground Control</i> .....</b>	<b>13</b>
5.1.	API de Observação .....	13
5.2.	API do <i>Ground Control</i> .....	13
<b>6.</b>	<b>Integração e Testes .....</b>	<b>16</b>
6.1.	Topologia e Cenários de Rede .....	16
6.2.	Testes Realizados .....	16
6.2.1.	Teste de Concorrência com Múltiplos <i>Rovers</i> .....	16
6.2.2.	Fiabilidade do <i>MissionLink</i> (UDP) .....	18
6.2.3.	Desempenho sobre Latência e Perda de Pacotes .....	19
<b>7.</b>	<b>Extras .....</b>	<b>21</b>
7.1.	Telemetria Estendida e Simulação Ambiental Realista .....	21
7.2.	Interface <i>Ground Control</i> Interativa ( <i>Dashboard</i> ) .....	21
7.3.	Tipologia de Missões Expandida .....	22

<b>8.</b>	<b>Revisão Final</b>	<b>23</b>
8.1.	Arquitetura e Componentes	23
8.2.	Design dos Protocolos	23
8.2.1.	<i>MissionLink</i> (ML)	23
8.2.2.	<i>TelemetryStream</i> (TS)	24
8.2.3.	API e <i>Ground Control</i>	24
8.3.	Resultados dos Testes	24
8.3.1.	Teste de Concorrência	24
8.3.2.	Fiabilidade do <i>MissionLink</i>	25
8.3.3.	Desempenho sob Condições Adversas	25
8.4.	Conclusão	25
<b>9.</b>	<b>Bibliografia</b>	<b>26</b>
<b>10.</b>	<b>Anexos</b>	<b>27</b>

# **Lista de Figuras**

Figura 1	Topologia no CORE (ligações e outros aspectos omitidos para visualização). . . . .	1
Figura 2	Dados de pacote enviado por protocolo <i>TelemetryStream</i> . . . . .	7
Figura 3	Diagrama de sequência do endpoint GET <code>/api/telemetry/latest</code> , ilustrando o acesso seguro aos dados partilhados através de <code>sync.RWMutex</code> . . . . .	8
Figura 4	Processo de receção de dados de <i>MissionLink</i> . . . . .	10
Figura 5	Processo de receção de dados de <i>TelemetryStream</i> . . . . .	12
Figura 6	Interface Web do <i>Ground Control</i> . . . . .	14
Figura 7	Interface operacional unificada do <i>Ground Control</i> . . . . .	15
Figura 8	<i>Ground Control</i> conectado à nave-mãe de Venús, com acesso a 8 rovers operacionais. . . . .	17
Figura 9	<i>Ground Control</i> conectado à nave-mãe de Marte, com acesso a 8 rovers operacionais. . . . .	18
Figura 10	Captura de tráfego Wireshark que evidencia a retransmissão de pacotes UDP após <i>timeout</i> e a subsequente recuperação da comunicação. . . . .	19

## **Lista de Anexos**

1	Anexo 1: Diagrama de Sequência do processo de recuperação em caso de erro ( <i>Telemetry Stream</i> ). ....	27
2	Anexo 2: Diagrama de Sequência do funcionamento de <i>Telemetry Stream</i> . ....	28
3	Anexo 3: Diagrama de Sequência do processo de recuperação em caso de erro ( <i>Mission Link</i> ). ....	29
4	Anexo 4: Diagrama de Sequência do funcionamento de <i>Mission Link</i> . ....	30

# 1. Preparação e Ambiente

O presente trabalho prático foi desenvolvido no âmbito da Unidade Curricular de **Comunicações por Computador**, com o objetivo principal de **simular e implementar protocolos de comunicação num cenário de exploração espacial**. O sistema proposto simula uma missão composta por duas Naves-Mãe em órbita de planetas distintos, múltiplos *Rovers* na superfície planetária de cada um deles e uma rede de satélites intermédia, introduzindo desafios típicos de redes com restrições, como latência e perda de pacotes.

Para a concretização do projeto, foi selecionada a linguagem de programação **Go**, devido à sua eficiência, forte suporte a concorrência (através de *goroutines* e *channels*) e facilidade na criação de aplicações de rede robustas. O ambiente de simulação e testes decorreu no emulador **CORE** (*Common Open Research Emulator*), permitindo a virtualização da topologia de rede e a validação do comportamento dos protocolos em condições controladas.

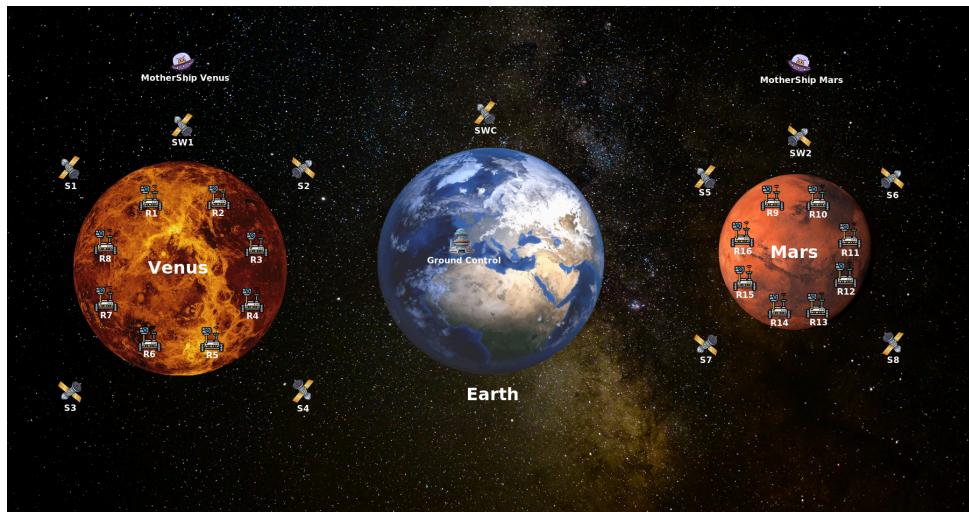


Figura 1: Topologia no CORE (ligações e outros aspectos omitidos para visualização).

A arquitetura do projeto foi estruturada de forma modular, separando claramente os componentes lógicos:

- **Mothership (Nave-Mãe)**: O servidor central de coordenação;
- **Rovers**: As unidades clientes que executam missões e enviam telemetria;
- **Ground Control**: A interface de observação e comando;
- **Protocolos**: Implementações isoladas do **MissionLink** (sobre UDP) e **TelemetryStream** (sobre TCP).

Esta organização permitiu um desenvolvimento iterativo e facilitou a integração dos diversos componentes nas fases posteriores do projeto.

## 2. *Design* dos Protocolos

### 2.1. *MissionLink* (ML)

O ***MissionLink*** constitui o núcleo do protocolo aplicacional desenvolvido para suportar a comunicação crítica no âmbito da missão espacial simulada entre a Nave-Mãe (*MotherShip*) e os múltiplos *Rovers* de superfície. Implementado intencionalmente sobre o **protocolo de transporte UDP**, que por natureza não oferece garantias de entrega, ordenação ou integridade, o *MissionLink* incorpora, a nível aplicacional, um conjunto abrangente de **mecanismos de fiabilidade** que compensam estas limitações.

#### 2.1.1. Arquitetura Global e Filosofia de *Design*

A arquitetura do ML organiza-se em dois componentes principais que refletem os papéis assimétricos na comunicação:

- o ***MissionLinkReceiver***, que opera exclusivamente na Nave-Mãe;
- o ***MissionLinkSender***, que opera em cada *Rover* individualmente.

Esta separação, claramente demarcada, permite uma especialização otimizada das responsabilidades de cada entidade. A **Nave-Mãe, atuando como servidor centralizado**, precisa de gerir concorrência, manter o estado de múltiplos clientes e coordenar a atribuição de missões. Os ***Rovers, atuando como clientes***, focam-se na comunicação ponto-a-ponto com o servidor, na execução das missões recebidas e no reporte periódico do seu estado.

A filosofia de *design* subjacente assenta em três pilares fundamentais: **simplicidade operacional, robustez perante falhas e extensibilidade futura**. O protocolo foi concebido para ser suficientemente simples para ser implementado e depurado dentro do contexto académico, mas simultaneamente robusto o suficiente para lidar com cenários de falha de rede plausíveis. A **estrutura das mensagens e os mecanismos de controlo** foram desenhados de modo a **permitir a introdução futura de novos tipos de mensagem ou campos** adicionais sem ruturas na compatibilidade, graças ao uso de um campo de versão no cabeçalho e a uma estratégia de serialização flexível.

### 2.1.1.1. MissionLinkReceiver (Nave-Mãe)

O *MissionLinkReceiver* opera como **servidor UDP central**, gerindo comunicações de múltiplos *Rovers* simultaneamente. Utiliza uma goroutine principal (*receiveLoop*) para escutar datagramas na porta configurada, **registando automaticamente os endereços dos Rovers** à primeira comunicação. Cada mensagem recebida é processada numa goroutine dedicada (*handleMessage*) que **desserializa o conteúdo e direciona-o para os canais apropriados**. Para mensagens críticas (REQUEST\_MISSION e MISSION\_UPDATE), **gera automaticamente ACKs de confirmação**. O envio de atribuições de missão (MISSION\_ASSIGNMENT) é feito com **retransmissão garantida** através do mecanismo *sendWithRetry*, que **tenta até 3 vezes e aguarda confirmação**. Tudo isto mantendo dois mapas sincronizados: um de endereços de cliente e outro que associa *RoverIDs* a endereços físicos, permitindo comunicação direcionada.

### 2.1.1.2. MissionLinkSender (Rover)

Cada *Rover* instancia um *MissionLinkSender* que **estabelece uma ligação UDP orientada para a Nave-Mãe**. O componente central é a goroutine *ackHandler*, que processa mensagens recebidas: **trata ACKs removendo sequências pendentes e processa atribuições de missão** invocando um *callback* registado. **Para envio de mensagens críticas** (pedidos e atualizações), emprega *sendWithRetry*, que **regista o número de sequência como pendente** e repete a transmissão até receber confirmação ou esgotar 3 tentativas. Gerencia internamente a numeração sequencial das mensagens e mantém um mapa de *ACKs* pendentes sincronizado. Também oferece um *shutdown* gracioso via canal de paragem.

## 2.1.2. Estrutura e Semântica das Mensagens

O protocolo *MissionLink* define **quatro tipos de estruturas de dados que representam as mensagens trocadas** entre a Nave-Mãe e os *Rovers*. Cada estrutura implementa um formato binário específico que inclui sempre um **cabeçalho comum seguido de campos específicos** conforme o tipo de mensagem.

### 2.1.2.1. Estrutura Comum (Cabeçalho - 17 bytes)

Todas as mensagens partilham os seguintes campos iniciais:

- **Type (MessageType - 4 bytes)**: Identificador numérico do tipo de mensagem;
- **SeqNum (uint32 - 4 bytes)**: Número de sequência único para controlo de fiabilidade;
- **Version (uint8 - 1 byte)**: Versão do protocolo para compatibilidade futura;
- **Timestamp (int64 - 8 bytes)**: *Timestamp Unix* que marca o momento de criação da mensagem.

### 2.1.2.2. Estrutura MissionRequest

Esta estrutura é utilizada quando um *Rover* solicita uma nova missão. Para além dos campos do cabeçalho, inclui:

- **RoverId (uint8 - 1 byte)**: Identificador único do *Rover*;
- **RoverName (string - tamanho variável)**: Nome descritivo do *Rover*;
- **Tamanho mínimo**: 18 bytes + tamanho do *RoverName*.

### 2.1.2.3. Estrutura MissionAssignment

Esta estrutura é a mais complexa e é enviada pela Nave-Mãe para atribuir uma missão. Inclui:

#### Campos de tamanho fixo:

- **RoverId (uint8 - 1 byte)**: Identificador do *Rover* alvo;
- **RoverName (string - tamanho variável)**: Nome do *Rover*;
- **MissionId (uint8 - 1 byte)**: Identificador único da missão;
- **MissionConfig (MissionTypeConfig - tamanho variável)**: Configuração do tipo de missão;
- **Tarefa (string - tamanho variável)**: Descrição da tarefa;
- **Prioridade (uint8 - 1 byte)**: Prioridade da missão (1 – 5);
- **ToleranciaFalhas (uint8 - 1 byte)**: Percentagem de tolerância (0 – 100);
- **DataInicial (int64 - 8 bytes)**: *Timestamp* de início;
- **DuracaoMaxima (uint32 - 4 bytes)**: Duração máxima, em segundos;
- **Atualizacoes (uint32 - 4 bytes)**: Número de atualizações esperadas;
- **AreaGeografica ([] Coordinate - tamanho variável)**: *Array* de coordenadas da área a explorar (cada *Coordinate* = 10 bytes);
- **PontosInteresse ([] Coordinate - tamanho variável)**: *Array* de coordenadas dos pontos de interesse a visitar (cada *Coordinate* = 10 bytes);
- **Tamanho mínimo**: 44 bytes + campos variáveis.

#### 2.1.2.4. Estrutura *MissionProgress*

Utilizada pelo *Rover* para reportar progresso da missão. Inclui:

**Campos de tamanho fixo (além do cabeçalho):**

- ***RoverId* (`uint8 - 1 byte`)**: Identificador do *Rover*;
- ***MissionId* (`uint8 - 1 byte`)**: Identificador da missão;
- ***Progress* (`float32 - 4 bytes`)**: Percentagem de conclusão;
- ***Status* (`MissionStatus - 4 bytes`)**: Estado atual da missão.

**Campos opcionais (podem ser nulos):**

- ***CurrentPosition* (`Coordinate - 10 bytes`)**: indica a posição atual do *Rover*;
- ***NrImagesCaptured* (`uint32 - 4 bytes`)**: Número de imagens capturadas;
- ***NrSamplesCollected* (`uint8 - 1 byte`)**: Número de amostras recolhidas;
- ***Diagnostics* (`string - tamanho variável`)**: Informação de diagnóstico.
- **Tamanho mínimo:** 22 bytes + campos opcionais

#### 2.1.2.5. Estrutura *MissionAck*

Utilizada para confirmar receção de mensagens. Inclui:

- ***AckForSeq* (`uint32 - 4 bytes`)**: Número de sequência da mensagem confirmada;
- ***RoverId* (`uint8 - 1 byte`)**: Identificador do *Rover*;
- ***Status* (`AckStatus - 4 bytes`)**: Estado da confirmação.
- **Tamanho fixo:** 26 bytes

Esta estrutura de tamanhos permite otimizar o uso da largura de banda, enquanto mantém flexibilidade para diferentes tipos de conteúdo nas mensagens.

Embora a representação visual através de diagramas seja frequentemente útil para ilustrar estes tipos de estruturas de dados, optámos por não incluir diagramas detalhados de cada tipo de mensagem no presente relatório. Esta decisão fundamenta-se na **natureza variável dos campos que compõem as mensagens**, particularmente no caso das estruturas *MissionAssignment* e *MissionProgress*. Em vez disso, focámo-nos numa **descrição textual precisa**, complementada com **cálculos de tamanho mínimo**, que transmitem de forma mais útil e realista as características do protocolo para fins de análise de desempenho e dimensionamento de rede.

### 2.1.3. Mecanismos de Fiabilidade e Controlo

A fiabilidade baseia-se em **confirmações positivas e retransmissão por *timeout***, implementando um **esquema *stop-and-wait* simplificado**. Cada mensagem crítica **aguarda um ACK** específico antes de considerar a entrega concluída. O *timeout* padrão é de 2 segundos com máximo de 3 retransmissões, **funcionando como *back-off* básico** contra congestionamento. A **numeração sequencial** permite detetar duplicados e correlacionar respostas. O *design*, deliberadamente leve, evita *handshakes* complexos, **mantendo a agilidade típica do UDP**, garantindo a entrega através de lógica aplicacional.

Um exemplo do mecanismo de recuperação em caso de erros no envio de *MissionLink* pode ser consultado em [Anexo 3](#).

## 2.2. TelemetryStream (TS)

O **TelemetryStream** constitui o protocolo fundamental para a monitorização contínua e em tempo real da frota de *Rovers*. O seu *design* assenta sobre a **camada de transporte TCP**, uma escolha arquitetural que assegura a entrega fiável, ordenada e íntegra de dados críticos. Esta abordagem garante que a Nave-Mãe mantém uma **representação exata dos atributos do Rover**, tais como: “saúde”, localização e estado operacional, sem o risco de perda de informação vital por instabilidade da rede.

### 2.2.1. Arquitetura e Transporte

O protocolo segue o modelo Cliente-Servidor com gestão de estado persistente, otimizado para múltiplos clientes simultâneos:

- **Cliente (Rover):** Atua como o **iniciador ativo** da conexão. Cada *Rover* possui um transmissor dedicado que estabelece e mantém o túnel TCP, gerindo o ciclo de envio periódico de acordo com o intervalo definido *TelemetryInterval*;
- **Servidor (Nave-Mãe):** Opera em **modo de escuta contínua** na porta designada (*TelemetryServerPort*). A arquitetura do servidor utiliza um modelo de concorrência baseado em goroutines: para cada nova conexão aceite, o sistema lança uma *thread* para processamento isolado. O acesso à lista de clientes ativos é protegido por **exclusão mútua** (*Mutex*), o que garante a segurança das operações em memória (*thread-safety*) mesmo com múltiplos *Rovers* a conectar-se ou desconectar-se em simultâneo, evitando assim *race conditions*.

### 2.2.2. Formato da Mensagem

Com o objetivo de maximizar a **eficiência de débito** e assegurar a previsibilidade da **largura de banda**, o protocolo estabelece uma estrutura de mensagem binária rígida, com um tamanho fixo de **64 bytes**. A organização interna do pacote divide-se nos seguintes blocos lógicos:

- **Identificação (13 bytes):** Inicia a sequência e contém o *RoverId*, o Nome e o *Status* operacional do emissor, o que permite a validação imediata da origem;
- **Navegação (10 bytes):** Segue-se imediatamente e agrega as coordenadas espaciais (X, Y e Z), elementos essenciais no mapeamento topológico na Nave-Mãe;
- **Sensores (18 bytes):** Compacta as leituras dos sensores ambientais (como Temperatura, Pressão e Radiação) e métricas internas críticas, tal como o nível de Bateria;
- **Metadados (23 bytes):** Inclui o *Timestamp* para ordenação temporal, os vetores de movimento (*Velocity*, *Direction*) e o indicador global de integridade do sistema (*SystemHealth*).

Segue-se a representação visual desta mensagem:

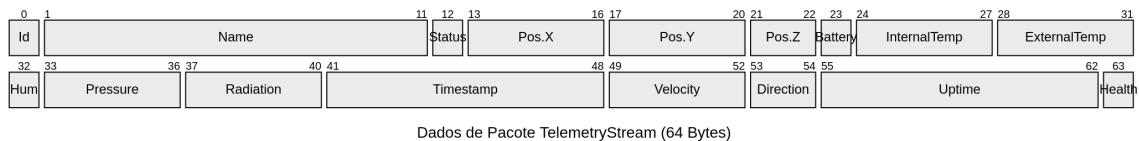


Figura 2: Dados de pacote enviado por protocolo *TelemetryStream*.

### 2.2.3. Controlo e Tolerância a Falhas

O protocolo implementa mecanismos de resiliência para lidar com a instabilidade da rede simulada no *CORE*:

1. **Sessão Persistente:** A conexão mantém-se ativa durante toda a operação, evitando o *overhead* de múltiplos *handshakes*;
2. **Validação de Integridade:** O receptor descarta pacotes que não correspondam exatamente ao tamanho esperado (*network.TelemetryPacketSize*), de modo a prevenir o processamento de dados fragmentados;
3. **Reconexão Automática:** Em caso de interrupção da rede (erro de I/O), o transmissor inicia automaticamente um ciclo de recuperação, tentando restabelecer a conexão TCP em intervalos fixos (*ReconnectDelay*) até ao limite de tentativas configurado (*MaxReconnectTries*).

Um exemplo do mecanismo de recuperação em caso de erros no envio de *TelemetryStream* pode ser consultado em [Anexo 1](#).

## 2.3. API de Observação

A **API de Observação** foi desenhada como o ponto central de acesso aos dados da missão, permitindo que entidades externas, especificamente o nó ***Ground Control***, consultem o estado do sistema em tempo real.

Optou-se por uma arquitetura **REST** (*Representational State Transfer*) sobre HTTP, utilizando JSON como formato de intercâmbio de dados. Esta escolha justifica-se pela ubiquidade do protocolo HTTP, facilidade de *debugging* e compatibilidade direta com **interfaces Web modernas**.

A API foi projetada para expor os seguintes **endpoints principais**:

- GET `/api/telemetry/latest`: **Retorna o estado mais recente** de todos os *Rovers* ativos. Inclui dados sensíveis como bateria, posição (X, Y, Z), saúde do sistema e dados ambientais;
- GET `/api/missions`: **Disponibiliza o registo completo de missões**, incluindo o seu estado atual (Em Progresso, Concluída ou Falhada), tipo de tarefa e progresso percentual;
- POST `/api/missions/create`: **Permite a submissão de novas missões** para os *Rovers*. Este endpoint aceita um *payload* JSON com os parâmetros da missão e coloca-a na fila de processamento da Nave-Mãe;
- GET `/`: **Endpoint de diagnóstico** (*Health Check*) que fornece estatísticas gerais do servidor e o tempo de simulação sincronizado.

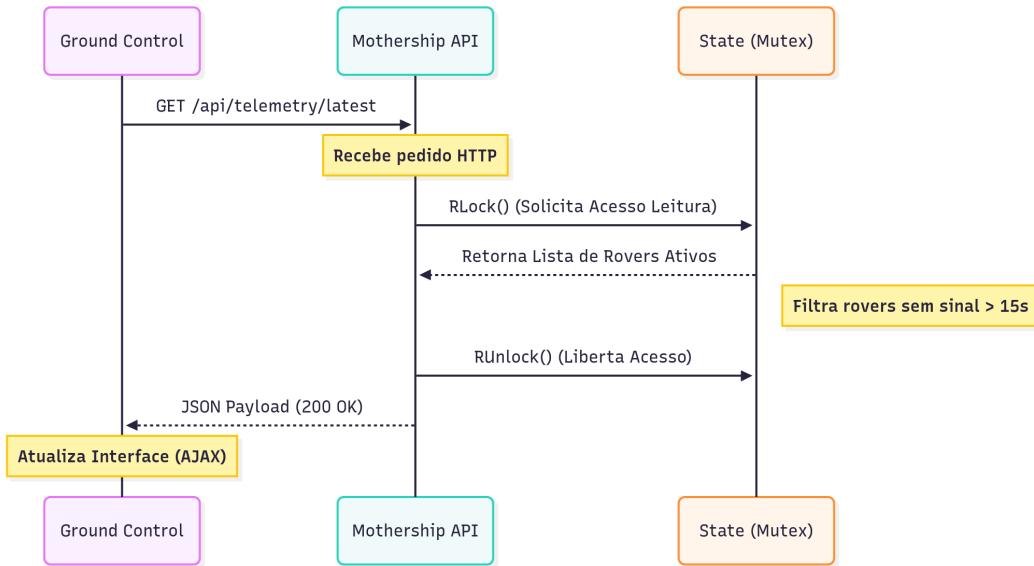


Figura 3: Diagrama de sequência do endpoint `GET /api/telemetry/latest`, ilustrando o acesso seguro aos dados partilhados através de `sync.RWMutex`.

Para garantir a **integridade dos dados** num ambiente concorrente (onde múltiplos *Rovers* escrevem dados via UDP/TCP e o *Ground Control* lê dados via HTTP), o desenho da API incorpora um modelo de gestão de estado *thread-safe*, utilizando primitivas de sincronização (`sync.RWMutex`) para mediar o acesso às estruturas de dados em memória.

### 3. Implementação do *MissionLink* (ML)

A implementação do *MissionLink* materializa-se através de três componentes fundamentais que transformam o *design* conceitual num sistema funcional. O serializador/deserializador **converte estruturas de dados em binário para transmissão**, implementando formatos otimizados com campos de tamanho fixo e *arrays* variáveis. O ***MissionLinkReceiver*, executado na Nave-Mãe**, atua como servidor UDP concorrente que regista dinamicamente *Rovers*, processa mensagens em paralelo e garante entregas críticas através de retransmissões com confirmação. O ***MissionLinkSender*, operando em cada *Rover***, estabelecendo ligações orientadas, gerindo números de sequência locais e implementando mecanismos simétricos de *retry*. Juntos, estes componentes concretizam um **protocolo que mantém a agilidade do UDP**, enquanto **assegura fiabilidade** através de lógica aplicacional robusta, *timeout* controlados e gestão de estado distribuída.

#### 3.1. Transmissor (*Rover*)

O *MissionLinkSender* é o componente executado em cada *Rover* que gere toda a comunicação de saída para a Nave-Mãe. Funciona como um **cliente UDP orientado** que estabelece uma ligação direta ao servidor da Nave-Mãe. A sua principal responsabilidade é **garantir que as mensagens críticas**, tais como pedidos de missão e atualizações de progresso, **sejam entregues de forma fiável, apesar de usar UDP** como transporte subjacente.

Para tal, **implementa um mecanismo de retransmissão inteligente**: quando envia uma mensagem importante, **regista o seu número de sequência num mapa de ACKs** pendentes e entra num ciclo de até 3 tentativas, aguardando 2 segundos entre cada tentativa pela confirmação correspondente. Paralelamente, uma goroutine dedicada (*ackHandler*) fica à escuta de respostas da Nave-Mãe, processando tanto confirmações, como atribuições de missão. Esta arquitetura permite que o ***Rover* mantenha comunicação bidirecional fiável**, enquanto minimiza o bloqueio de operações.

#### 3.2. Recetor (Nave-Mãe)

O *MissionLinkReceiver* opera na Nave-Mãe **como um servidor UDP concorrente** que aceita ligações de múltiplos *Rovers* simultaneamente. A sua função central é **desmultiplexar mensagens recebidas**, mantendo o estado dos *Rovers* ativos e garantindo a entrega fiável das atribuições de missão.

Quando um datagrama chega, o *receiveLoop* principal **delega o seu processamento para uma goroutine** dedicada (*handleMessage*). Esta identifica o tipo de mensagem, desserializa o conteúdo e direciona-o para o canal apropriado. **Para mensagens de entrada** que requerem confirmação, **gera automaticamente um ACK**. Para mensagens

de saída críticas (como atribuições de missão), usa um **mecanismo de sendWithRetry** idêntico ao do *Sender*, **assegurando entrega** mesmo em condições de rede adversas.

Um aspecto crucial do *Receiver* é o mapeamento dinâmico de endereços: ao receber a primeira mensagem de um *Rover*, **associa o seu RoverID ao endereço UDP físico**, permitindo comunicação direcionada futura sem necessidade de descoberta contínua.

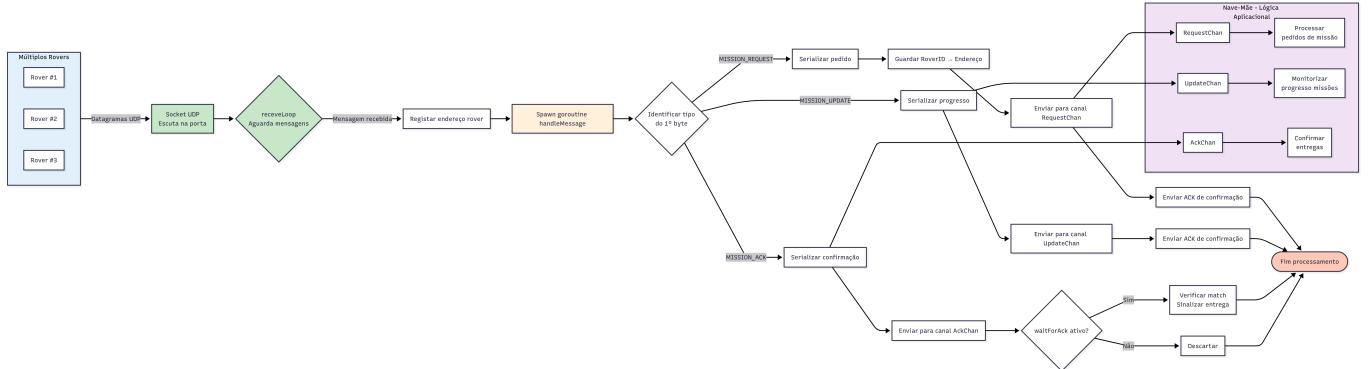


Figura 4: Processo de receção de dados de *MissionLink*.

### 3.3. Serialização do ML

O módulo *MissionLinkData* fornece as **funções de serialização e desserialização** que transformam as estruturas de dados em memória em formatos binários transmissíveis pela rede, e vice-versa. Esta camada é **fundamental para a interoperabilidade entre Nave-Mãe e Rovers**, garantindo que ambos os lados interpretam os dados de forma consistente.

A serialização segue um formato binário híbrido: **campos fixos** para dados comuns e previsíveis, combinados com mecanismos flexíveis para **conteúdo variável**. Por exemplo, **strings têm tamanho máximo pré-definido** (20 bytes para nomes, 50 para descrições), enquanto **arrays de coordenadas são precedidos por um byte** indicando o número de elementos. Para mensagens com campos opcionais (como *MissionProgress*), usa-se um *byte* de **flags** que indica **quais campos estão presentes**, otimizando assim o tamanho dos pacotes.

As funções implementam **conversões byte-a-byte em little endian**, garantindo compatibilidade entre diferentes arquiteturas, e seguindo um **padrão consistente de leitura/escrita sequencial** que simplifica tanto a implementação como a depuração.

Para uma representação visual detalhada do fluxo de comunicação e dos estados de interação de *MissionLink* consulte o diagrama de sequência disponível em [Anexo 4](#).

## 4. Implementação do *TelemetryStream* (TS)

A implementação do subsistema de telemetria divide-se em três componentes lógicos distintos: **o transmissor** (localizado no *Rover*), **o recetor** (integrado na Nave-Mãe) e **o módulo de serialização partilhado**. A arquitetura tira partido das primitivas de concorrência da linguagem Go (*goroutines* e *channels*) para assegurar um débito elevado e baixa latência.

### 4.1. Transmissor (*Rover*)

O componente ***TelemetrySender*** assume a responsabilidade de recolher e enviar os dados. A sua implementação baseia-se num ciclo de envio periódico controlado por um temporizador de precisão (`time.Ticker`), o qual dispara eventos de transmissão de acordo com o intervalo definido em `network.TelemetryInterval`.

A lógica de envio opera da seguinte forma:

- **Recolha:** O método invoca `GetTelemetryData()` da interface do *Rover* para obter o estado instantâneo dos sensores e sistemas.
- **Serialização:** Os dados sofrem conversão imediata para um *slice* de *bytes* através do pacote de serialização.
- **Transmissão:** A função `conn.Write()` envia o *buffer* resultante para o socket TCP.

Para mitigar bloqueios na *thread* principal do *Rover*, o transmissor corre numa *goroutine* dedicada (`go ts.sendLoop()`). Esta separação assegura que eventuais atrasos na rede (latência TCP) não afetam a simulação física do movimento ou a lógica de decisão do veículo. Adicionalmente, o transmissor monitoriza um canal de paragem (`stopChan`) para garantir o encerramento gracioso das conexões e rotinas quando o *Rover* é desligado.

### 4.2. Recetor (Nave-Mãe)

O ***TelemetryReceiver*** implementa um servidor **TCP concorrente**, capaz de escalar para múltiplos *rovers* em simultâneo. A sua estrutura interna utiliza um padrão de conexão onde múltiplas conexões de entrada convergem para um único canal de processamento de dados, conforme ilustrado na Figura 5.

- **Gestão de Conexões:** O método `acceptLoop` escuta a porta do servidor e, para cada nova conexão aceite (`listener.Accept()`), lança imediatamente uma *goroutine* de atendimento (`handleClient`).
- **Sincronização:** O acesso ao mapa de clientes ativos (`clients`) encontra-se protegido por um ***mutual exclusion lock*** (`clientsMux`). Esta proteção previne

*race conditions* quando múltiplos *rovers* tentam conectar-se ou desconectar-se no mesmo instante.

- **Validação de Integridade:** Antes do processamento, cada leitura sofre **validação quanto ao tamanho esperado** (`network.TelemetryPacketSize`). O recetor deteta e descarta pacotes com tamanho incorreto, resultantes de fragmentação ou erros de rede, para evitar a corrupção do estado da aplicação.
- **Desacoplamento e Load Shedding:** A estrutura utiliza um canal com *buffer* de capacidade fixa (`DataChan`, tamanho 100 mensagens) para encaminhar as mensagens. A implementação recorre a uma **escrita não-bloqueante** (`select` com `default`): caso o canal esteja cheio (i.e., a Nave-Mãe não consiga processar à velocidade de chegada), o recetor opta por descartar o pacote mais recente em vez de bloquear a conexão TCP. Esta estratégia de *load shedding* impede que um atraso no processamento cause um efeito de cascata que bloquee toda a rede de telemetria.

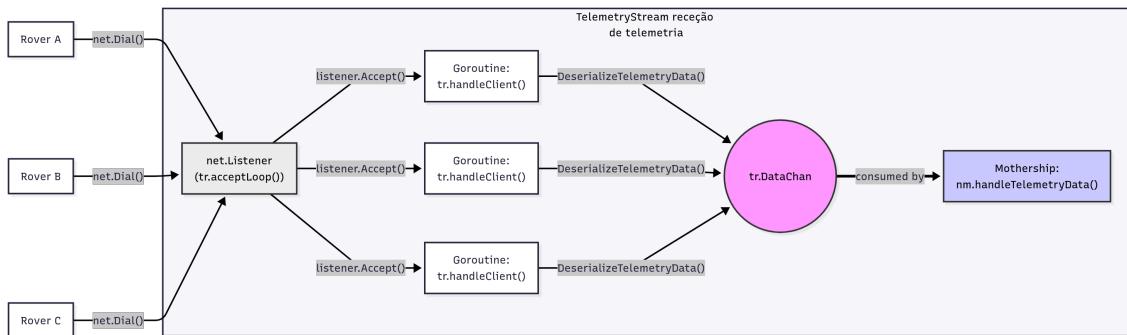


Figura 5: Processo de receção de dados de *TelemetryStream*

### 4.3. Serialização de Alta Performance

Ao abdicar de formatos *autodescritivos* como *JSON* ou *XML*, a implementação recorre à manipulação direta de memória no pacote *serialization*.

As funções `SerializeTelemetryData` e `DeserializeTelemetryData` realizam a **conversão diretamente byte-a-byte** em *little endian*. Esta técnica oferece duas vantagens cruciais na implementação:

- **Determinismo:** O tamanho do *buffer* de saída é **sempre conhecido a priori** (`network.TelemetryPacketSize`), o que elimina a necessidade de alocação dinâmica de memória repetitiva.
- **Eficiência de CPU:** A ausência de *parsing* de texto ou uso de *reflection* reduz drasticamente o tempo de CPU necessário para preparar cada pacote, um fator essencial dado a frequência elevada de atualizações de telemetria.

Para uma representação visual detalhada do fluxo de comunicação e dos estados de interação de *TelemetryStream* consulte o diagrama de sequência disponível em [Anexo 2](#).

## 5. Implementação da API de Observação e *Ground Control*

Esta secção detalha a **implementação técnica da camada de observação**, destacando as soluções adotadas para garantir performance, usabilidade e fiabilidade.

### 5.1. API de Observação

A **implementação da API na Nave-Mãe** reside no pacote `internal/api` e atua como uma interface para o estado partilhado da aplicação (*MothershipState*).

Um aspeto crucial da implementação foi a **gestão de concorrência**. Dado que a Nave-Mãe recebe pacotes UDP e TCP assincronamente enquanto serve pedidos HTTP, utilizou-se um `sync.RWMutex` para **proteger os mapas de telemetria e missões**. Isto permite leituras paralelas ilimitadas (ideal para múltiplos clientes de observação) enquanto garante exclusividade nas escritas.

Adicionalmente, implementou-se **lógica de negócio inteligente** diretamente na API:

- **Filtragem de “Stale Data”**: O método `getLatestTelemetry` filtra automaticamente *rovers* que não tenham comunicado nos últimos 15 segundos. Isto garante que o *Ground Control* apenas visualiza unidades efetivamente ativas, evitando a apresentação de dados obsoletos ou enganadores.
- **Configuração Dinâmica de Missões**: No *handler* de criação de missões, a API atribui automaticamente a duração máxima da missão com base no seu tipo (ex: **Mapeamento Global** recebe **3600s**, enquanto **tarefas simples** recebem **600s**), centralizando as regras de negócio no servidor.

### 5.2. API do *Ground Control*

O **componente *Ground Control*** foi implementado não como uma simples ferramenta de linha de comandos (CLI), mas sim como um **Dashboard Web Interativo**, excedendo os requisitos funcionais básicos para proporcionar uma experiência de operação superior.

Desenvolvido em **Go** (servidor) com **HTML/CSS/JavaScript embutidos** (cliente), o *Ground Control* apresenta as seguintes características distintivas:

- **Interface Visual Moderna:** Utilização de **CSS personalizado** (inspirado em estilos *Dark Mode*) e **iconografia** (*FontAwesome*) para uma leitura rápida e intuitiva do estado da frota. Indicadores críticos como bateria e saúde do sistema utilizam **códigos de cor** semânticos (verde/laranja/vermelho) para alertar visualmente o operador.
- **Atualização em Tempo Real (AJAX):** A interface implementa um ciclo de **polling via JavaScript** (*fetchData*) que atualiza o DOM dinamicamente sem necessidade de recarregar a página. Isto cria uma **experiência fluida**, semelhante a uma *Single Page Application* (SPA).

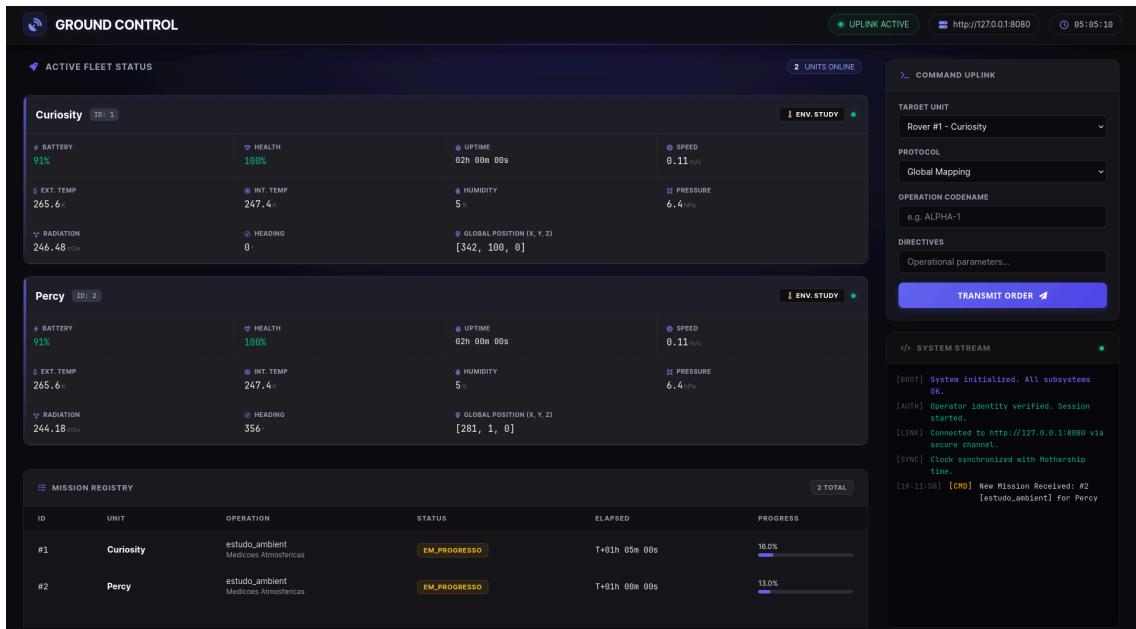


Figura 6: Interface Web do *Ground Control*.

- **Terminal de Sistema Integrado:** Foi incluída uma janela de “Terminal” na interface *web* que apresenta um **log em tempo real** dos eventos do sistema (conexões, falhas, conclusões de missão), aumentando a imersão e facilitando o diagnóstico de problemas.
- **Robustez e Persistência:** Como funcionalidade extra, implementou-se um **mechanismo de persistência** no lado do cliente utilizando *localStorage*. Isto permite que o *Ground Control* recorde missões que foram interrompidas por falha de comunicação com o *rover*, mantendo o estado de alerta **INTERROMPIDA**.
- **Cálculo de Uptime e Sincronização:** O cliente **sincroniza o seu relógio** com o tempo de simulação do servidor (*server\_time*), garantindo que todos os tempos de missão apresentados são **consistentes com a “realidade”** da Nave-Mãe, independentemente do desfasamento temporal do computador local.
- **Comando e Controlo Ativo:** O sistema transcende a simples visualização passiva ao integrar um módulo de *Command Uplink* funcional. Através de um

**formulário dedicado**, o operador pode configurar e submeter novas missões para a Nave-Mãe via pedidos HTTP *POST*. A resposta da operação é processada e exibida instantaneamente num Terminal de Sistema simulado na interface, fornecendo **feedback visual imediato** (“*Uplink Confirmed*” ou “*Transmission Error*”) e fechando o ciclo de comunicação bidirecional sem necessidade de recarregar a página.

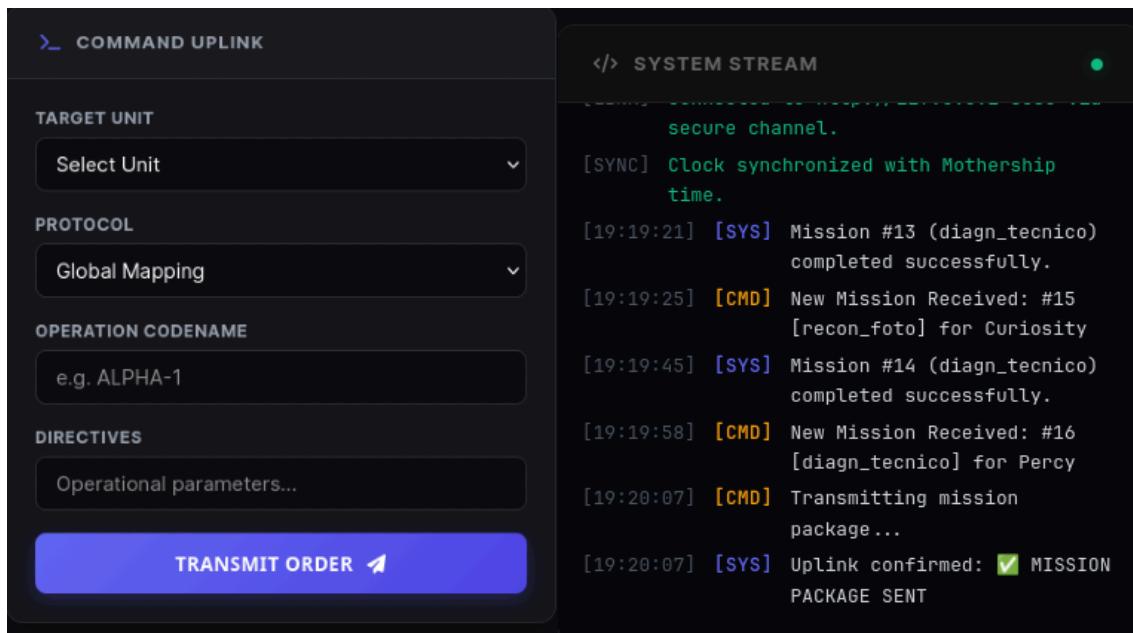


Figura 7: Interface operacional unificada do *Ground Control*.

## 6. Integração e Testes

Esta etapa do projeto consistiu na **unificação dos módulos desenvolvidos** e na sua **validação num ambiente de rede controlado**. Para tal, recorreu-se ao emulador **CORE** (*Common Open Research Emulator*), o qual permitiu a construção da topologia de rede espacial e a execução do código real em cada nó.

### 6.1. Topologia e Cenários de Rede

A infraestrutura de teste foi expandida para além dos requisitos mínimos do enunciado, implementando um **cenário multi-planetário** de alta complexidade. A topologia simula simultaneamente **operações em Marte e Vénus**, totalizando duas Naves-Mãe ativas. Cada ambiente planetário possui uma infraestrutura dedicada composta por uma constelação de 4 satélites para encaminhamento (*routing*) e suporta uma frota de 8 *rovers*. A topologia pode ser consultada em Figura 1.

Para validar a robustez dos protocolos neste ambiente complexo, definiram-se dois perfis de configuração distintos no *CORE*:

1. **Cenário Ideal (Baseline)**: Configuração com largura de banda elevada e taxas de perda/latência nulas em ambos os planetas. Este cenário serviu para validar a lógica funcional, o estabelecimento de sessões e a correção dos formatos de dados em larga escala.
2. **Cenário Adverso (Stress Test)**: Introdução artificial de latência (simulando a distância orbital e interferências atmosféricas) e perda de pacotes (*packet loss*) nos links entre satélites. Este perfil teve como objetivo forçar a ativação dos mecanismos de fiabilidade e reconexão implementados, bem como testar o comportamento do sistema sob carga máxima com 16 *rovers* a transmitir simultaneamente.

### 6.2. Testes Realizados

Para validar o cumprimento dos requisitos funcionais e a robustez da solução, a equipa executou um conjunto de testes focado nas três dimensões críticas definidas no enunciado. A validação dos resultados recorreu a diferentes métodos de prova, desde a análise de tráfego até à verificação visual na interface de controlo.

#### 6.2.1. Teste de Concorrência com Múltiplos Rovers

Este teste responde ao requisito de assegurar que a Nave-Mãe processa missões e telemetria em simultâneo, sem bloqueios. Tirou-se partido da topologia multi-planetária para levar este cenário ao limite.

## Procedimento:

- **Cenário Ideal:** Inicializaram-se simultaneamente os 16 *rovers* da topologia (8 em Marte, 8 em Vénus) com a rede em condições perfeitas.
- **Carga Máxima:** Todos os *rovers* iniciaram o envio de telemetria, enquanto se solicitavam missões aleatórias a vários *rovers* distintos em cada planeta.

## Resultados Observados:

- Ambas as Naves-Mãe processaram os **fluxos de telemetria** e os **pedidos de missão concorrentes** sem atrasos perceptíveis.
- O painel do *Ground Control* refletiu corretamente o **estado Online** de todos os veículos, sem intermitências ou dados misturados.
- **Verificou-se o isolamento estrito de contextos:** a Nave-Mãe de Marte geriu apenas os seus veículos, sem interferência dos *rovers* de Vénus, validando a arquitetura de canais concorrentes implementada no servidor.

De seguida pode-se observar as duas interfaces correspondentes à conexão do *ground-control* com a nave-mãe de Vénus e Marte respetivamente:

The screenshot displays the 'GROUND CONTROL' application interface. At the top, it shows 'ACTIVE FLEET STATUS' with 8 units online. Below this, two rovers are listed: 'Vanguard' (ID: 1) and 'Pioneer' (ID: 2). Each rover section provides detailed status information:

- Vanguard:** Battery 81%, Health 100%, Uptime 24h 00m 00s, Speed 0.27 m/s. External Temp: 497.1 K, Internal Temp: 495.4 K, Humidity: 0%, Pressure: 9200000.0 hPa. Radiation: 289.94 RSV, Heading: 359°. Global Position: [34113, 24323, 9].
- Pioneer:** Battery 86%, Health 100%, Uptime 23h 00m 00s, Speed 0.27 m/s. External Temp: 496.6 K, Internal Temp: 494.9 K, Humidity: 0%, Pressure: 9175580.0 hPa. Radiation: 309.07 RSV, Heading: 15. Global Position: [54071, 46550, 5].

To the right of the rovers, there are sections for 'COMMAND UPLINK' (with fields for Target Unit, Protocol, Operation Codename, and Directives), and a 'TRANSMIT ORDER' button. Below these is a 'SYSTEM STREAM' section showing log entries:

- [23:05:38] [SYS] Mission #28 (map\_global) completed successfully.
- [23:05:42] [CHD] New Mission Received: #35 [diagn\_tecnico] for Pioneer
- [23:05:46] [SYS] Mission #28 (map\_global) completed successfully.

Figura 8: *Ground Control* conectado à nave-mãe de Vénus, com acesso a 8 *rovers* operacionais.

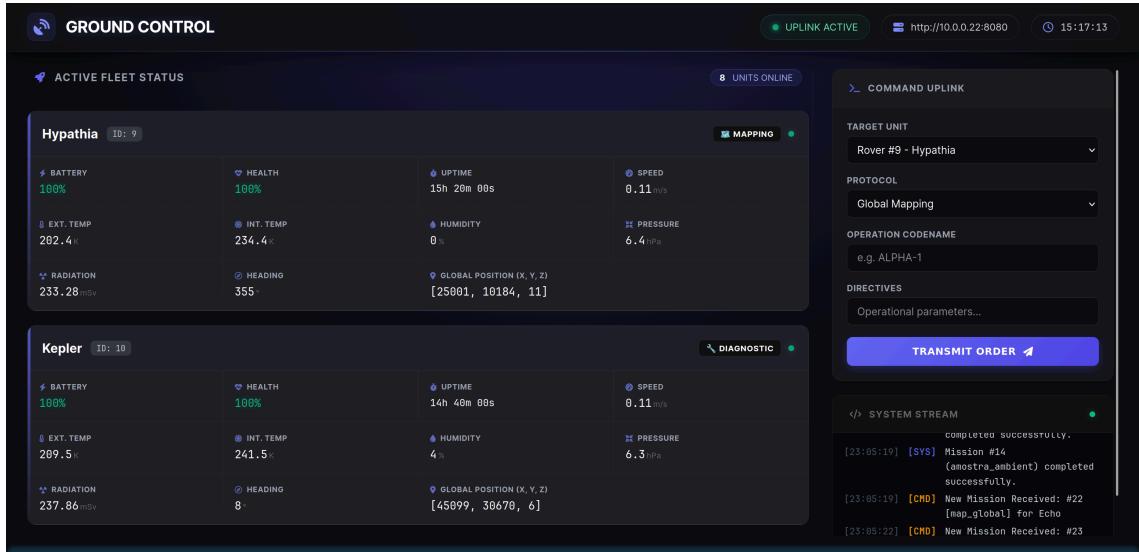


Figura 9: *Ground Control* conectado à nave-mãe de Marte, com acesso a 8 rovers operacionais.

### 6.2.2. Fiabilidade do *MissionLink* (UDP)

Este teste focou-se na validação dos mecanismos de fiabilidade implementados a nível aplicacional para compensar a natureza “*best-effort*” do protocolo UDP. Dado tratar-se de uma validação de protocolo, utilizou-se a captura de pacotes para prova.

#### Procedimento:

- **Cenário de Perda:** Configurou-se uma taxa de perda de pacotes de 30% (*Packet Loss*) nos enlaces de satélite no CORE.
- **Ação:** Um Rover solicitou uma missão à Nave-Mãe, num momento em que a rede apresentava instabilidade.

#### Resultados Observados:

- A análise da captura de tráfego (Figura 10) evidencia uma sequência de tentativas de comunicação iniciadas pelo *Rover* (origem 10.0.8.20) destinadas à Nave-Mãe (destino 10.0.1.20) que não obtiveram resposta imediata.
- Observam-se envios sucessivos nos instantes 93.09s, 95.11s e 97.12s. O intervalo rigoroso de 2.0 segundos entre cada pacote comprova a atuação precisa do temporizador de retransmissão (*timeout*) definido na lógica da aplicação.
- A recuperação da comunicação concretiza-se finalmente ao segundo 99.14, onde o pedido do *Rover* chega com sucesso.
- A Nave-Mãe responde imediatamente com os dados da missão (pacote de 186 bytes), e o protocolo conclui-se com o envio da confirmação (*Ack*) pelo *Rover*, fechando o ciclo da transação.

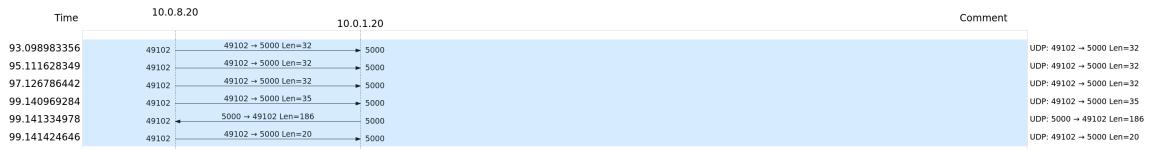


Figura 10: Captura de tráfego *Wireshark* que evidencia a retransmissão de pacotes UDP após *timeout* e a subsequente recuperação da comunicação.

### 6.2.3. Desempenho sobre Latência e Perda de Pacotes

Este teste avaliou o comportamento global do sistema sob condições de rede degradadas, simulando as distâncias orbitais.

#### Procedimento:

- Configuração:** Introduziu-se uma latência artificial elevada (RTT > 2000ms) e uma perda aleatória de pacotes em todos os canais de comunicação.
- Monitorização:** Compararam-se os *timestamps* de envio nos *logs* do *Rover* com os *timestamps* de receção na Nave-Mãe durante alguns minutos.

#### Resultados Observados:

- A sessão TCP **manteve a estabilidade** apesar do elevado RTT, sem desconexões inesperadas.
- Apesar da perda de pacotes na rede, a aplicação na Nave-Mãe recebeu a **sequência de telemetria completa e ordenada**.
- A comparação de registos evidencia o atraso consistente imposto pela latência física, mas confirma que o sistema **não acumula atraso de processamento adicional**, mantendo o débito em tempo real.

Apresentam-se de seguida os registos de execução de ambos os intervenientes, os quais evidenciam o desfasamento temporal provocado pela latência:

#### Terminal Rover:

```

1 23:46:28.105: Sending telemetry packet (seq 501) sh
2 23:46:29.105: Sending telemetry packet (seq 502)
3 23:46:29.450: Timeout waiting for ACK (attempt 1), retrying...
4 23:46:30.105: Sending telemetry packet (seq 503)
5 23:46:31.105: Sending telemetry packet (seq 504)
6 23:46:31.455: Timeout waiting for ACK (attempt 2), retrying...
7 23:46:32.105: Sending telemetry packet (seq 505)
8 23:46:33.105: Sending telemetry packet (seq 506)
9 23:46:33.460: Timeout waiting for ACK (attempt 3), retrying...

```

**Terminal Nave-Mãe:**

- |   |               |                                       |    |
|---|---------------|---------------------------------------|----|
| 1 | 23:46:29.118: | Received telemetry from rover 2       | sh |
| 2 | 23:46:30.120: | Received telemetry from rover 2       |    |
| 3 | 23:46:31.125: | Received telemetry from rover 2       |    |
| 4 | 23:46:32.122: | Received telemetry from rover 2       |    |
| 5 | 23:46:33.128: | Received telemetry from rover 2       |    |
| 6 | 23:46:34.130: | Received telemetry from rover 2       |    |
| 7 | 23:46:35.455: | Received mission request from rover 2 |    |
| 8 | 23:46:35.458: | Assignment delivered to rover 2       |    |

## 7. Extras

### 7.1. Telemetria Estendida e Simulação Ambiental Realista

Enquanto o enunciado exigia apenas a identificação, posição e estado operacional do *rover*, a implementação final expandiu significativamente o vetor de telemetria para incluir dados ambientais e sistémicos, simulados com base na física planetária.

**Atributos Adicionais de Telemetria:** Foram adicionados campos na estrutura de dados binária para monitorizar:

- **Estado do Hardware:** Temperatura interna, Saúde do Sistema (0 – 100%) e Nível de Bateria.
- **Dados Atmosféricos:** Pressão atmosférica, Humidade relativa, Temperatura externa e Níveis de Radiação.
- **Cinemática:** Vetor de Velocidade e Direção.
- **Física Multi-Planeta:** O sistema suporta perfis ambientais distintos para Marte e Vénus. Os valores gerados pelos sensores dos *rovers* não são aleatórios puros, mas sim limitados pelas constantes físicas definidas no código (ex: gravidade, intervalos de temperatura e pressão atmosférica base de cada planeta).

### 7.2. Interface *Ground Control* Interativa (*Dashboard*)

O enunciado solicitava uma interface de visualização (leitura). O nosso grupo desenvolveu uma ***Dashboard Web* Interativa** completa que permite não só a monitorização, mas também o controlo ativo da missão.

**Submissão Manual de Missões:** Foi implementado um formulário na interface *web* (/mission/submit) que permite ao operador criar e enviar novas missões para a Nave-Mãe em tempo real via API. A Nave-Mãe processa este pedido e despacha-o para o *rover* via *MissionLink*, fechando o ciclo de comando.

#### Visualização em Tempo Real:

A interface apresenta:

- **Listagem dinâmica de *rovers*** com indicadores visuais de saúde (cores baseadas no estado da bateria/sistema).
- **Log de estado das missões** com barras de progresso e *badges* de estado (Planeada, Em Progresso, Concluída).

- **Arquitetura Offline-First:** Ao contrário de interfaces web comuns que dependem de *CDNs*, todos os *assets* (ícones *FontAwesome*, fontes *Inter/JetBrains Mono*) são servidos localmente pelo binário *Go*. Isto permite que a simulação gráfica funcione em ambientes isolados (como o emulador CORE) sem acesso à Internet.

### 7.3. Tipologia de Missões Expandida

De forma a superar o requisito de implementar uma “tarefa” genérica , o grupo desenvolveu um sistema polimórfico com cinco tipos de missões distintos, cada um com lógica de execução e critérios de sucesso próprios:

- **Mapeamento Global:** Implementa algoritmos de navegação autónoma (Espiral ou Aleatória) para cobertura de área.
- **Estudo Ambiental:** Foca na monitorização contínua de sensores atmosféricos e de radiação.
- **Reconhecimento Fotográfico:** Simula a gestão de armazenamento e contagem de capturas de multimédia.
- **Diagnóstico Técnico:** Executa rotinas de validação interna dos subsistemas do *rover*.
- **Recolha de Amostras:** Introduz um elemento probabilístico, onde o sucesso depende da geologia do planeta configurado (Marte ou Vénus).

Esta diversidade valida a robustez do protocolo *MissionLink* ao processar diferentes estruturas de dados e regras de negócio sem alterações na camada de transporte.

## 8. Revisão Final

O presente trabalho implementou um sistema completo de comunicação para simulação de missões espaciais, envolvendo duas Naves-Mãe em órbita de planetas distintos (Marte e Vénus), 16 *rovers* de superfície (8 por planeta) e uma infraestrutura de satélites intermediária. **Desenvolvido em Go e validado no emulador CORE**, o projeto demonstrou capacidade de operar sob condições adversas de rede com latência elevada ( $RTT > 2000$  ms) e perda significativa de pacotes (30%), mantendo a comunicação fiável em tempo real.

### 8.1. Arquitetura e Componentes

A solução adota uma arquitetura modular com quatro componentes principais: ***Mothership***, que funciona como o servidor central de coordenação, ***Rovers***, que representam as unidades cliente, e ***Ground Control***, a interface *web* de comando e observação. A comunicação entre componentes realiza-se através de dois protocolos personalizados complementares, implementados sobre diferentes camadas de transporte conforme os requisitos de cada tipo de tráfego.

### 8.2. Design dos Protocolos

#### 8.2.1. *MissionLink* (ML)

O *MissionLink* foi implementado intencionalmente sobre UDP para demonstrar como compensar as limitações de um protocolo não-fiável através de mecanismos aplicacionais robustos. O design assenta em três pilares: **arquitetura assimétrica**, com o *MissionLinkReceiver* na Nave-Mãe e o *MissionLinkSender* nos *rovers*, **serialização binária híbrida**, com um cabeçalho fixo de 17 bytes adicionando os possíveis campos variáveis e **fiabilidade aplicacional**, um esquema *stop-and-wait* com ACKs positivos, *timeout* de 2 segundos e máximo de 3 retransmissões.

O protocolo define quatro tipos de mensagens: ***MissionRequest***, podendo ter mais de 18 bytes, ***MissionAssignment***, sendo que a mensagem mais complexa pode ultrapassar os 44 bytes, ***MissionProgress***, com 22 bytes acompanhada de campos opcionais indicados por *flags*, e ***MissionAck***, sendo o único tipo de mensagem com um tamanho fixo de 26 bytes. A numeração sequencial permite **detetar duplicados e correlacionar respostas**, enquanto o mapeamento dinâmico de endereços elimina *handshakes* complexos, mantendo a agilidade do UDP.

### 8.2.2. *TelemetryStream* (TS)

O *TelemetryStream* garante uma transmissão contínua e fiável de dados de sensores através de TCP, assegurando uma entrega ordenada e íntegra. O *design* implementa **sessões persistentes**, que evitam *overhead* de múltiplos *handshakes*, **formato binário fixo de 64 bytes**, maximizando previsibilidade de largura de banda, e **processamento concorrente**, com uma *goroutine* dedicada por *rover* com acesso protegido por *Mutex*.

Uma característica distintiva é o mecanismo de ***load shedding* inteligente**: utilizando um canal com *buffer* de 100 mensagens e escrita não-bloqueante, o sistema **descarta pacotes mais recentes quando o canal está cheio**, impedindo que atrasos de processamento bloqueiem toda a rede de telemetria. A serialização recorre à **manipulação direta de memória byte-a-byte em little endian**, eliminando *parsing* de texto e oferecendo uma eficiência crucial dada a frequência elevada de atualizações. O transmissor implementa **reconexão automática em caso de falha**, tentando restabelecer a conexão em intervalos fixos até ao limite configurado.

### 8.2.3. API e *Ground Control*

A API de observação implementa arquitetura *REST* sobre *HTTP* com *JSON*, expondo quatro *endpoints* principais: **telemetria em tempo real**, com filtragem automática de dados obsoletos  $> 15\text{s}$ ; **listagem de missões**; **criação de missões**, com atribuição automática de duração baseada no tipo; e **health check** com sincronização temporal. A gestão de concorrência utiliza *sync.RWMutex*, permitindo leituras paralelas ilimitadas, enquanto garante exclusividade nas escritas.

O *Ground Control* excede os requisitos básicos ao oferecer uma **dashboard web interativa** com interface visual moderna, **atualização em tempo real via AJAX**, **terminal de sistema integrado**, **persistência local** e **comando e controlo ativo**. O cliente sincroniza o relógio com o tempo de simulação do servidor, garantindo consistência temporal independentemente do desfasamento local.

## 8.3. Resultados dos Testes

### 8.3.1. Teste de Concorrência

Com 16 *rovers* operacionais simultâneos (8 por planeta), ambas as Naves-Mãe processaram fluxos de telemetria e pedidos de missão concorrentes sem atrasos percetíveis. O *Ground Control* refletiu corretamente o estado de todos os veículos sem intermitências ou dados misturados. Verificou-se um **isolamento estrito de contextos**, isto é, cada Nave-Mãe geriu exclusivamente os seus *rovers* sem interferência, validando a arquitetura de canais concorrentes e demonstrando capacidade de escalar horizontalmente.

### **8.3.2. Fiabilidade do *MissionLink***

Sob 30% de perda de pacotes, a análise de tráfego, obtida através da utilização do *software Wireshark*, evidenciou o mecanismo de retransmissão funcionando precisamente, uma vez que ilustra as tentativas sucessivas com intervalo rigoroso de 2.0 segundos (93.09s, 95.11s, 97.12s) até uma recuperação bem-sucedida aos 99.14s. A Nave-Mãe respondeu imediatamente com os dados da missão e o protocolo concluiu-se com o *ACK* do *Rover*. **O sistema recuperou eficazmente das perdas severas**, garantindo a entrega de mensagens críticas através de retransmissão automática.

### **8.3.3. Desempenho sob Condições Adversas**

Com latência elevada ( $RTT > 2000ms$ ) e perda aleatória de pacotes, a sessão TCP manteve estabilidade sem desconexões inesperadas. Os registos evidenciaram um **atraso constante de ~ 1 segundo** (ex: envio às 23:46:28.105, com receção às 23:46:29.118) provocado pela latência física, confirmando que o sistema não acumula atraso de processamento adicional e mantém o débito em tempo real. Apesar das condições adversas, a Nave-Mãe **recebeu a sequência de telemetria completa** e ordenada, e o mecanismo de *timeout* do *MissionLink* adaptou-se às condições, realizando três tentativas antes da entrega bem-sucedida.

## **8.4. Conclusão**

O projeto demonstrou com sucesso a implementação de um sistema de comunicação robusto e escalável para ambientes com restrições severas de rede. A escolha de **protocolos complementares**: *MissionLink* sobre UDP para controlo com fiabilidade aplicacional e *TelemetryStream* sobre TCP para monitorização contínua, revelou-se acertada, permitindo otimizar cada canal de comunicação conforme as suas características específicas.

Os **mecanismos de fiabilidade implementados provaram ser eficazes** mesmo sob perda de 30% de pacotes e latências superiores a 2 segundos. A arquitetura concorrente baseada em *goroutines* e primitivas de sincronização garantiu o processamento sem bloqueios de 16 *rovers* simultâneos, validando a escalabilidade horizontal do sistema.

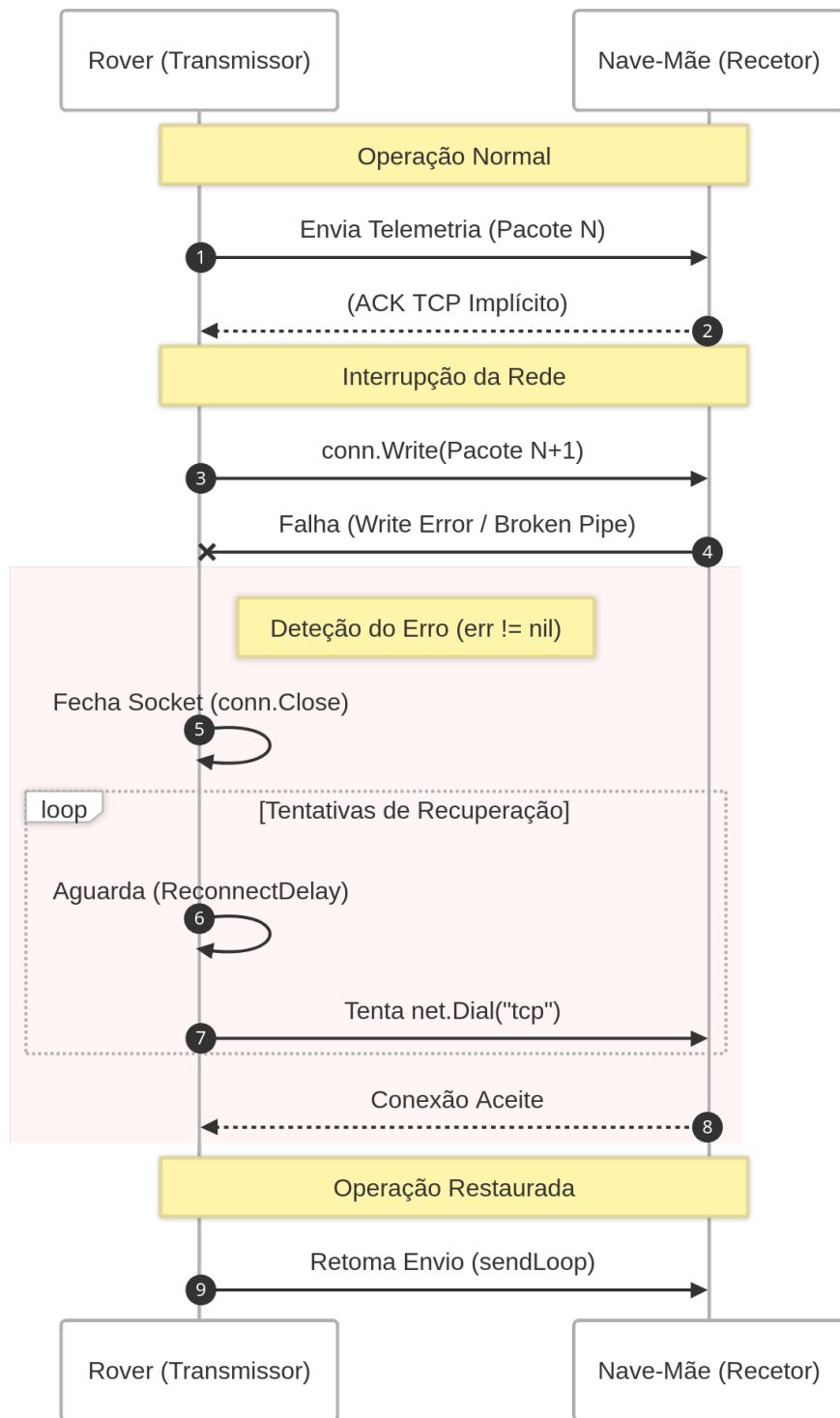
O **Ground Control**, ao transcender os requisitos básicos com uma interface web completa e funcional, evidencia como a integração cuidada de todos os componentes, desde os protocolos de baixo nível até à camada de apresentação, resulta numa experiência operacional superior. A capacidade de observar, comandar e controlar missões em tempo real através de um **dashboard intuitivo demonstra o valor de uma abordagem holística** ao design de sistemas distribuídos.

Em suma, o trabalho **alcançou todos os objetivos propostos**, demonstrando domínio dos conceitos fundamentais de comunicações por computador e capacidade de os aplicar na resolução de problemas complexos em ambientes desafiantes.

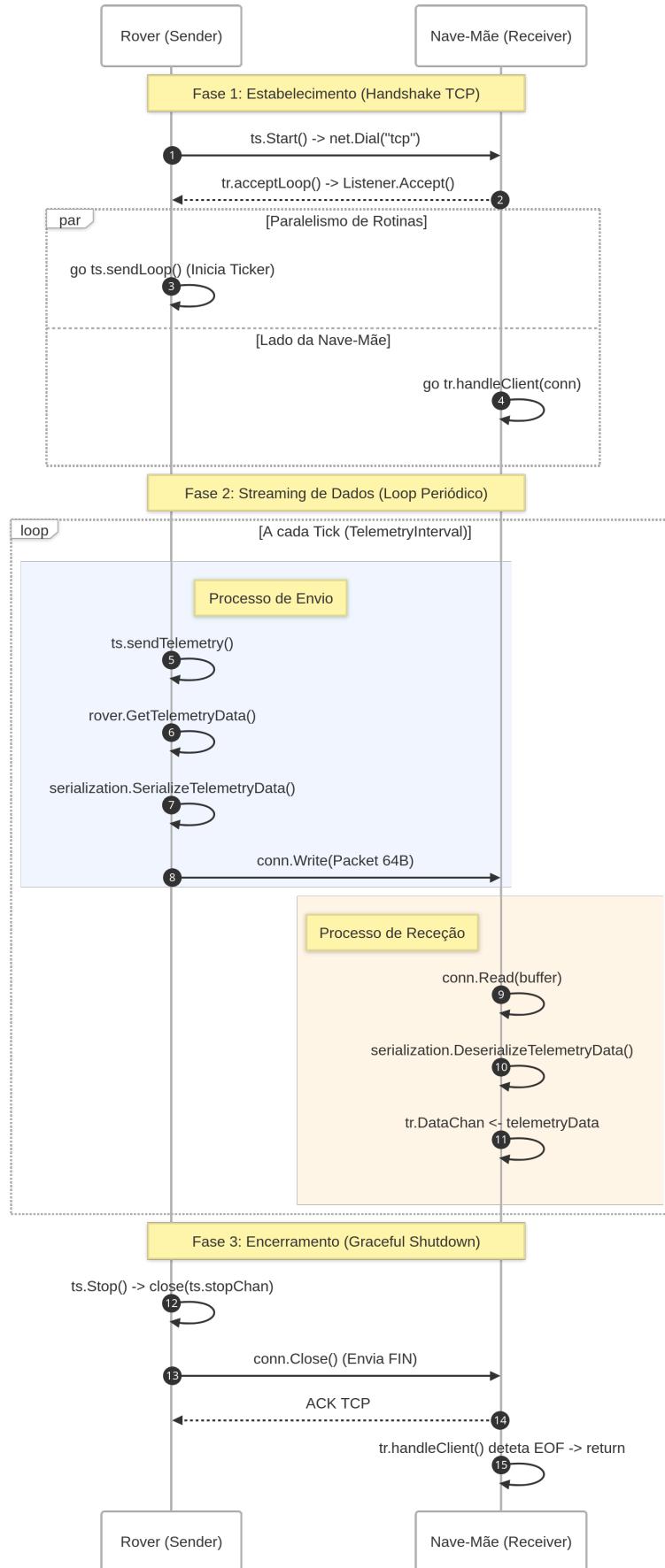
## 9. Bibliografia

- [1] Google, «Go Programming Language Documentation». [Em linha]. Disponível em: <https://go.dev/doc/>
- [2] António Costa, «Slides da Unidade Curricular de Comunicações por Computador».
- [3] U.S. Naval Research Laboratory, «Common Open Research Emulator (CORE)». [Em linha]. Disponível em: <https://github.com/coreemu/core>
- [4] K. W. Kurose J. F. & Ross, *Computer Networking: A Top-Down Approach*, 8.º ed. Pearson, 2020.
- [5] Rodrigo Pereira e António Duarte Costa, «Dockerized-Coreemu-Template». [Em linha]. Disponível em: <https://github.com/eivarin/Dockerized-Coreemu-Template>

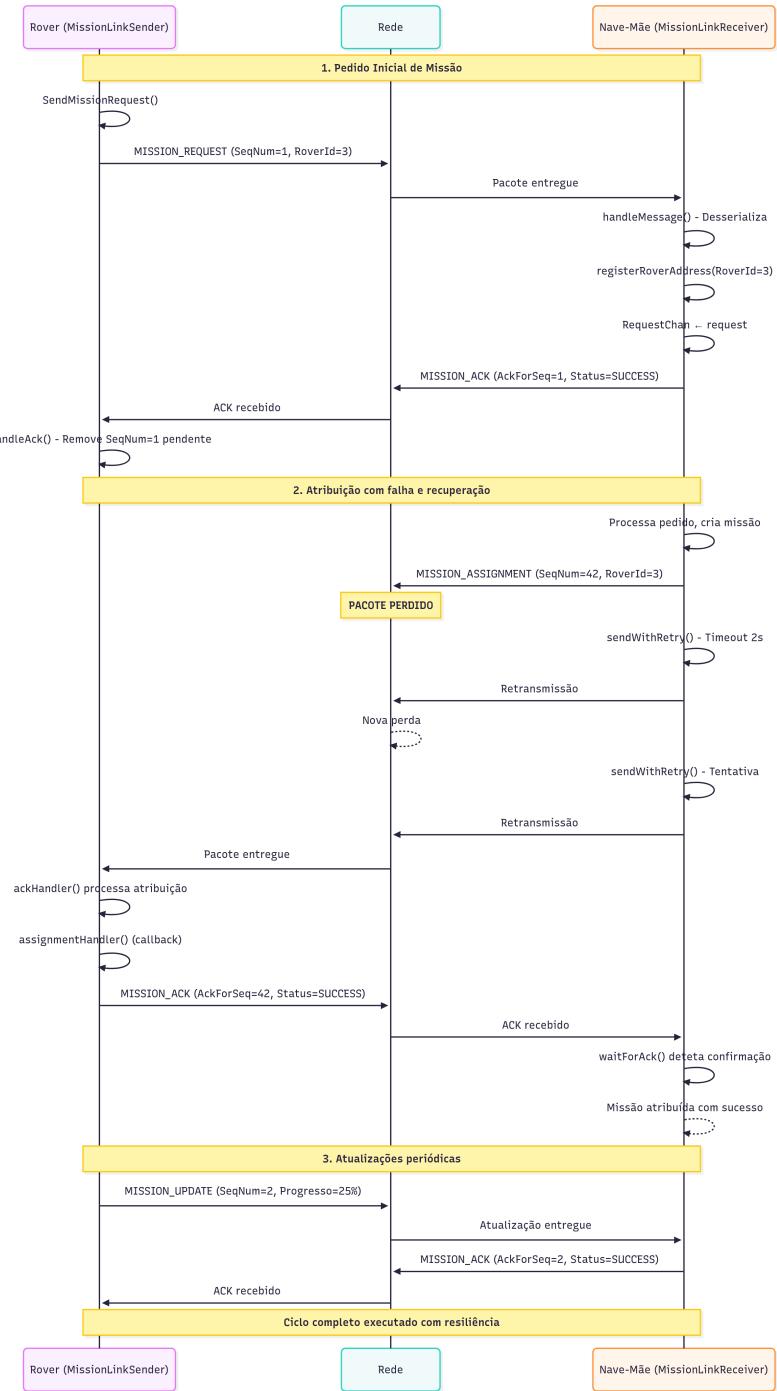
## 10. Anexos



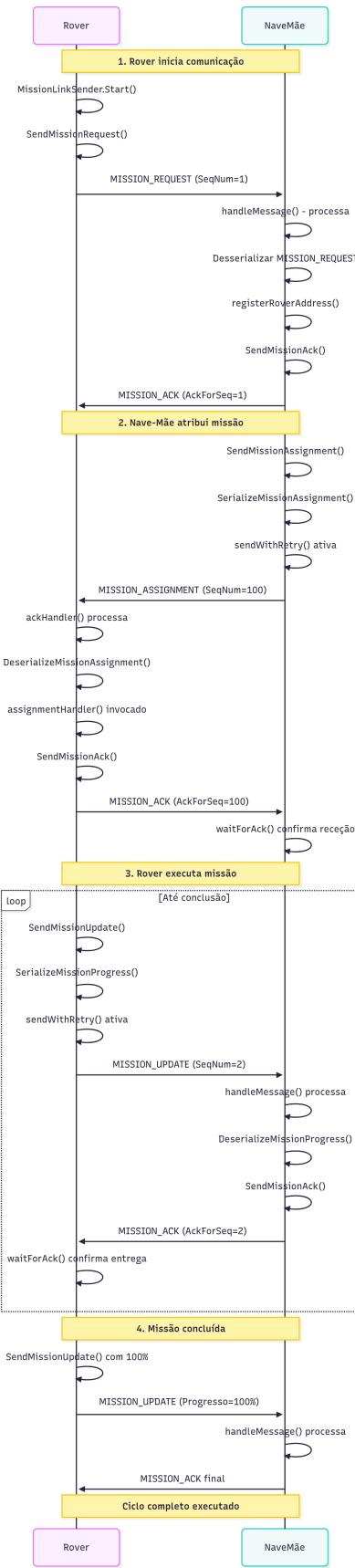
Anexo 1: Diagrama de Sequência do processo de recuperação em caso de erro (*Telemetry Stream*).



Anexo 2: Diagrama de Sequência do funcionamento de *Telemetry Stream*.



Anexo 3: Diagrama de Sequência do processo de recuperação em caso de erro (*Mission Link*).



Anexo 4: Diagrama de Sequência do funcionamento de *Mission Link*.