Comunicações por Computadores

Trabalho Prático 2

Gonçalo Sá, Luís Maciel e Pedro Pereira
Turno PL3 - Grupo 2
Universidade do Minho
e-mail: {a107376,a106896,a107332}@alunos.uminho.pt

FASE 1 — Preparação e Ambiente

Esta fase define as bases do sistema, o comportamento geral dos rovers e da Nave-Mãe, os tipos de missões suportadas, a estrutura dos dados trocados e os sistemas internos simulados (como bateria e sensores).

O objetivo é garantir que todas as entidades (Nave-Mãe, Rovers e Ground Control) partilham o mesmo formato de comunicação e simulam comportamentos realistas antes da implementação dos protocolos de rede.

Tipos de Missões

A Nave-Mãe é responsável por atribuir missões aos rovers através do protocolo MissionLink (UDP). Cada missão define o tipo de tarefa, os parâmetros de operação e o ritmo de comunicação. Os rovers podem ter várias missões ativas e reportam o progresso periodicamente à Nave-Mãe.

Quando o rover está em missão → speed = 1.0.

Quando está parado ou a carregar → speed = 0.0

1. scan_area

Descrição geral:
O rover percorre uma área pré-definida, de forma a simular a recolha de imagens ou dados topográficos. Serve para testar a movimentação contínua e as atualizações periódicas de progresso.

Parâmetros enviados pela Nave-Mãe:

- mission_id – identificador único da missão
- task – “scan_area”
- area – limites da área a percorrer (lista [[x1, y1], [x2, y2]])
- resolution – passo de varrimento entre linhas
- duration – tempo máximo (segundos)
- update_interval – intervalo entre relatórios

Comportamento do rover:
O rover move-se em padrão “zig-zag” dentro da área,

- Ele move-se no eixo X até ao limite direito;
- Quando chega lá, desce no eixo Y e inverte a direção do X;
- Repete até cobrir a área.

progress = (tempo / duration) * 100

, enviando atualizações com progresso, posição e estado. A missão termina quando a área é totalmente percorrida.

Exemplo de missão:

```
{
  "mission_id": "M-101",
  "task": "scan_area",
  "area": [[0, 0], [10, 10]],
  "resolution": 1.0,
  "duration": 300,
  "update_interval": 20
```

}

Exemplo de update:

{

"type": "missionlink",

"action": "mission_update",

"mission_id": "M-101",

"progress": 0.4,

"status": "in_progress",

"timestamp": 1730123456

}

2. collect_sample

Descrição geral:
O rover desloca-se até pontos específicos definidos pela Nave-Mãe e simula a recolha de amostras. Permite testar deslocações discretas e envio de dados pontuais.

Parâmetros enviados pela Nave-Mãe:

- mission_id – identificador único
- task – “collect_sample”
- points – lista de coordenadas (locais de recolha)
- sample_type – tipo de amostra (“rock”, “dust”, “ice”)
- duration – tempo total máximo
- update_interval – intervalo de atualização

Comportamento do rover:
O rover percorre os pontos por ordem, simula a recolha em cada um e envia

atualizações com progresso e ponto atual. A missão termina quando todos os pontos são visitados.

progress = current_index / len(points)

Exemplo de missão:

```
{
  "mission_id": "M-202",
  "task": "collect_sample",
  "points": [[2, 3], [6, 7], [9, 4]],
  "sample_type": "rock",
  "duration": 180,
  "update_interval": 15
}
```

Exemplo de update:

```
{
  "type": "missionlink",
  "action": "mission_update",
  "mission_id": "M-202",
  "progress": 0.66,
  "status": "in_progress",
  "last_point": [6, 7],
```

"timestamp": 1730123567

}

3. analyze_environment (extra)

Descrição geral:
O rover analisa condições ambientais (temperatura, radiação, poeira, etc.), simulando a recolha de dados científicos. Este tipo de missão gera leituras dinâmicas de sensores, úteis para testar telemetria variável.

O rover move-se em círculos pequenos aleatórios.

Parâmetros enviados pela Nave-Mãe:

- mission_id – identificador único
- task – “analyze_environment”
- area – zona de recolha
- sensors – lista de sensores a ativar (ex.: [“temperature”, “radiation”])
- duration – tempo máximo
- update_interval – frequência de envio

Exemplo de missão:

{

"mission_id": "M-303",

"task":"analyze_environment",

"area":[[5,5],[15,15]],

"sensors":["temperature","radiation","dust_level"],

"duration":300,

"update_interval":20

}

Exemplo de update:

{

"type":"missionlink",

"action":"mission_update",

"mission_id":"M-303",

"progress":0.9,

"status":"almost_done",

"data":{"temperature": -42, "radiation": 3.5},

"timestamp":1730124000

}

Missão	**Objetivo**	**Parâmetros principais**	**Dados enviados**
scan_area	Mapear área delimitada	area,resolution, duration,update_interval	Progresso, posição
collect_sample	Recolher amostras	points, sample_type,duration, update_interval	Progresso, ponto atual
analyze_environment	Medir condições ambientais	area,sensors,duration, update_interval	Progresso, leituras

Dados de Telemetria (Rover → Nave-Mãe)

Os rovers enviam dados de telemetría periódicos via TelemetryStream (TCP) e permitem à Nave-Mãe monitorizar o estado de cada unidade em tempo real.

Estrutura mínima:

```
{
  "rover_id": "R-001",
  "position": [x, y, z],
  "status": "in_mission",
  "battery": 85,
  "speed": 1.0
}
```

Campos definidos:

- rover_id – identificador do rover
- position – posição simulada
- status – estado (idle, in_mission, charging, etc.)
- battery – percentagem de carga
- speed – velocidade atual

Dados que o Rover precisa receber

Cada missão enviada pela Nave-Mãe contém as informações essenciais para a execução:

- mission_id – identificador da missão
- task – tipo de tarefa

- area/points – zona de operação
- duration – tempo máximo
- update_interval – frequência dos relatórios

Os dados são enviados via MissionLink e guardados localmente em active_missions.json.

Extras e Simulações

Além das missões, cada rover possui sistemas internos simulados que influenciam os dados de telemetria e a execução das tarefas.

1. Bateria

- Valor inicial: 100%
- Consumo base: 0.1%/s
- Consumo adicional por tarefa:
    - scan_area: +0.05%/s
    - collect_sample: +0.1%/s
    - analyze_environment: +0.02%/s
- Carregamento (Smart Charging): O rover gere a energia de forma autónoma com dois níveis de alerta:
    - Em Repouso (Idle): Inicia o carregamento automaticamente se a bateria descer abaixo de 20%.
    - Em Missão: O rover tenta prolongar a operação, mas força um carregamento de emergência se a bateria atingir um nível crítico de 5%.
- Impacto na Missão: Se o modo de carregamento for ativado durante uma missão (emergência < 5%), a missão é imediatamente abortada (reportada como aborted) e não é retomada. O rover prioriza a sua sobrevivência energética.
- Recuperação: Em modo charging, a bateria recupera a uma taxa de 1%/s. Ao atingir 100%, o rover regressa automaticamente ao estado idle, ficando disponível para novas ordens.

- Falha Total: Se a bateria atingir 0%, o rover deixa de operar. O software encerra conexão TCP, parando o envio de telemetria, o que é interpretado pelo Ground Control como estado "OFFLINE".

Exemplo:

{

"rover_id": "R-001",

"status": "charging",

"battery": 67

}

2. Sensores

Para garantir consistência nos dados, os valores dos sensores não são puramente aleatórios; são calculados utilizando funções trigonométricas (seno e cosseno) baseadas nas coordenadas (x, y) do rover, criando 'manchas' de temperatura ou radiação virtuais no mapa.

Sensor	Unidade	Intervalo	Frequência
temperature	°C	+5 a +35	update_interval
radiation	mSv	>5.0	update_interval
dust_level	%	0–100	update_interval

{

"temperature": -35.2,

"radiation": 2.8,

"dust_level": 76

}

3. Navegação e Posição

O progresso é calculado temporalmente: progress = (tempo_decorrido / duração_total) * 100. Esta abordagem garante que a barra de progresso no Ground Control avança de forma linear e previsível, independentemente da geometria complexa do trajeto do rover.

4. Resumo dos Sistemas Extra

Sistema	Variáveis	Controlado por	Enviado via
Bateria	battery, status	Rover	TelemetryStream
Sensores	temperature, radiation, dust_level	Rover	MissionLink
Falhas	low_signal	Rover	MissionLink
Navegação	position, speed	Rover	TelemetryStream
Modo de carga	status: "charging"	Rover/Nave-Mãe	TelemetryStream

FASE 2 — Design dos Protocolos

Esta secção descreve o design dos protocolos aplicacionais desenvolvidos no âmbito do trabalho prático.
O objetivo é definir, de forma clara e estruturada, o comportamento e o formato de comunicação entre a **Nave-Mãe**, os **Rovers** e o **Ground Control**.

Foram definidos dois protocolos aplicacionais principais:

- **MissionLink (ML)** — responsável pela comunicação crítica de missões, desenvolvido sobre **UDP**;

- **TelemetryStream (TS)** — responsável pela transmissão contínua de dados de monitorização, desenvolvido sobre **TCP**.

A estes protocolos junta-se a **API de Observação**, exposta pela Nave-Mãe via HTTP, que permite a integração com o módulo Ground Control.
O design apresentado estabelece a base para as fases seguintes de implementação e testes no ambiente CORE.

Estrutura Geral das Mensagens

Todas as mensagens trocadas entre a Nave-Mãe e os Rovers são transmitidas em formato **binário**, com um **cabeçalho fixo de 8 bytes** seguido de um **payload variável**.

O cabeçalho contém informações de controle (identificação, tipo e tamanho), enquanto o payload contém os dados da mensagem, codificados em **Pickle (Binário Python)**.

Esta estrutura permite:

- **Comunicação binária real** sobre UDP e TCP;
- **Parsing rápido e determinístico** (através do cabeçalho);
- **Leitura e depuração simples** (por o payload ser Pickle).

Formato binário padrão das mensagens

Campo	Tipo	Bytes	Descrição
Version	uint8	1	Versão do protocolo (atualmente 1)
MsgType	uint8	1	Identifica o protocolo: 1 = MissionLink, 2 =

			TelemetryStream
Action	uint8	1	Código da ação (ex: 1=new_mission, 2=ack, 3=mission_update, etc.)
Seq	uint16	2	Número de sequência da mensagem (big-endian)
Length	uint16	2	Tamanho do payload em bytes
Checksum	uint8	1	Soma módulo 256 de todos os bytes do payload
Payload	bytes	variável	Dados serializados em Pickle

Total do cabeçalho fixo: 8 bytes
Payload: variável (geralmente 50–200 bytes)

MissionLink (UDP)

Objetivo Geral

O **MissionLink** é o protocolo aplicacional responsável pela comunicação crítica entre a Nave-Mãe e os Rovers.
É utilizado para a atribuição, atualização e gestão de missões.
Como o **UDP** não garante entrega nem ordem de pacotes, o protocolo inclui mecanismos próprios de fiabilidade e controlo, garantindo retransmissões, confirmação de mensagens e deteção de erros.

Fluxo de Mensagens

A comunicação entre Rover e Nave-Mãe segue o fluxo apresentado na Tabela 1.

#	Direção	Action	Descrição	Campos principais
1	Rover → Nave-Mãe	request_mission (6)	O rover solicita uma missão.	rover_id,, seq
2	Nave-Mãe	mission_batch	Atribui uma missão.	mission_id, task, parâmetros

	→ Rover	(1)		
3	Rover → Nave-Mãe	ack (2)	Confirma receção.	mission_id, seq
4	Rover → Nave-Mãe	mission_update (3)	Atualiza progresso.	mission_id, progress, status
5	Nave-Mãe → Rover	cancel_mission (4)	Cancela uma missão ativa.	mission_id, reason
6	Ambos	error (5)	Reporta falha ou parsing incorreto.	code, message, seq

Exemplo de mensagem codificada

A) Pedido de missão (`request_mission`)

Cabeçalho:

Version = 1
MsgType = 1
Action = 6
Seq = 0001
Length = 22
Checksum = 7F

Payload (Objeto Pickle):
{ "rover_id": "R-002" }

Mensagem final (hex dump):
01 01 06 00 01 00 16 7F 7B 22 72 6F 76 65 72 5F 69 64 22 3A 20 22 52 2D 30 30 32 22 7D

B) Atualização de missão (`mission_update`)

Payload:

{ "mission_id":"M-462", "progress":0.5, "status":"in_progress" }

Cabeçalho (binário):
01 01 03 00 21 00 35 5D

Mensagem final:
01 01 03 00 21 00 35 5D 7B 22 6D 69 73 73 69 6F 6E 5F 69 64 22 3A 20 22 4D 2D 34 36 32 22 2C 20 22 70 72 6F 67 72 65 73 73 22 3A 20 30 2E 35 2C 20 22 73 74 61 74 75 73 22 3A 20 22 69 6E 5F 70 72 6F 67 72 65 73 73 22 7D

Mecanismos de Fiabilidade

Parâmetro	Valor	Função
`ACK_TIMEOUT`	2 s	Tempo máximo de espera por ACK
`MAX_RETRIES`	5	Número máximo de retransmissões
`UPDATE_INTERVAL`	5, 10, 15 ou 20 s	Frequência de envio de progresso
`MISSION_TIMEOUT`	300–600 s	Tempo máximo de execução

O rover retransmite mensagens críticas até receber confirmação.
A Nave-Mãe ignora duplicados e regista falhas após cinco tentativas.

Gestão de Estados

Estado	Descrição	Transições
`idle`	Rover à espera de missão	→ in_mission (ao receber missão) / → charging (bateria < 20%)

`in_mission`	Execução de missão ativa	→ charging (bateria < 5%, missão abortada) / → idle (conclusão)
`charging`	Modo de carregamento solar	→ idle (ao atingir 100%)
`offline`	Bateria a 0% ou perda de rede	Estado terminal (requer reinício)

TelemetryStream (TCP)

Objetivo geral

O **TelemetryStream** é o protocolo responsável pelo envio contínuo de dados de monitorização dos rovers para a Nave-Mãe.
A comunicação ocorre sobre TCP, garantindo entrega ordenada e fiável.

Cada rover estabelece uma ligação persistente com a Nave-Mãe e envia periodicamente dados de telemetria, contendo informação sobre posição, velocidade, estado e bateria.
A Nave-Mãe mantém uma tabela de dados atualizados para cada rover e utiliza essa informação para monitorizar o estado global da missão.

O envio ocorre de forma periódica, com um intervalo padrão de dois segundos.
Em caso de falha da ligação, o rover tenta reconectar automaticamente.
A utilização de TCP assegura que a telemetria não se perde nem chega fora de ordem, mantendo a consistência dos dados recebidos.

Fluxo de Mensagens

#	Direção	Action	Descrição	Campos principais
1	Rover → Nave-Mãe	connect (1)	Inicia ligação e identifica rover.	rover_id, timestamp

2	Rover → Nave-Mãe	telemetry_update (2)	Envia dados de estado e posição.	rover_id, position, battery, status, speed
3	Nave-Mãe → Rover	ack (3)	Confirma receção (opcional).	rover_id, seq
4	Rover → Nave-Mãe	heartbeat (4)	Mantém a ligação viva.	rover_id, timestamp
5	Rover → Nave-Mãe	disconnect (5)	Fecha ligação TCP.	rover_id, reason
6	Ambos	error (6)	Reporta falha.	code, message, timestamp

- Cada rover mantém uma ligação TCP persistente com a Nave-Mãe.
- A Nave-Mãe gere múltiplas conexões em paralelo, uma *thread* por rover ativo.
- As mensagens **telemetry_update** são enviadas a cada *update_interval*.
- O **TS_HEARTBEAT** é enviado esporadicamente em paralelo com as atualizações normais. Isto serve como mecanismo de redundância para validar a vivacidade da conexão TCP mesmo quando os dados de telemetria permanecem inalterados.
- Em caso de falha de ligação, o rover tenta reconectar automaticamente.
- A mensagem **ack** é opcional e usada apenas para debug ou teste de latência, não sendo necessária em operação normal.

Formato binário (igual ao MissionLink)

Campo	Tipo	Bytes	Descrição
Version	uint8	1	Versão (1)
MsgType	uint8	1	2 (TelemetryStream)
Action	uint8	1	Código da operação
Seq	uint16	2	Sequência
Length	uint16	2	Tamanho do payload
Checksum	uint8	1	Verificação
Payload	bytes	variável	Pickle com telemetria

Exemplo de mensagem `telemetry_update`

Payload:

```
{
  "rover_id": "R-001",
  "position": [15, 7, 0],
  "battery": 82,
  "status": "in_mission",
  "speed": 1.3,
  "timestamp": 1730123580
}
```

Cabeçalho:

```
01 02 02 00 42 00 67 9A
```

Mensagem final:

```
01 02 02 00 42 00 67 9A 7B 22 72 6F 76 65 72 5F 69 64 22 3A 20
22 52 2D 30 30 31 22 2C ...
```

Mecanismos de Fiabilidade

Para compensar as limitações do UDP, o MissionLink implementa mecanismos de fiabilidade a nível aplicacional.
Cada mensagem possui um número de sequência que permite detetar duplicados e perdas.
Mensagens importantes, como a atribuição de missões, exigem confirmações explícitas (ACK).
É definida uma temporização de 2 segundos para aguardar respostas e, caso não sejam recebidas, a mensagem é retransmitida até um máximo de cinco tentativas.
Em caso de falha persistente, o rover considera a missão abortada.
Mensagens inválidas ou fora de sequência originam mensagens de erro com o respetivo código e descrição.

Gestão de Estados

O comportamento do protocolo é definido através de máquinas de estados simples.

O sistema suporta múltiplos rovers em operação simultânea.

A Nave-Mãe gere cada ligação TCP (TelemetryStream) através de threads dedicadas, permitindo receção concorrente de telemetria.

O canal UDP (MissionLink) é tratado de forma assíncrona, permitindo que múltiplos rovers enviem pedidos e atualizações sem bloqueios.

1. **Estados do Rover**

Estado	Descrição	Transição
idle	Rover inativo, à espera de missão	→ in_mission (recebeu missão) / → charging (bateria < 20%)
requesting	Enviou pedido de missão e aguarda resposta	→ in_mission (ao receber missão)

in_mission	Execução de missão ativa	→ charging (bateria < 5%, missão abortada) / → idle (conclusão com sucesso)
charging	Em modo de carregamento solar	→ idle (ao atingir 100%)
offline	Bateria a 0% ou perda de rede	Estado terminal (requer reinício da simulação)

2. **Estados da Nave-Mãe**

Estado	Descrição	Transição
waiting_request	À espera de pedidos de missão	→ assigning
assigning	Atribui missões e aguarda ACK	→ monitoring
monitoring	Recebe atualizações de progresso	→ completed ou cancelled
completed	Missão terminada com sucesso	→ waiting_request

Temporizações e Parâmetros Principais

Parâmetro	Valor	Função
ACK_TIMEOUT	2 s	Tempo máximo de espera por confirmação
MAX_RETRIES	5	Retransmissões máximas permitidas
TELEMETRY_RATE	2 s	Intervalo entre mensagens de estado via TCP
MISSION_TIMEOUT	300-600 s	Valor configurável consoante tipo de missão

Tratamento de Erros

Falhas de rede, perda de pacotes ou problemas internos dos rovers são reportados através de mensagens de erro.
A Nave-Mãe pode cancelar a missão ou registar o erro para análise posterior.
Cada mensagem de erro contém um código numérico e uma breve descrição do problema detetado.

API de Observação

Para permitir a monitorização em tempo real, a Nave-Mãe expõe uma API HTTP acessível ao nó Ground Control.

A API é separada dos protocolos principais para evitar interferência com o canal de controlo e garantir isolamento entre as comunicações críticas (UDP/TCP) e o acesso externo (HTTP).

Esta API fornece endpoints para consulta de estado e telemetria.

Os principais endpoints definidos são:

- **/rovers** – lista todos os rovers ativos e respetivo estado;
- **/missions** – apresenta missões ativas e concluídas;
- **/telemetry/<rover_id>** – devolve os últimos dados de telemetria de um rover;
- **/events** – fornece um fluxo contínuo de atualizações (opcional, via WebSocket ou SSE).

A API utiliza formato JSON e segue uma arquitetura REST simples e compatível com aplicações externas de visualização.

Integração no Ambiente CORE

Os protocolos definidos serão implementados e testados no simulador CORE de forma distribuída.

Cada instância de rover executa os clientes ML e TS em threads independentes, permitindo a execução paralela e contínua de missões.

O servidor central (Nave-Mãe) gere várias conexões TCP em simultâneo e processa mensagens UDP de forma assíncrona.

O Ground Control liga-se por HTTP à Nave-Mãe através de uma rede sem interferência.
As condições de latência, jitter e perda configuradas nos links do CORE simulam as comunicações espaciais entre satélites e rovers.

FASE 3 — Implementação do MissionLink (ML)

Esta fase teve como objetivo implementar o protocolo MissionLink (ML), o sistema de comunicação crítica entre a Nave-Mãe e os rovers, conforme os requisitos do enunciado. Operando sobre **UDP** , o principal desafio foi a implementação de mecanismos de fiabilidade a nível aplicacional para compensar a natureza não-confiável deste transporte.

A implementação segue o formato de protocolo definido na Fase 2, utilizando um cabeçalho binário de 8 bytes (que inclui `MsgType=1`, `Action`, `Seq` e `Checksum`) seguido de um payload serializado (Pickle).

Arquitetura do Sistema

O sistema é composto por um servidor assíncrono na Nave-Mãe e um cliente multi-threaded no Rover.

1. Servidor ML (Nave-Mãe)

O servidor (`navemae/missionlink_server.py`) foi desenhado para ser altamente concorrente e *stateless* (no sentido de gestão de ligação).

- **Servidor UDP Concorrente:** O servidor opera num *socket* UDP (Porta 5000). Ao receber *qualquer* pacote, ele instancia uma nova *thread* (`threading.Thread(target=handle_request, ...)`), passando-lhe os dados e o endereço do remetente.
- **Processamento Assíncrono:** Este modelo permite que o servidor processe múltiplos pedidos (pedidos de missão, ACKs, *updates* de progresso) de múltiplos rovers em paralelo, sem bloqueios.
- **Gestão de Estado Centralizada:** A função `handle_request` atua como um *dispatcher*. Ela descodifica o pacote e, com base na `action` (ex: `ML_UPDATE`), chama as funções apropriadas do módulo `navemae/state/rover_state.py` (como `update_mission`). O `rover_state` utiliza um `threading.Lock` para garantir que todas as atualizações ao estado global dos rovers são atómicas e seguras (*thread-safe*).

2. Cliente ML (Rover)

O cliente (`rover/missionlink_client.py`) é o componente mais complexo, pois implementa a lógica da missão e a gestão de estado.

- **Arquitetura Dual-Thread:** O cliente ML opera com duas *threads* principais:
    1. A *thread* principal (`start_missionlink`) é responsável por *enviar* dados (pedir missão, enviar *updates* de progresso na `run_mission`).
    2. Uma *thread* de escuta (`handle_server_messages`) é iniciada para *receber* mensagens da Nave-Mãe (como `ML_NEW_MISSION` e ACKs).
- **Execução da Missão:** A função `run_mission` implementa a lógica de negócio descrita na Fase 1 do relatório. Ela recebe a missão, entra num *loop* e, com base na `task` (`scan_area`, `collect_sample`, `analyze_environment`), simula o movimento, atualiza a posição local e envia pacotes `ML_UPDATE` nos intervalos corretos.

Fluxo do Protocolo e Mecanismos de Fiabilidade

Para cumprir o requisito de fiabilidade sobre UDP, foi implementado um protocolo de *handshake* (confirmação) para a atribuição de missões:

1. **Solicitação (Rover):** O rover inicia a comunicação enviado um pacote `ML_REQUEST` (Ação 6) para o servidor.
2. **Atribuição (Nave-Mãe):** O servidor recebe o pedido, gera uma nova missão (usando `get_next_mission_id`) e envia-a ao rover num pacote `ML_NEW_MISSION` (Ação 1).
3. **Confirmação (Rover):** O rover recebe a missão e envia um `ML_ACK` (Ação 2). Imediatamente após o envio, o rover transita para o estado `in_mission` e inicia a execução, assumindo que a confirmação chegou à Nave-Mãe. Caso o ACK se perca, a Nave-Mãe reenviará a missão após o *timeout*, e o rover tratará o duplicado descartando-o mas reenviando o ACK.
4. **Execução:** Durante a `run_mission`, o rover envia pacotes `ML_UPDATE` (Ação 3). Estes não são confirmados (ACK) pela Nave-Mãe, uma decisão de design para reduzir o tráfego, assumindo que a perda de um *update* não é catastrófica, pois o próximo corrigirá o estado.
5. **Conclusão:** No final da missão, o rover envia `ML_COMPLETE` (Ação 7).

Este *handshake* de 3 vias (Rover-Nave-Rover) garante que uma missão é atribuída e confirmada por ambas as partes antes da execução. A fiabilidade é complementada pelo `Checksum` no cabeçalho binário, que permite descartar pacotes corrompidos.

FASE 4 — Implementação do TelemetryStream (TS)

Esta fase detalha a implementação do protocolo TelemetryStream (TS), responsável pela comunicação contínua e fiável de dados de monitorização, conforme os requisitos do enunciado. Este sistema corre sobre **TCP**, garantindo a entrega ordenada e sem perdas dos dados de estado dos rovers para a Nave-Mãe.

O objetivo é permitir que a Nave-Mãe tenha uma visão em tempo real do estado de todos os rovers, gerindo múltiplas ligações em simultâneo.

Arquitetura do Sistema

O TS é composto por um servidor central na Nave-Mãe e um cliente dedicado em cada rover, ambos a utilizar o protocolo binário (cabeçalho + payload Pickle) definido na Fase 2.

1. Servidor TS (Nave-Mãe)

O servidor TS (`navemae/telemetry_server.py`) é robusto e está preparado para alta concorrência:

- **Multi-Threading:** O servidor corre num *socket* TCP dedicado (Porta 6000). Ao receber uma nova ligação (`srv.accept()`), ele instancia imediatamente uma nova thread `handle_client` para gerir esse rover específico. Isto cumpre o requisito de gestão de múltiplos rovers em paralelo.

- **Gestão de Sessão (Anti-Concorrência):** Para resolver o problema de rovers duplicados (ex: arranque acidental de uma segunda instância `R-001`), o servidor implementa uma política de **"Última Ligação Vence"**. Um dicionário global (`_ACTIVE_CONNECTIONS`) mapeia o `rover_id` à respetiva *socket* TCP. Se um `R-001` tentar conectar-se enquanto já existe uma sessão ativa para esse ID:
    1. O servidor deteta o conflito no dicionário;
    2. A ligação antiga é **imediatamente encerrada** (o servidor fecha a *socket*), forçando a terminação da instância "zombie" anterior (que deteta a quebra de ligação e encerra a sua execução);
    3. A nova ligação é aceite e registada como a ativa. Esta abordagem previne inconsistências de dados e garante que o comando está sempre com a instância mais recente do rover.
- **Armazenamento de Estado (Thread-Safe):** Cada vez que recebe uma mensagem `TS_UPDATE`, a thread `handle_client` chama a função

`update_telemetry`. Esta função, localizada no módulo `navemae/state/rover_state.py`, utiliza um `threading.Lock` global para garantir que o estado centralizado do rover é atualizado atomicamente, sem conflitos com as threads do MissionLink ou da API.

2. Cliente TS (Rover)

O cliente (`rover/telemetry_client.py`) é um componente crítico que corre numa thread dedicada (`ts_thread`) no processo do rover.

- **Ligação Persistente:** O cliente estabelece uma ligação TCP de longa duração com a Nave-Mãe. Ao conectar-se, envia uma mensagem `TS_CONNECT` (Ação 1) para se identificar.
- **Loop de Telemetria:** A thread entra num `telemetry_loop` que, a cada 2 segundos, envia uma mensagem `TS_UPDATE` (Ação 2).
- **Payload:** O payload enviado cumpre todos os requisitos do enunciado, incluindo `rover_id`, `position`, `status`, `battery` e `speed`.
- **Heartbeat:** Ocasionalmente, o cliente envia uma mensagem `TS_HEARTBEAT` (Ação 4) para garantir que a ligação TCP é mantida ativa, mesmo que não haja novos dados, e para que o servidor saiba que o rover continua online.

Gestão de Estado e Concorrência Interna (Rover)

O maior desafio no design do rover foi a partilha de estado entre a *thread* do MissionLink (que executa missões) e a *thread* do TelemetryStream (que gere a bateria). A solução implementada baseia-se num modelo de memória partilhada (`rover_identity`) com acesso coordenado:

1. **O TS gere a Bateria:** A thread de telemetria é a única responsável por calcular o decaimento e carregamento da bateria com base no tempo decorrido (`dt`).
2. **O ML informa a Tarefa:** Para calcular o consumo corretamente, o TS precisa de saber a tarefa atual (ex: `scan_area` gasta mais que `idle`). Isto é

resolvido em `rover/main.py`, passando a função `get_current_task` (do módulo MissionLink) como *callback* para dentro da *thread* de telemetria.

3. **Lógica de "Smart Charging":** A *thread* TS monitoriza a bateria a cada ciclo e impõe mudanças de estado globais:
    - **Em Repouso (Idle):** Se a bateria descer abaixo de **20%**, o TS chama imediatamente `set_status("charging")`.
    - **Em Missão:** O sistema permite um consumo mais profundo, forçando o carregamento apenas se a bateria atingir o nível crítico de **5%**.
4. **Reação do MissionLink:** A função `set_status` é *thread-safe* e altera o estado global. A *thread* do MissionLink (na função `run_mission`) verifica este estado periodicamente. Se detetar que o estado mudou para `charging`, a missão é **imediatamente abortada** (reportada como "aborted") para proteger o rover, não sendo retomada posteriormente.
5. **Recuperação:** Quando a bateria atinge **100%**, o TS chama `set_status("idle")`, libertando o rover para receber novas missões.

Esta arquitetura garante que ambos os protocolos reagem um ao outro: o MissionLink consome energia que o TelemetryStream monitoriza, e o TelemetryStream pode interromper o MissionLink para garantir a sobrevivência do rover.

FASE 5 — Implementação da API de Observação e Ground Control

Para permitir a monitorização e controlo da missão em tempo real, foi desenvolvido um sistema híbrido composto por uma API REST na Nave-Mãe e uma interface web para o operador.

API de Observação

A Nave-Mãe expõe uma API REST na porta 8000, implementada com a biblioteca **Flask**. Esta API corre numa *thread* independente (`start_api_server`), permitindo que o servidor HTTP funcione em paralelo com os servidores UDP (MissionLink) e TCP (TelemetryStream).

- **Gestão de Concorrência:** O acesso aos dados é protegido por locks. Quando a API lê o estado para enviar ao frontend, adquire o lock global do módulo `rover_state`, garantindo que não lê dados parciais ou incoerentes enquanto uma thread de telemetria está a atualizar a posição de um rover.
- **Integração com MissionLink:** O endpoint não comunica diretamente com os rovers. Em vez disso, coloca a missão numa fila de espera segura (`PENDING_MISSIONS` no `missionlink_server.py`). Quando o rover alvo envia o seu próximo `ML_REQUEST`, o servidor verifica esta fila e, se houver uma missão agendada, atribui-a nesse momento.
- **Endpoints Principais:**
    - `GET /api/state`: Devolve o JSON completo com a frota (Posição, Bateria, Status, Velocidade).
    - `GET /api/history`: Devolve o log das últimas 10 missões (Concluídas ou Abortadas).
    - `POST /api/missions`: Recebe comandos do Ground Control para agendar novas tarefas.

Ground Control

O Ground Control é uma aplicação desenvolvida em HTML, CSS e JavaScript, desenhada para ser leve e executada diretamente no browser.

- **Arquitetura de Comunicação:** A interface mantém a sincronização com a Nave-Mãe através de um ciclo de *polling* de 1 segundo. A cada ciclo, o cliente executa pedidos `GET` aos endpoints de estado e histórico para atualizar a visualização.

- **Lógica de Cliente:** Antes de enviar uma missão, o código JavaScript realiza validações locais para proteger a integridade do sistema:
    - **Verificação Offline:** Impede o envio de ordens se o rover estiver marcado como "OFFLINE" ou sem bateria.
    - **Gestão de Conflitos:** Se o operador tentar enviar uma missão para um rover que já está ocupado (`in_mission`) ou a carregar (`charging`), a interface deteta o estado e solicita uma confirmação explícita para adicionar a missão à **Fila de Espera** da Nave-Mãe, em vez de rejeitar o comando ou causar erros no servidor.

FASE 6 — Validação e Testes

O sistema foi validado num ambiente de emulação local, executando todos os componentes em simultâneo e simulando condições adversas.

Cenários de Teste Realizados

- **Teste A: Gestão de Sessões (Anti-Concorrência):**
    - *Cenário:* Iniciou-se uma segunda instância do rover com o mesmo ID "R-001" enquanto a primeira estava ativa.
    - *Resultado:* O servidor de telemetria detetou a duplicação no dicionário `_ACTIVE_CONNECTIONS`. Encerrou imediatamente a *socket* da instância antiga e aceitou a nova. A instância antiga detetou a quebra de ligação (`ConnectionResetError`) e terminou a execução, garantindo a unicidade do rover na rede.
- **Teste B: Ciclo de Missão e Fila de Espera:**
    - *Cenário:* O operador enviou uma missão via Web enquanto o rover ainda estava a processar a anterior.
    - *Resultado:* A nova missão ficou retida em `PENDING_MISSIONS`. Assim que o rover terminou a primeira tarefa e enviou um novo

`ML_REQUEST`, a Nave-Mãe entregou a segunda missão, validando o sistema de agendamento assíncrono.

- **Teste C: Gestão Crítica de Energia (Aborto):**
  - *Cenário:* Configurou-se uma missão longa para um rover com 10% de bateria.
  - *Resultado:* O rover iniciou a missão. Ao atingir 5% de bateria, o `telemetry_client` forçou o estado `charging`. O `missionlink_client` detetou a mudança de estado, abortou a navegação imediatamente e enviou um `ML_COMPLETE` com status `aborted`. O Ground Control exibiu uma notificação vermelha de falha, validando a prioridade de sobrevivência do rover.
- **Teste D: Persistência de Dados:**
  - *Cenário:* A Nave-Mãe foi reiniciada abruptamente.
  - *Resultado:* Ao arrancar, o servidor carregou o ficheiro `rover_state.json`, restaurando a última posição e nível de bateria conhecidos de todos os rovers, evitando a perda de contexto da missão.

Conclusão

Este trabalho permitiu implementar uma arquitetura robusta de comando e controlo espacial simulada. A separação clara entre tráfego crítico (MissionLink/UDP com fiabilidade aplicacional) e monitorização (TelemetryStream/TCP) provou ser eficaz.

A introdução de mecanismos avançados como **Locks globais** para *thread-safety*, **gestão de sessões únicas** para evitar conflitos de identidade e **serialização híbrida** (Header Binário + Payload Pickle) resultou num sistema estável, capaz de gerir múltiplos rovers e reagir autonomamente a condições críticas de bateria.