TP2: Perdidos no espaço

Grupo 3 - Turno PL7

2025-12-5

Authors:
Pedro Argainha a104351
João Machado a104084
Miguel Santos a72443

Universidade do Minho
2025
Comunicações por Computador

Sumário

1 | Introdução 4
2 | Estrutura de um rover 5
2.1 | Variaveis de um rover 5
2.2 | Métodos do rover 5
3 | ML (Mission Link) 7
3.1 | Objetivo do MissionLink 7
3.2 | Estrutura das Mensagens 7
3.2.1 | Header do Protocolo 7
3.2.2 | Payloads Específicos 8
3.3 | Tipos de Mensagens e Semântica 9
3.4 | Fluxo Normal do Protocolo 11
3.5 | Mecanismos de Fiabilidade 12
3.5.1 | Numeração de Sequência 13
3.5.2 | Confirmações Explícitas (ACK) 13
3.5.3 | Timeouts e Retransmissões 13
3.5.4 | Deteção de Duplicados 15
3.6 | Repetição Segura na Atribuição de Missões 16
3.7 | READY com Confirmação Implícita 18
3.8 | Ausência de Trabalho (NOMISSION) 19
3.9 | Tratamento de Mensagens Fora de Contexto 20
3.10 | Garantias Formais do Protocolo 21
3.11 | Tratamento de Casos Extremos 21
3.11.1 | Falha Persistente de Comunicação 22
3.11.2 | Abandono Controlado e Não-Bloqueio 22
3.11.3 | Limite de Tentativas e Compromisso Latência-
-Fiabilidade 22
4 | TS (Telemetry System) 24
4.1 | Estrutura da mensagem 24
4.2 | Detalhes do protocolo 25
4.3 | Diagrama de sequencia do TS 26
4.4 | Implementação em python 27

5 | GC (Ground Control) 28
5.1 | Arquitetura e Função 28
5.2 | Funcionalidades de Monitorização 28
5.3 | Atribuição Manual de Missões 28
5.3.1 | Interface de Configuração 28
5.3.2 | Transmissão ao Servidor 29
5.3.3 | Processamento na Nave-Mãe 29
5.3.4 | Prioridade e Atribuição 29
5.3.5 | Confirmação e Limpeza 29
5.3.6 | Benefícios e Casos de Uso 30
5.4 | Modelo Cliente-Servidor 30
5.5 | Diagrama de conexão 31
5.6 | Transferência de dados 32
5.7 | Implementação em Python 32
5.8 | Segurança e Robustez 33
6 | Inicialização da Nave-Mãe e Gestão de Cenários 35
6.0.1 | Processo de Inicialização da Nave-Mãe 35
6.0.2 | Definição de Cenários 35
6.0.3 | Interação entre Cenários e MissionLink 37
7 | Topologia 38
8 | Testes aplicados 39
8.1 | Testes e Validação 39
8.1.1 | Testes funcionais com 3 rovers 39
8.1.2 | Validação por cenários 39
8.1.3 | Testes de stress e tolerância a perdas 40
8.1.4 | Considerações finais 40
9 | Conclusão 41

1 | Introdução

Este trabalho prático teve como objetivo desenvolver e testar dois protocolos aplicacionais: MissionLink (ML) sobre UDP e TelemetryStream (TS) sobre TCP, para a comunicação entre uma Nave-Mãe e vários rovers numa missão espacial simulada. Posteriormente ja no ambiente CORE, foram configuradas e avaliadas comunicações em rede com múltiplos saltos, abordando desafios como latência, perda de pacotes e fiabilidade. Além disso, foi implementada uma API de Observação e um módulo Ground Control para visualizar em tempo real o estado das missões e dos rovers.

2 | Estrutura de um rover

Neste modulo iremos explicar a nossa abordagem e interpretação do que é um rover e todas as suas especificações. Como a nava mãe ja ia ter bastante carga foi tentado passar o máximo de carga possível para o rover de maneira a distribuir melhor toda a carga de trabalho.

2.1 | Variaveis de um rover

Um rover vai ser definido pelas seguintes variáveis.

```
    id: int
    tick: float = 1.0 #numero usado para calcular movimento de tick em
tick
    pos_x: float = 0.0
    pos_y: float = 0.0
    pos_z: float = 0.0
    destino: Tuple[float, float, float] = (0.0, 0.0, 0.0)
    velocidade: float = 0.0
    direcao: float = 0.0
    bateria: float = 100.0
    state: int = 0  # 0=free, 1=working, 2=em movimento, 3=erro
    proc_use: float = 0.0
    storage: float = 0.0
    sensores: int = 0 #numero de sensores avariados
    freq: int = 0.4  # mensagens/updates por segundo
    dirty: bool = False #usado para saber se tem informação nova
    missao:int = 0 #missão atribuida ao rover (0=NoMission)
    progresso: int = 0 #progresso em % da missão atual
    duracao: int = 0 #duração total da missão atual
    working: int = 0 #tempo atual a relalizar missão
```

Desta forma, todos os rovers tem a informação da sua localização, e seu objetivo. E desta forma e possível calcular através do tick e da velocidade a sua deslocação em cada intervalo de tempo. Também cada rover irá saber a sua missão e tempo atribuído a mesma e consoante o tempo a passar o progresso da mesma e atualizado. Optamos por usar um atributo *dirty* é usado para sempre que algum dos valores principais do rover e alterado este atributo fica a 1 e apenas volta a 0 quando envia para a nave mãe a sua telemetria.

2.2 | Métodos do rover

```
    def updateInfo
```

```
(self,x,y,z,destino,vel,dir,bat,estado,proc,sto,sens,freq,missao,pro):
    def limpaDity (self):
    def to_dict(self) -> dict: #Método usado para passar de rover para
JSON
    def ajustarEstatisticas(self, mission: str): #Método usado para
ajustar stats do rover de sensores/bateria/...
    def iterar(self):
    def resetarWork(self): #Método usado para acabar trabalho atual
    def atribiuMission(self,miss):
    def traduzEstado(self)->str: #Método usado para traduzir o estado
para str
    def to_string(self) -> str: #Método usado para passar de rover-
>String
    def to_stringProgresso(self) -> str: #Método usado para passar o
progresso->String
    def from_dict(data: dict) ->: #Método usado para passar de JSON para
Rover
    def update_from_dict(self, data: dict): #Método usado para dar
update a partir de um JSON
```

A partir destes métodos conseguimos garantir que todos os rovers são independentes, implementando internamente toda a lógica necessária para gerir estado, executar tarefas e converter dados para os formatos usados pelos protocolos de comunicação.

3 | ML (Mission Link)

3.1 | Objetivo do MissionLink

O MissionLink (ML) constitui o protocolo aplicacional responsável pela coordenação e gestão de missões entre a Nave-Mãe e os rovers. Enquanto o TelemetryStream assegura a monitorização contínua do estado operacional, o ML dedica-se exclusivamente à comunicação crítica relacionada com a atribuição, execução e conclusão de missões.

O protocolo foi concebido para garantir que cada missão é atribuída de forma fiável a um rover, que o progresso da sua execução é reportado de forma consistente, e que a conclusão é confirmada inequivocamente. Esta separação de responsabilidades permite que o sistema mantenha uma visão clara do estado lógico de cada missão, independentemente do fluxo contínuo de telemetria.

A coordenação de missões através do ML segue um modelo de requisição-resposta, onde o rover manifesta disponibilidade para receber trabalho e a Nave-Mãe responde com uma tarefa adequada ou indica a ausência temporária de missões. Uma vez atribuída, a missão passa a ser gerida através de um ciclo de atualizações periódicas que permitem à Nave-Mãe acompanhar o progresso em tempo real, sem depender exclusivamente dos dados de telemetria genéricos transmitidos via TCP.

3.2 | Estrutura das Mensagens

O protocolo MissionLink utiliza um formato binário estruturado, composto por um header fixo de 20 bytes seguido de um payload de comprimento variável. Esta abordagem garante eficiência na transmissão e permite validação rigorosa da integridade das mensagens, aspeto crítico quando se utiliza UDP como protocolo de transporte.

3.2.1 | Header do Protocolo

O header ML contém toda a informação necessária para o controlo do protocolo, independentemente do tipo de mensagem. A estrutura segue o formato *big endian* (network byte order), garantindo interoperabilidade entre diferentes arquiteturas de sistema.

Tabela 1 : Estrutura do header do protocolo MissionLink

Campo	Tipo	Tamanho	Descrição
`version`	uint8	1 byte	Versão do protocolo (atualmente 1)
`msg_type`	uint8	1 byte	Tipo de mensagem (READY, MISSION, etc.)
`flags`	uint8	1 byte	Flags de controlo (NEEDS_ACK, RETX, etc.)
`hdr_len`	uint8	1 byte	Tamanho do header (fixo: 20 bytes)
`seq`	uint32	4 bytes	Número de sequência da mensagem
`ack`	uint32	4 bytes	Número de sequência sendo confirmado
`stream_id`	uint16	2 bytes	Identificador do rover (stream)
`payload_len`	uint16	2 bytes	Tamanho do payload em bytes
`checksum`	uint32	4 bytes	CRC32 do payload

Os campos `seq` e `ack` constituem a base do mecanismo de fiabilidade implementado. Cada mensagem enviada possui um número de sequência único e crescente, permitindo ao recetor identificar duplicados e processar mensagens de forma monotónica. O campo `ack` é utilizado para confirmar a receção de mensagens anteriores, criando um canal bidirecional de confirmação.

O campo `flags` permite configurar o comportamento esperado para cada mensagem. A flag `NEEDS_ACK` indica que o emissor espera uma confirmação explícita, enquanto `RETX` assinala uma retransmissão devido a timeout. A flag `ACK_ONLY` identifica mensagens que servem exclusivamente para confirmar receção, sem payload adicional.

O `checksum` utiliza o algoritmo CRC32 sobre o payload, fornecendo uma camada adicional de validação de integridade para além das garantias (limitadas) oferecidas pelo UDP. Esta redundância é essencial para detetar corrupção de dados que possa ocorrer durante a transmissão.

3.2.2 | Payloads Específicos

O conteúdo do payload varia consoante o tipo de mensagem, seguindo sempre um formato binário compacto e bem definido.

Payload MISSION (19 bytes):

Tabela 2 : Estrutura do payload MISSION

Campo	Tipo	Tamanho	Descrição
`mission_id`	uint8	1 byte	Tipo de missão (1-6)
`task_number`	uint16	2 bytes	Identificador único da tarefa
`x`	float32	4 bytes	Coordenada X do destino
`y`	float32	4 bytes	Coordenada Y do destino
`radius`	float32	4 bytes	Raio de atuação em torno do destino
`duracao`	float32	4 bytes	Duração estimada em segundos

Payload PROGRESS (14 bytes):

Tabela 3 : Estrutura do payload PROGRESS

Campo	Tipo	Tamanho	Descrição
`mission_id`	uint8	1 byte	Identificador da missão em curso
`status`	uint8	1 byte	Estado (0=em curso, 1=pausado, 2=concluído, 3=erro)
`percent`	uint8	1 byte	Percentagem de conclusão (0-100)
`battery`	uint8	1 byte	Nível de bateria (0-100)
`x`	float32	4 bytes	Posição atual X
`y`	float32	4 bytes	Posição atual Y

Payload DONE (2 bytes):

Tabela 4 : Estrutura do payload DONE

Campo	Tipo	Tamanho	Descrição
`mission_id`	uint8	1 byte	Identificador da missão concluída
`result_code`	uint8	1 byte	Código de resultado (0=OK, 1=ABORT, 2=ERROR)

As mensagens `READY`, `NOMISSION` e `ACK` não transportam payload, utilizando apenas o header para comunicar a sua semântica.

3.3 | Tipos de Mensagens e Semântica

O protocolo MissionLink define seis tipos de mensagens, cada uma com uma função específica no ciclo de vida de uma missão.

READY (`msg_type=0`): Enviada pelo rover para sinalizar disponibilidade para receber uma nova missão. Esta mensagem não contém payload e serve como

pedido explícito de trabalho. A Nave-Mãe interpreta a receção de um READY como indicação de que o rover concluiu qualquer missão anterior e está pronto para iniciar uma nova tarefa.

MISSION (`msg_type=1`): Resposta da Nave-Mãe a um pedido READY, contendo todos os parâmetros necessários para a execução da missão. O payload inclui a identificação da tarefa, as coordenadas de destino, o raio de atuação e a duração estimada. A flag `NEEDS_ACK` está sempre ativa, exigindo confirmação explícita por parte do rover.

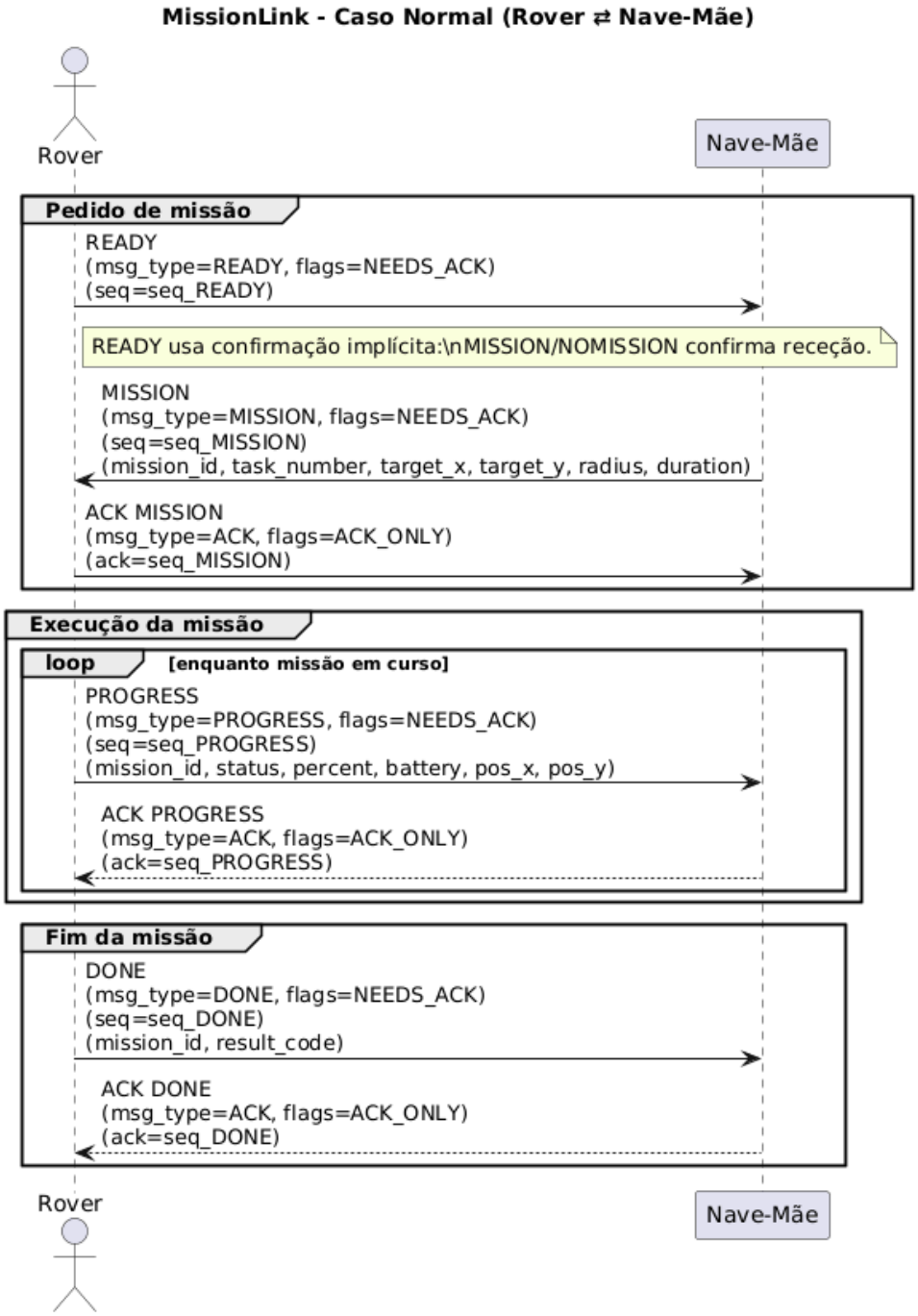
PROGRESS (`msg_type=2`): Enviada periodicamente pelo rover durante a execução da missão, reportando o progresso atual, a posição e o estado da bateria. A frequência de envio é determinada pela lógica interna do rover, mas tipicamente ocorre a cada 0.3 segundos. Cada mensagem PROGRESS requer confirmação da Nave-Mãe.

DONE (`msg_type=3`): Sinaliza a conclusão da missão por parte do rover, incluindo um código de resultado que indica sucesso, aborto ou erro. Esta mensagem marca o fim do ciclo de uma missão e deve ser confirmada pela Nave-Mãe antes do rover solicitar nova tarefa.

NOMISSION (`msg_type=5`): Resposta alternativa da Nave-Mãe a um pedido READY, indicando que não existem missões disponíveis no momento. O rover deve aguardar um período antes de voltar a solicitar trabalho, evitando sobrecarga do canal de comunicação.

ACK (`msg_type=4`): Mensagem de confirmação pura, sem payload, utilizada para confirmar a receção bem-sucedida de mensagens que possuem a flag `NEEDS_ACK` ativa. O campo `ack` no header contém o número de sequência da mensagem sendo confirmada.

3.4 | Fluxo Normal do Protocolo



O ciclo de vida completo de uma missão segue uma sequência bem definida de trocas de mensagens, ilustrada no diagrama de sequência apresentado anteriormente. O fluxo começa sempre com o rover a manifestar disponibilidade e termina com a confirmação da conclusão da missão.

Inicialmente, o rover envia uma mensagem **READY** para a Nave-Mãe. A Nave-Mãe, ao receber o READY, verifica se existem missões disponíveis para atribuir. Caso exista trabalho pendente, a Nave-Mãe responde com uma mensagem **MISSION** contendo todos os parâmetros da tarefa. Esta resposta funciona simultaneamente como confirmação implícita do READY recebido, eliminando a necessidade de um ACK dedicado para o pedido inicial.

O rover, ao receber a MISSION, valida o checksum e processa os parâmetros. Envia então um **ACK** explícito para confirmar a receção, permitindo que a Nave-Mãe marque a missão como efetivamente atribuída e remova-a da sua fila de trabalho pendente. Apenas após este ACK é que a Nave-Mãe considera a transação completa.

Durante a execução da missão, o rover entra num ciclo de atualização periódica. A cada intervalo (tipicamente 0.3 segundos), envia uma mensagem **PROGRESS** reportando a percentagem de conclusão, o estado atual, o nível de bateria e a posição corrente. Cada PROGRESS contém a flag `NEEDS_ACK` e aguarda confirmação da Nave-Mãe. Este mecanismo garante que a Nave-Mãe tem uma visão consistente do progresso da missão, mesmo em condições de rede adversas.

A Nave-Mãe, ao receber cada PROGRESS, valida a mensagem e responde com um **ACK**. Este ciclo repete-se até que o rover conclua a missão, momento em que envia uma mensagem **DONE** com o código de resultado apropriado. A Nave-Mãe confirma a receção do DONE e o rover retorna ao estado inicial, pronto para enviar um novo READY e receber outra missão.

No caso de não existirem missões disponíveis, a Nave-Mãe responde ao READY com **NOMISSION**. O rover confirma esta resposta com um ACK e aguarda um período de backoff (tipicamente 2 segundos) antes de voltar a enviar READY, evitando assim sobrecarregar o canal com pedidos desnecessários.

3.5 | Mecanismos de Fiabilidade

A utilização de UDP como protocolo de transporte impõe a necessidade de implementar mecanismos robustos de fiabilidade a nível aplicacional. O MissionLink incorpora diversas técnicas que, em conjunto, garantem a entrega fiável das mensagens críticas mesmo em cenários com perda de pacotes, duplicação ou reordenação.

3.5.1 | Numeração de Sequência

Cada mensagem enviada por qualquer entidade (rover ou Nave-Mãe) possui um número de sequência único e crescente. Este contador é mantido localmente por cada entidade e incrementado após cada envio. O número de sequência permite ao recetor processar mensagens de forma monotónica, distinguir mensagens novas de retransmissões e detetar potenciais perdas de pacotes.

A implementação utiliza contadores de 32 bits (`uint32`), proporcionando um espaço de numeração suficientemente amplo para evitar wraparound em qualquer cenário realista de operação. O método `_next_ml_seq()` encapsula a lógica de incremento, garantindo que cada mensagem recebe um identificador único.

3.5.2 | Confirmações Explícitas (ACK)

Mensagens críticas que requerem garantia de entrega são marcadas com a flag `NEEDS_ACK` no header. Ao receber uma mensagem com esta flag, o destinatário deve responder com um ACK explícito contendo o número de sequência da mensagem confirmada no campo `ack` do header.

Este mecanismo cria um protocolo de confirmação positiva explícita, onde o emissor só considera uma mensagem como entregue após receber o ACK correspondente. A ausência de ACK dentro de um timeout pré-definido desencadeia uma retransmissão.

3.5.3 | Timeouts e Retransmissões

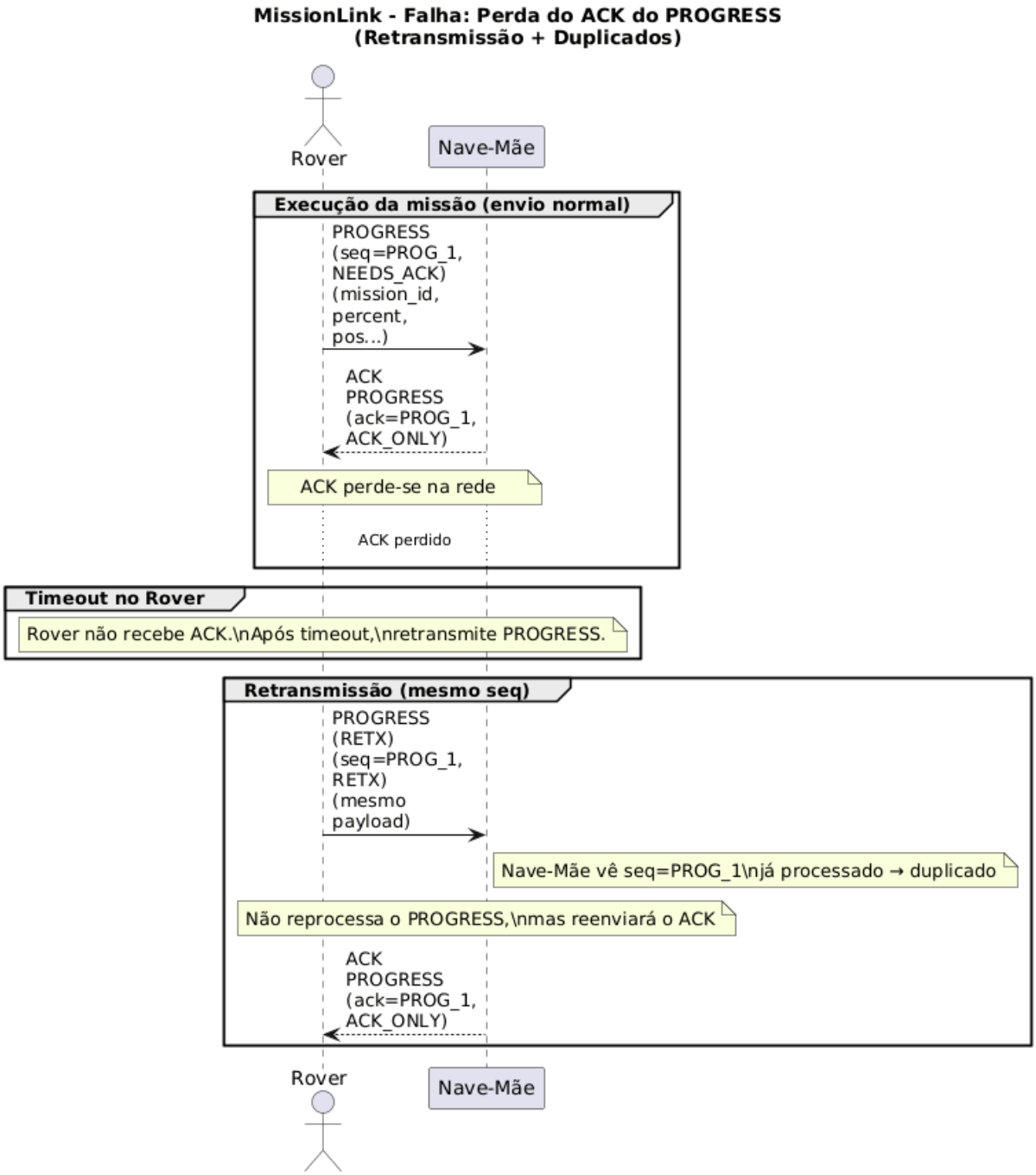
O método `send_reliable()` implementa um mecanismo sofisticado de retransmissão com timeout adaptativo. Quando uma mensagem requer confirmação, o emissor aguarda um ACK durante um período configurável (por defeito 0.5 segundos). Caso este timeout expire sem receção do ACK esperado, a mensagem é retransmitida com a flag `RETX` ativa, sinalizando ao destinatário que se trata de uma retransmissão.

O processo repete-se até um máximo de 5 tentativas (`max_retries`). Se após todas as tentativas não houver confirmação, o método retorna falha, permitindo à camada superior decidir como proceder. Esta abordagem evita bloqueios indefinidos e permite deteção de falhas persistentes de comunicação.

```
def send_reliable(self, msg_bytes: bytes, seq: int, addr,
                  timeout: float = 0.5, max_retries: int = 5) -> bool:
    retries = 0
    while retries <= max_retries and not self.eventoParar.is_set():
        self.ml_sock.sendto(msg_bytes, addr)
```

```
        try:
            data, _ = self.ml_sock.recvfrom(4096)
            h_ack, _ = ml.parse_message(data)
            if h_ack.msg_type == ml.TYPE_ACK and h_ack.ack == seq:
                return True
        except socket.timeout:
            retries += 1
            # Marca RETX para próxima tentativa
```

3.5.4 | Deteção de Duplicados



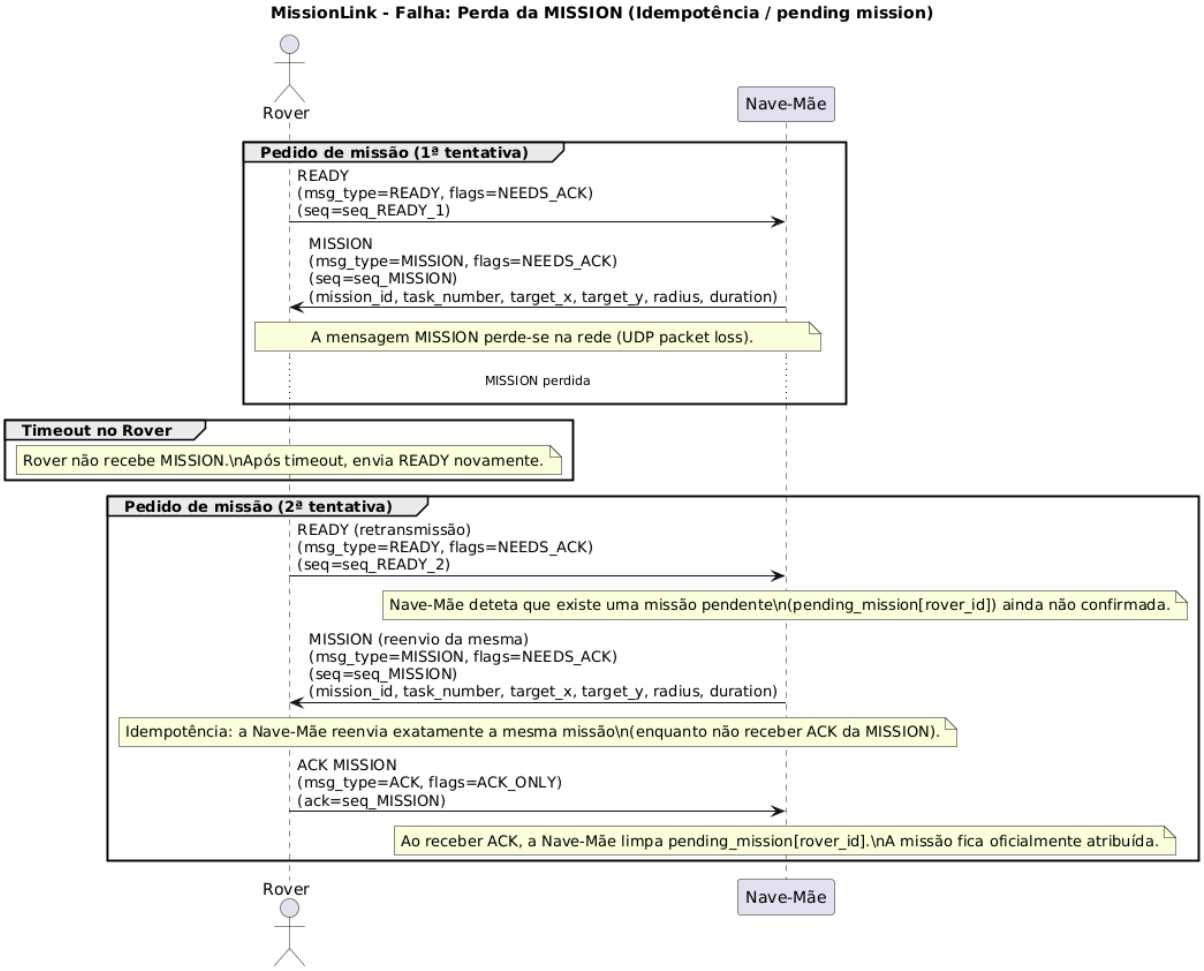
A receção de mensagens duplicadas é uma consequência natural da retransmissão em redes com perda variável. O MissionLink implementa deteção de duplicados através do método `_ml_is_duplicate()`, que mantém registo do último número de sequência recebido de cada rover.

Quando uma mensagem chega, o seu número de sequência é comparado com o último valor conhecido. Se for igual ou inferior, a mensagem é classificada como

duplicado e processada adequadamente: o ACK é reenviado (para evitar que o emissor continue a retransmitir), mas o payload não é reprocessado, garantindo semântica *exatamente-uma-vez* para operações críticas.

```
def _ml_is_duplicate(self, stream_id: int, header: ml.MLHeader) -> bool:
    last = self.ml_last_seq.get(stream_id)
    if last is None:
        self.ml_last_seq[stream_id] = header.seq
        return False
    if header.seq > last:
        self.ml_last_seq[stream_id] = header.seq
        return False
    return True  # seq <= last -> duplicado
```

3.6 | Repetição Segura na Atribuição de Missões



Um dos aspetos mais críticos do protocolo MissionLink é a garantia de que cada missão é atribuída exatamente uma vez a cada rover, mesmo face a retransmissões e duplicados. Esta propriedade é essencial para evitar situações onde um rover

recebe múltiplas cópias da mesma missão ou, pior ainda, "salta" missões devido a problemas de sincronização.

A implementação utiliza o conceito de **missão pendente**, mantido no dicionário `ml_pending_mission` na Nave-Mãe. Quando a Nave-Mãe decide atribuir uma missão em resposta a um READY, serializa a mensagem MISSION completa e armazena-a em memória, associada ao `stream_id` do rover. Esta cópia permanece guardada até que a Nave-Mãe receba o ACK correspondente.

Se o rover enviar um novo READY antes de ter confirmado a missão anterior (por exemplo, devido a perda do pacote MISSION original), a Nave-Mãe não seleciona uma nova missão da fila. Em vez disso, reenvia a **mesma** mensagem MISSION previamente serializada, incluindo o mesmo número de sequência. Este comportamento idempotente garante que retransmissões não causam atribuição duplicada.

```
def _ml_handle_ready(self, stream_id: int, header: ml.MLHeader, addr):
    # Verifica se já existe resposta pendente
    pending = self.ml_pending_mission.get(stream_id)
    if pending is not None:
        # Reenvia a MESMA resposta (idempotente)
        self.ml_sock.sendto(pending["reply_bytes"], addr)
        return

    # ... lógica de seleção de missão ...

    # Guarda resposta como pendente
    self.ml_pending_mission[stream_id] = {
        "mission_seq": mission_seq,
        "reply_bytes": msg,
        "missao": missao,
    }
```

Apenas quando o ACK chega é que a entrada correspondente é removida de `ml_pending_mission`, sinalizando que a transação foi completada com sucesso. Nesse momento, e só nesse momento, a missão é consumida da fila de trabalho (no caso de missões automáticas dos cenários 2 e 4) ou removida da fila de missões manuais enviadas pelo Ground Control.

Esta abordagem implementa um handshake de confirmação robusto: a missão só é considerada definitivamente atribuída após o ACK do rover, eliminando janelas de inconsistência entre as duas entidades.

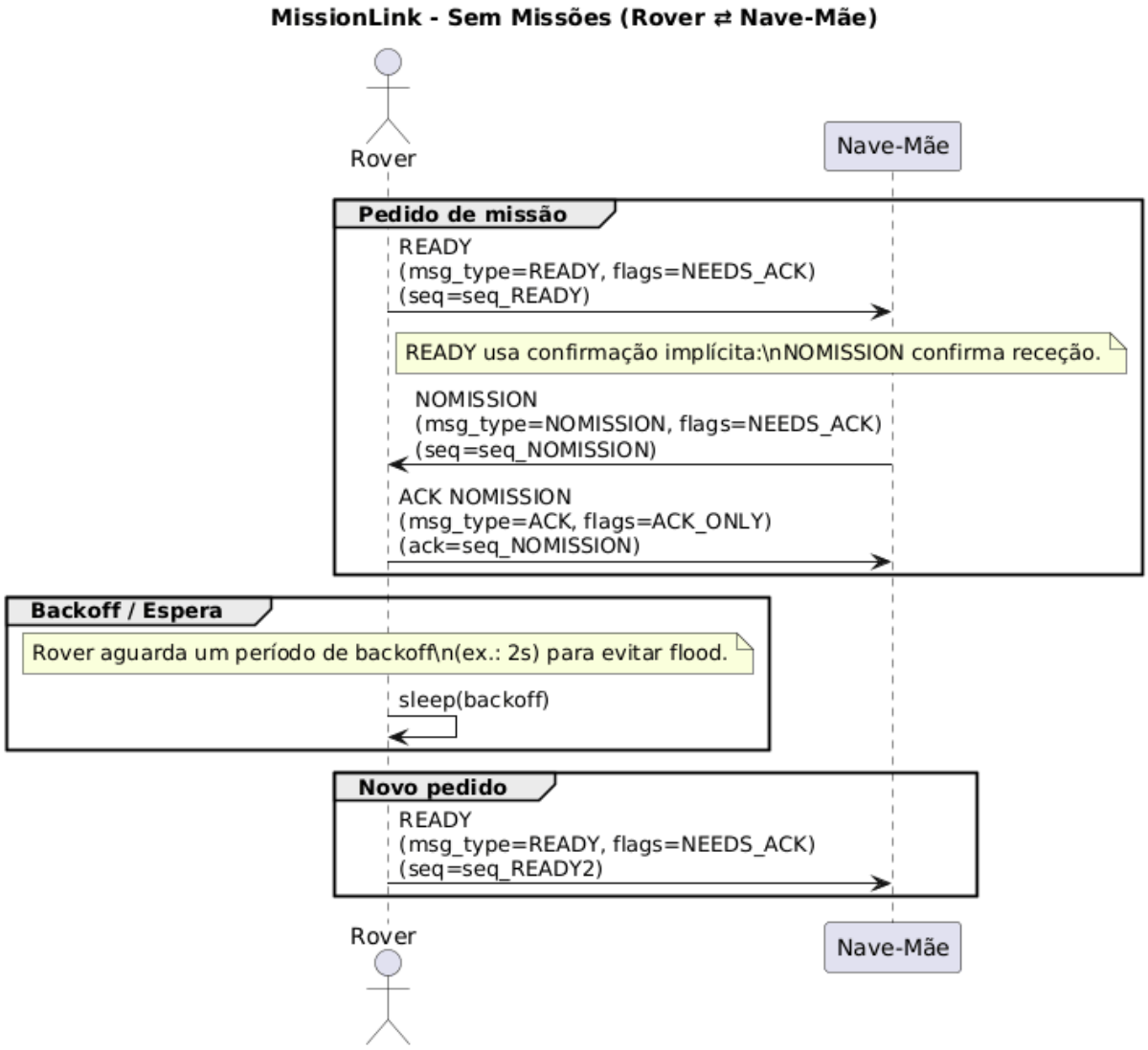
3.7 | READY com Confirmação Implícita

Uma particularidade interessante do protocolo é que a mensagem READY não recebe um ACK dedicado. Em vez disso, a própria resposta MISSION ou NOMISSION funciona como confirmação implícita do READY recebido.

Esta decisão de design baseia-se na observação de que um ACK puro não acrescentaria informação útil. O rover, ao enviar READY, está efetivamente a fazer uma pergunta: "Existe trabalho disponível?". A resposta a esta pergunta é necessariamente MISSION ou NOMISSION, e qualquer uma destas mensagens confirma implicitamente que o READY foi recebido e processado.

Opcionalmente, a Nave-Mãe pode refletir o `seq` do READY no campo `ack` da resposta para facilitar debugging e rastreabilidade, embora esta prática não seja estritamente necessária para o funcionamento correto do protocolo. Esta abordagem reduz o número de mensagens trocadas sem comprometer a fiabilidade, uma vez que ambas as respostas (MISSION e NOMISSION) requerem confirmação explícita por parte do rover através de um ACK dedicado.

3.8 | Ausência de Trabalho (NOMISSION)



Quando a Nave-Mãe não possui missões disponíveis para atribuir, responde ao READY com NOMISSION. Esta situação pode ocorrer nos cenários 1 e 3 (antes de haver tarefas geradas) ou em qualquer cenário se todos os rovers estiverem simultaneamente ocupados e não houver missões suficientes na fila.

O rover, ao receber NOMISSION, confirma a mensagem com um ACK e entra num período de espera (*backoff*) antes de voltar a solicitar trabalho. A implementação atual utiliza um intervalo fixo de 2 segundos, suficiente para evitar flood do canal mas curto o bastante para manter boa responsividade quando novas missões se tornam disponíveis.

Este mecanismo de backoff é essencial para prevenir que rovers ociosos sobrecarreguem a Nave-Mãe com pedidos READY contínuos. Num cenário com

múltiplos rovers em estado de espera, a ausência de backoff resultaria numa tempestade de pacotes que degradaria o desempenho geral do sistema, especialmente considerando que UDP não oferece controlo de congestão nativo.

```
if header.msg_type == ml.TYPE_NOMISSION:
    print(f"[Rover {self.rover.id}] Nave-Mae sem missao. "
          "Vou esperar e tentar de novo.")
    # ACK da NOMISSION
    # ...
    time.sleep(2.0)  # Backoff de 2 segundos
    continue
```

3.9 | Tratamento de Mensagens Fora de Contexto

Para além dos fluxos normais de operação, o protocolo define comportamentos específicos para situações anómalas onde mensagens chegam fora do contexto esperado. Esta robustez adicional previne efeitos colaterais indesejados e facilita a recuperação de estados inconsistentes.

Mensagens do tipo **PROGRESS** ou **DONE** que referenciam uma missão não correspondente à atualmente ativa para aquele rover são reconhecidas através de ACK (para travar retransmissões do lado do emissor), mas ignoradas do ponto de vista lógico. Esta política evita que mensagens atrasadas ou duplicadas de missões anteriores interfiram com o estado atual.

Quando a Nave-Mãe recebe um **ACK** que não corresponde a nenhuma mensagem pendente conhecida, simplesmente descarta a confirmação. Esta situação pode ocorrer se um ACK for duplicado ou chegar após a janela de timeout já ter expirado e a retransmissão ter sido bem-sucedida por outra via.

Do lado do rover, a receção de uma mensagem **MISSION** duplicada (detetada através de comparação do número de sequência com mensagens já processadas) resulta no reenvio do ACK correspondente, mas sem reprocessamento dos parâmetros da missão. Isto garante que o rover não reinicia uma missão já em curso devido a problemas de rede.

Estes mecanismos defensivos aumentam significativamente a resiliência do sistema face a condições adversas de rede, garantindo que estados temporariamente inconsistentes convergem rapidamente para uma visão coerente partilhada entre rover e Nave-Mãe.

3.10 | Garantias Formais do Protocolo

A combinação dos mecanismos descritos permite ao MissionLink oferecer garantias formais sobre a comunicação, essenciais para a fiabilidade do sistema como um todo.

Entrega Fiável sob Conectividade Eventual: Toda mensagem marcada com `NEEDS_ACK` é entregue com sucesso ou, em caso de falha persistente que exceda o limite de retransmissões, reportada como não entregue. Esta garantia pressupõe conectividade eventual dentro da janela de timeout $\times$ tentativas máximas.

Processamento Exatamente-Uma-Vez: No contexto da atribuição de missões, o protocolo garante que cada rover recebe e processa cada missão exatamente uma vez. Retransmissões são detetadas através do mecanismo de numeração de sequência e tratadas adequadamente, não resultando em reprocessamento.

Consistência Rover $\leftrightarrow$ Nave-Mãe: Ambas as entidades mantêm uma visão consistente do estado da missão através do handshake de confirmação. Quando a Nave-Mãe considera uma missão como atribuída (após receber ACK), o rover efetivamente a processou. Quando o rover reporta conclusão através de DONE confirmado, a Nave-Mãe reconhece essa conclusão.

Processamento Único: Através da rejeição de mensagens com números de sequência já processados, o sistema garante que não há "retrocessos" no estado lógico. Uma vez processada uma mensagem com seq=N, mensagens com seq $\leq$ N do mesmo emissor são tratadas como duplicados.

Resiliência a Falhas Transitórias: Perdas ocasionais de pacotes, reordenações ou duplicações na rede não causam inconsistências permanentes. O sistema recupera automaticamente através de retransmissões e deteção de duplicados, desde que a conectividade seja eventualmente restaurada dentro do limite de tentativas.

Estas garantias são particularmente relevantes num contexto espacial simulado, onde latências elevadas e perdas de pacotes são esperadas devido à natureza da comunicação via satélites intermédios.

3.11 | Tratamento de Casos Extremos

Apesar dos mecanismos de fiabilidade implementados, existem situações extremas onde a recuperação automática não é possível ou desejável. O protocolo foi

desenhado para lidar com estes casos de forma controlada e previsível, evitando bloqueios indefinidos ou degradação silenciosa.

3.11.1 | Falha Persistente de Comunicação

Se após `max_retries` tentativas (5 por defeito) não houver resposta, o método `send_reliable()` retorna `False`, sinalizando falha definitiva. Esta situação indica uma interrupção prolongada da conectividade, falha permanente do destinatário, ou condições de rede tão adversas que excedem a capacidade de recuperação automática do protocolo.

No contexto do rover, uma falha ao enviar PROGRESS ou DONE resulta na interrupção do ciclo de atualização. O rover não reenvia indefinidamente, evitando acumulação de mensagens e consumo desnecessário de recursos. A lógica de tratamento de erro pode então decidir abortar a missão atual e tentar restabelecer comunicação através de um novo ciclo começando por READY.

3.11.2 | Abandono Controlado e Não-Bloqueio

O sistema evita bloqueios indefinidos através de timeouts bem definidos em todas as operações de I/O. O socket UDP está configurado com timeout de 0.5 segundos, garantindo que nenhuma operação de receção bloqueia eternamente, mesmo na ausência total de tráfego de rede.

Adicionalmente, todas as threads de comunicação verificam periodicamente a flag `eventoParar`, permitindo encerramento limpo do sistema mesmo se existirem operações pendentes. Esta abordagem garante que o sistema pode ser terminado de forma controlada em qualquer estado, sem deixar recursos por libertar ou threads órfãs em execução.

3.11.3 | Limite de Tentativas e Compromisso Latência-Fiabilidade

A escolha de 5 tentativas como máximo representa um compromisso cuidadoso entre fiabilidade e latência. Com um timeout de 0.5 segundos por tentativa, o tempo máximo de espera antes de declarar falha definitiva é de aproximadamente 2.5 segundos. Este valor é suficiente para tolerar rajadas temporárias de perda de pacotes (comuns em redes congestionadas ou em transições de rotas) sem introduzir atrasos excessivos que bloqueariam o sistema em caso de falha real.

Em cenários com latências particularmente elevadas (centenas de milissegundos até segundos, comuns em comunicação espacial real com distâncias interplanetárias), estes parâmetros poderiam ser ajustados através de configuração. No entanto, para a topologia simulada no CORE com latências típicas de dezenas

de milissegundos e ocasionais picos devido à malha de satélites, os valores configurados demonstraram-se adequados e eficazes nos testes realizados.

A implementação mantém estes valores como constantes facilmente modificáveis, permitindo tuning futuro sem alterações estruturais ao código, caso o sistema seja adaptado para outras condições de rede ou requisitos operacionais distintos.

4 | TS (Telemetry System)

No nosso sistema, o envio de telemetria é feito pelo protocolo TelemetryStream (TS) sobre TCP, onde cada rover atua como cliente e a Nave-Mãe como servidor. Este protocolo garante uma ligação fiável e contínua, permitindo que cada rover envie periodicamente informação sobre o seu estado operacional e garante a sua entrega.

4.1 | Estrutura da mensagem

Header 16 bytes	tipo	id_rover	bateria	pos_x	pos_y
	pos_z	state	checksum	payload_len	freq
Payload(9 bytes)	proc_use, storage, velocidade, direcao, sensores,progresso,x,y,z				

Estrutura da mensagem usado em TS

tipo	id_rover	bateria	pos_x	pos_y
1 byte	1 byte	1 byte	1 byte	1 byte
pos_z	state	checksum	payload_len	freq
1 byte	1 byte	4 bytes	4 bytes	1 byte

Tamanho de cada campo do header

Desta forma no header temos todos os campos necessários e importantes referente a cada rover, sendo que o *checksum* e o *payload_len* servem apenas para fins de garantia da qualidade da mensagem. Embora o TCP garanta já a entrega fiável dos pacotes, adicionamos estes campos para adicionar assim uma segunda camada de proteção sobre os dados enviados. O tipo apesar de não ser de um uso necessário, está implementado para possibilitar futuros updates no que diz respeito a acomodar apenas telemetria enquanto rovers operam e no fim avisar que ja acabaram ou ate como avisar do começo de uma maneira especial.

Para ja o tipo apenas é usado para notificar que um rover inciou o socket tcp e envia a primeira mensagem apenas com o tipo a 0 para a nave mãe ser notificada e a a partir desse momento passa a enviar mensagens com o tipo a 1 para enviar a telemetria. No entanto não usamos mais tipos pois a telemetria e enviada constantemente até termino da nave mãe.

Já no que diz respeito ao Payload, enviamos assim informações sobre secundárias sobre o rover mas com a sua importância para uma posterior comunicação com o Ground Control. O segundo conjunto de coordenadas enviadas na comunicação(enviadas dentro do payload) dizem respeito ao destino do rover atual.

Desta forma conseguimos enviar para a nave mãe todas as métricas necessárias para a mesma saber do estado do rover assim como os detalhes da missão.

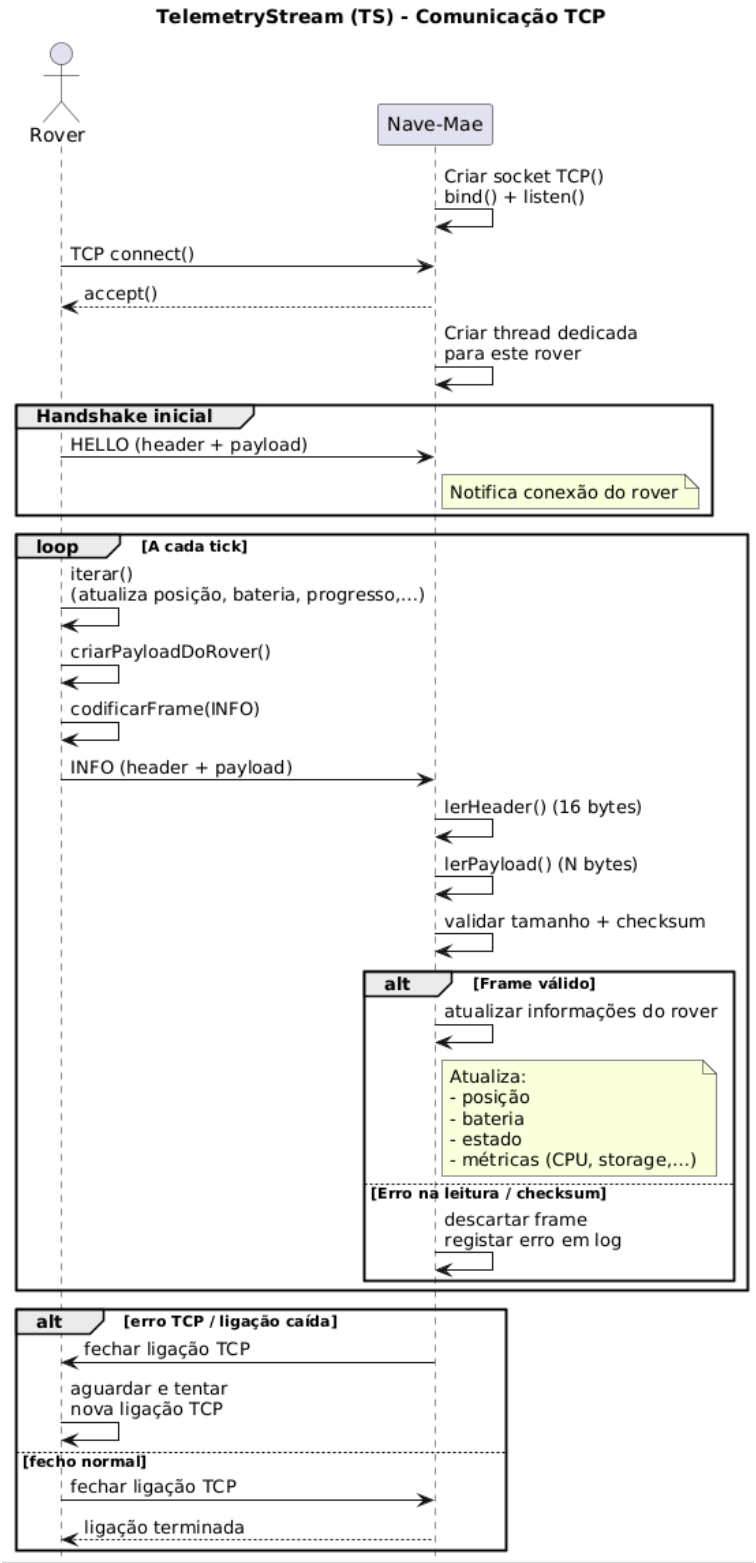
4.2 | Detalhes do protocolo

O protocolo apenas suporta números inteiros até 255.

Outro detalhe importante do protocolo é a validação estrutural das mensagens. Como cada frame tem um formato e tamanho bem definidos (header fixo e payload com comprimento indicado em *payload_len*), o recetor consegue verificar se a mensagem recebida está completa e consistente. Assim, qualquer frame incompleto, truncado ou que não respeite o tamanho/formato esperado é descartado, evitando que dados corrompidos sejam processados.

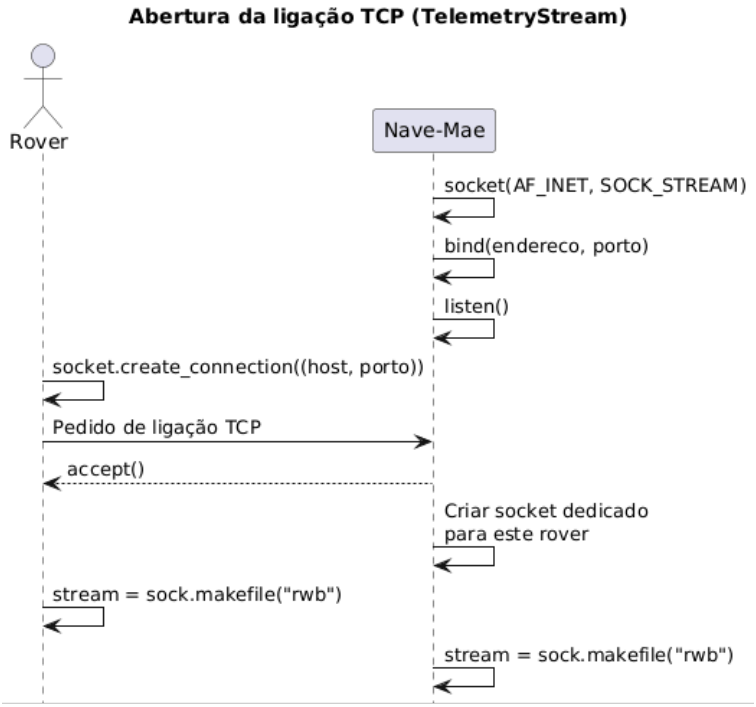
O protocolo está pronto para receber expansões, já que fazemos uso de *payload_len* e temos uma grande gama de tipos ainda por usar. Futuras expansões ao mesmo tornam-se bastante fáceis e práticas.

4.3 | Diagrama de sequencia do TS



Com este diagrama é possível observar toda a sequência de mensagens trocadas entre Rover e Nave-Mãe, assim como, tudo o que ambos os processos executam durante a troca da mensagens. Como a base da comunicação irá ser feita em TCP, a garantia de entrega é garantida. Desta forma, o desenho dese tipo de mensagem/comunicação é facilitada pelo uso do TCP.

4.4 | Implementação em python



Para implementar esta comunicação utilizámos os métodos já disponibilizados pelo módulo socket de Python, configurando a Nave Mãe como servidor TCP e cada rover como cliente. Desta forma, a Nave Mãe aceita ligações de múltiplos rovers e processa todos de forma simultânea. Foi usado um socket no modo *rwb* para implementar toda a troca de mensagens em bytes facilitando assim write e read do socket assim como uma fácil exportação/portabilidade para outra qualquer linguagem.

5 | GC (Ground Control)

5.1 | Arquitetura e Função

O Ground Control constitui a interface de visualização e controlo do sistema, atuando como cliente da Nave-Mãe num modelo cliente-servidor. A sua responsabilidade primária é consumir dados do estado atual do sistema e apresentá-los ao utilizador de forma clara e atualizada. O Ground Control não mantém a lógica ou o estado crítico do sistema, delegando todas as operações de gestão de missões e coordenação de rovers à Nave-Mãe, que atua como servidor central.

5.2 | Funcionalidades de Monitorização

A interface do Ground Control disponibiliza visualização em tempo real do estado de todos os rovers registados no sistema. Para cada rover, são apresentadas informações como identificador único, estado atual (livre, a realizar trabalho, em movimento, erro), posição atual, missão atualmente atribuída e progresso de execução. Esta informação é constantemente atualizada através da comunicação WebSocket com a Nave-Mãe, garantindo que o operador tem acesso aos dados mais recentes disponíveis. A apresentação do progresso de cada missão permite ao um operador acompanhar a execução em tempo real, identificando potenciais problemas ou atrasos. O sistema de visualização foi concebido para proporcionar uma visão global do estado operacional, facilitando a tomada de decisões e a identificação de situações que requeiram intervenção. A interface de terminal apresenta três modos de visualização principais: listagem de todas as missões atuais por rover, visualização apenas dos rovers ativos com as suas missões, e telemetria detalhada incluindo posição, bateria, estado e métricas de sistema.

5.3 | Atribuição Manual de Missões

Para além da monitorização passiva, o Ground Control implementa funcionalidade de atribuição manual de missões a rovers específicos através da opção "Missão prioridade" no menu principal.

5.3.1 | Interface de Configuração

O operador configura a missão através de prompts sequenciais que solicitam:

- **Rover destinatário:** Identificador validado entre 1 e N rovers disponíveis

- **Tipo de missão (`mission_id`):** Valores entre 1 e 6, correspondendo a tipos de missões diferentes.
- **Coordenadas de destino:** X e Y validados entre 0 e 15
- **Raio de operação:** Fixo em 2.0 unidades
- **Duração estimada:** Entre 10 e 600 segundos

5.3.2 | Transmissão ao Servidor

Após introdução dos parâmetros, o Ground Control:

1. Constrói objeto JSON com tipo `assign_mission` contendo todos os campos
2. Invoca `send_ws()` para transmissão via WebSocket
3. Utiliza `asyncio.run_coroutine_threadsafe()` para envio assíncrono
4. Aplica timeout de 2 segundos para evitar bloqueios indefinidos
5. Informa o operador sobre o sucesso ou falha da operação

5.3.3 | Processamento na Nave-Mãe

Na Nave-Mãe, o handler `_ws_msg_recebida()` processa a mensagem:

1. **Validação:** Extrai e valida campos do JSON recebido
2. **Geração de ID único:** Incrementa `manual_task_counter` para criar identificador de tarefa
3. **Enfileiramento:** Adiciona missão ao dicionário `manual_missions[rover_id]`
4. **Estrutura de dados:** Cada rover possui lista ordenada de missões pendentes

5.3.4 | Prioridade e Atribuição

O sistema implementa prioridade para missões manuais:

- Quando rover envia `READY` via protocolo MissionLink, a Nave-Mãe verifica primeiro a fila `manual_missions[rover_id]`
- Se existirem missões manuais pendentes, a primeira da fila é selecionada
- Apenas se não houver missões manuais é que o sistema considera missões automáticas
- A missão é enviada via mensagem `MISSION` do protocolo MissionLink
- Permanece pendente até receção de ACK

5.3.5 | Confirmação e Limpeza

Após confirmação bem-sucedida:

1. Nave-Mãe recebe ACK do rover confirmando receção da `MISSION`
2. Verifica se missão confirmada corresponde à primeira da fila manual

3. Remove missão da fila através de `fila_manual.pop(0)`
4. Mecanismo garante que missões não são perdidas em caso de falhas de comunicação

5.3.6 | Benefícios e Casos de Uso

Esta funcionalidade permite intervenção humana para:

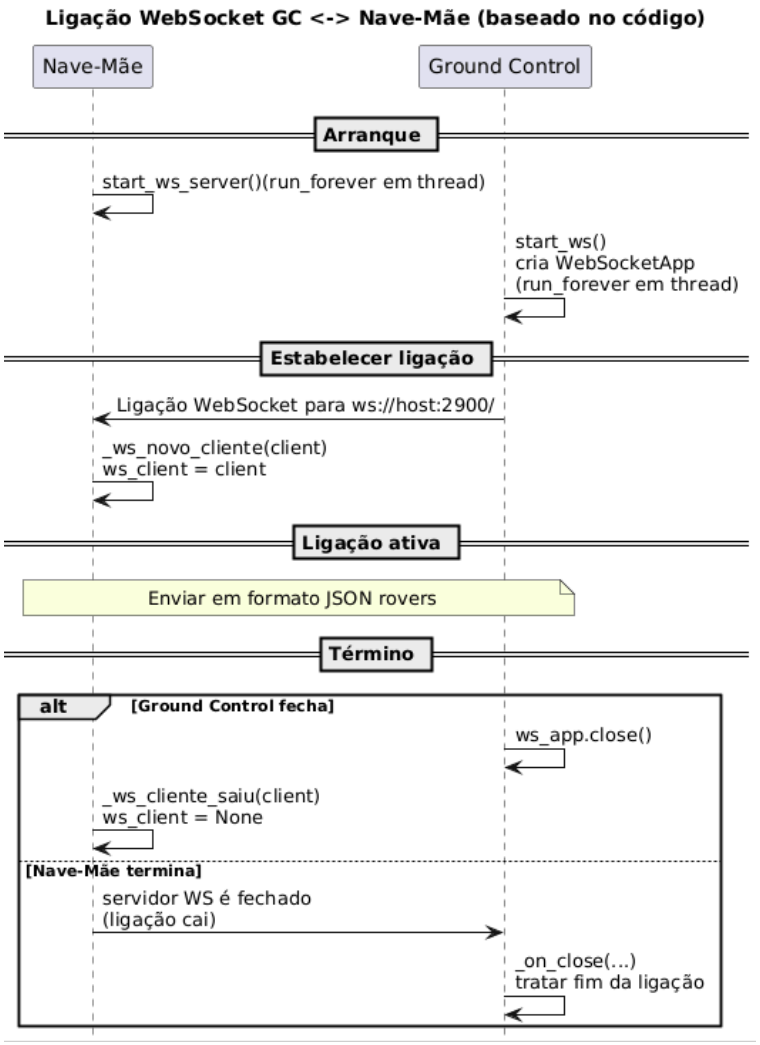
- **Testes específicos:** Validação de comportamentos em coordenadas determinadas
- **Situações excepcionais:** Controlo direto quando necessário
- **Priorização:** Sobrepõe-se ao escalonamento automático

A separação de responsabilidades garante que a consistência e fiabilidade do sistema são mantidas através dos mecanismos de confirmação e idempotência implementados na Nave-Mãe e no protocolo MissionLink, mesmo com intervenção manual.

5.4 | Modelo Cliente-Servidor

A arquitetura cliente-servidor adotada estabelece fronteiras claras de responsabilidade. A Nave-Mãe mantém o estado autoritativo do sistema, gere a fila de missões, coordena a comunicação com os rovers através do protocolo MissionLink e garante a consistência global. O Ground Control, como cliente, consome estes dados para visualização e submete pedidos de operação (como atribuição de missões) que são processados e validados pelo servidor. Esta separação permite que múltiplas instâncias do Ground Control possam conectar-se simultaneamente à mesma Nave-Mãe, cada uma fornecendo uma visão do sistema a diferentes operadores. O estado crítico permanece centralizado no servidor, evitando problemas de sincronização e inconsistência que surgiriam se o Ground Control mantivesse lógica de negócio local.

5.5 | Diagrama de conexão



Ao nível da conexão, optamos por correr o servidor WebSocket na Nave--Mãe numa thread própria. Do mesmo modo, o Ground Control (cliente), ao estabelecer a ligação, executa também o WebSocket numa thread separada, de forma a manter a ligação permanentemente ativa sem bloquear a restante lógica da aplicação.

Como se observa no diagrama, a ligação pode terminar de duas formas: por fecho inesperado do lado da nave mãe(por exemplo, encerramento do processo ou erro) ou por ação explícita do utilizador no Ground Control, que pode escolher encerrar a aplicação e, consequentemente, fechar a ligação WebSocket.

5.6 | Transferência de dados

Para o envio da informação relativa a cada rover optámos por utilizar o formato JSON, que se revela particularmente adequado por ser leve, amplamente suportado e facilmente portável para diferentes linguagens e plataformas. Este formato permite estruturar os dados de forma clara e hierárquica, facilitando a sua interpretação no Ground Control e garantindo extensibilidade futura do protocolo sem alterações significativas na infraestrutura de comunicação. De maneira a não sobrecarregar o canal de comunicação, implementámos um mecanismo de atualização incremental. A Nave-Mãe mantém uma flag dirty associada a cada rover, que indica se o estado daquele rover foi modificado desde a última transmissão. Quando esta flag está ativa, a Nave-Mãe serializa toda a informação desse rover específico para JSON e transmite-a através do WebSocket. Posteriormente, marca o rover como clean, evitando retransmissões desnecessárias de dados não modificados. Este mecanismo de dirty flags garante que apenas são enviadas atualizações relativas a rovers cujo estado efetivamente mudou, reduzindo significativamente o tráfego de rede e otimizando a utilização da largura de banda disponível. Em cenários com múltiplos rovers, onde apenas alguns estão ativamente a executar missões, esta otimização torna-se particularmente relevante.

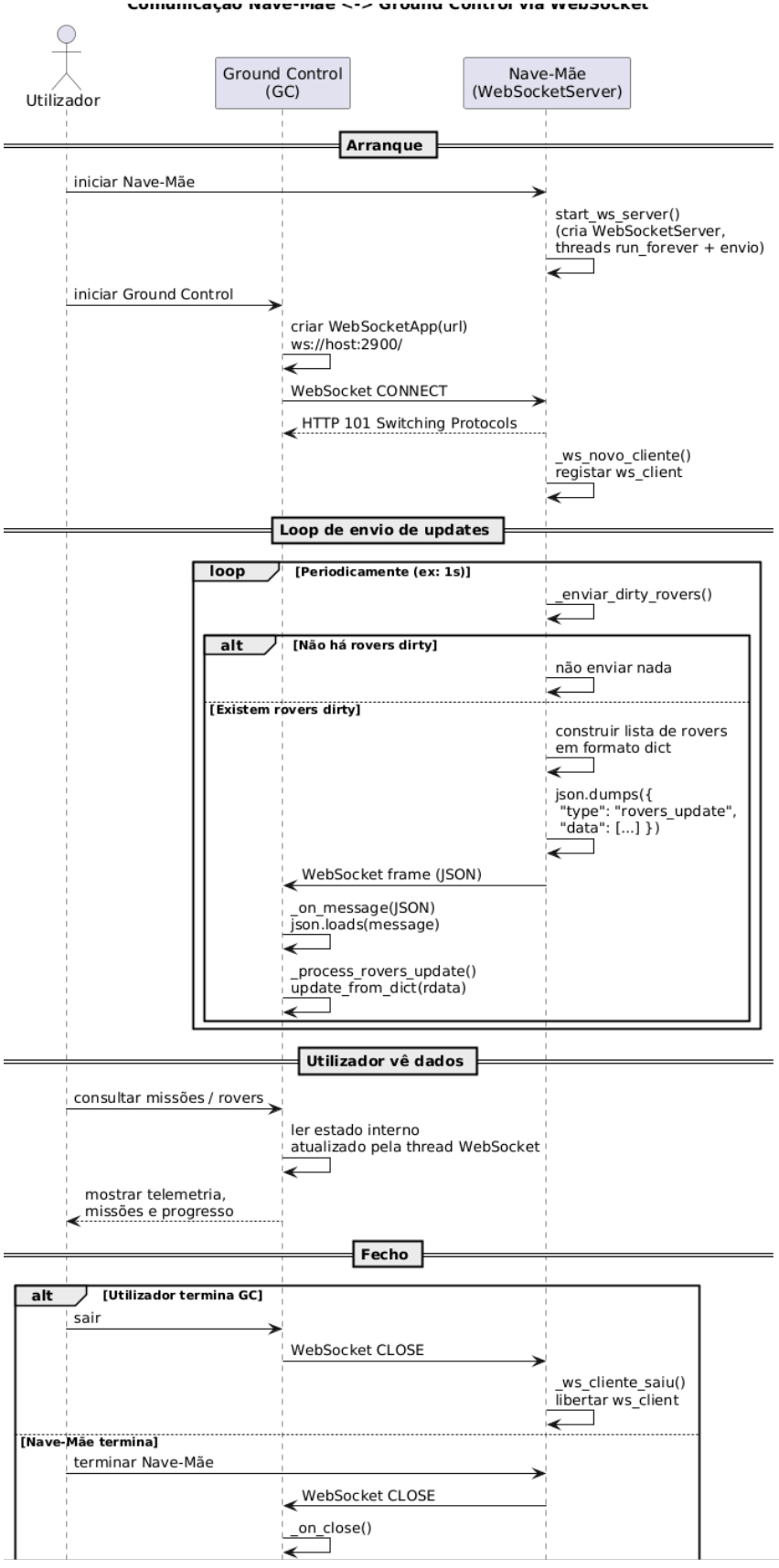
5.7 | Implementação em Python

A implementação em Python da comunicação entre Nave-Mãe e Ground Control foi desenvolvida com foco em modularidade e reutilização de código. Para suportar a serialização e desserialização eficiente de dados, desenvolvemos uma biblioteca dedicada integrada na API do rover, capaz de realizar transformações bidirecionais entre objetos Rover e representações JSON, bem como aplicar atualizações parciais através de uma função updateRover(). Esta biblioteca de serialização abstrai a complexidade da conversão de dados, permitindo que qualquer cliente com acesso à mesma possa facilmente processar e interpretar as informações recebidas. A abordagem de biblioteca partilhada garante consistência no formato de dados entre diferentes componentes do sistema e facilita a manutenção, uma vez que alterações no esquema de dados apenas necessitam ser implementadas num único local. A execução do WebSocket em threads dedicadas tanto na Nave-Mãe como no Ground Control segue o padrão de design de separação de responsabilidades. A thread de comunicação gere exclusivamente o envio e receção de mensagens, enquanto a thread principal de cada aplicação processa a lógica de negócio. Esta arquitetura evita bloqueios na interface do

utilizador e garante que a comunicação permanece responsiva mesmo durante operações computacionalmente intensivas. Para gestão de concorrência e sincronização entre a thread de comunicação e a thread principal, foram implementados mecanismos apropriados de locks e filas thread-safe, garantindo que atualizações de estado são aplicadas de forma atómica e consistente. Esta abordagem previne race conditions e garante a integridade dos dados visualizados no Ground Control.

5.8 | Segurança e Robustez

A implementação inclui tratamento robusto de erros de conexão, com lógica de reconexão automática no Ground Control caso a ligação com a Nave-Mãe seja interrompida inesperadamente. A validação de dados recebidos é realizada em ambos os lados da comunicação, garantindo que mensagens malformadas ou inválidas são rejeitadas antes de serem processadas. Esta camada de validação adicional protege contra potenciais problemas de corrupção de dados durante a transmissão e facilita debugging durante o desenvolvimento e manutenção do sistema


Comunicação Nave-Mãe <-> Ground Control via WebSocket
Utilizador
Ground Control (GC)
Nave-Mãe (WebSocketServer)
Arranque
iniciar Nave-Mãe
start_ws_server()
(cria WebSocketServer,
threads run_forever + envio)
iniciar Ground Control
criar WebSocketApp(url)
ws://host:2900/
WebSocket CONNECT
HTTP 101 Switching Protocols
_ws_novo_cliente()
registar ws_client
Loop de envio de updates
loop
[Periodicamente (ex: 1s)]
_enviar_dirty_rovers()
alt
[Não há rovers dirty]
não enviar nada
[Existem rovers dirty]
construir lista de rovers
em formato dict
json.dumps({
"type": "rovers_update",
"data": [...] })
WebSocket frame (JSON)
_on_message(JSON)
json.loads(message)
_process_rovers_update()
update_from_dict(rdata)
Utilizador vê dados
consultar missões / rovers
ler estado interno
atualizado pela thread WebSocket
mostrar telemetria,
missões e progresso
Fecho
alt
[Utilizador termina GC]
sair
WebSocket CLOSE
_ws_cliente_saiu()
libertar ws_client
[Nave-Mãe termina]
terminar Nave-Mãe
WebSocket CLOSE
_on_close()

6 | Inicialização da Nave-Mãe e Gestão de Cenários

A Nave-Mãe é responsável pela orquestração global do sistema, atuando como entidade central de coordenação entre os rovers e o Ground Control. Para permitir a avaliação de diferentes comportamentos do sistema distribuído, a Nave-Mãe suporta múltiplos cenários de execução, configuráveis no momento da inicialização.

6.0.1 | Processo de Inicialização da Nave-Mãe

No arranque, a Nave-Mãe executa as seguintes etapas fundamentais:

1. Inicialização das estruturas de estado internas, incluindo:
   - tabelas de sequência (`ml_last_seq`) para deteção de duplicados;
   - missões pendentes por rover (`ml_pending_mission`);
   - filas de missões automáticas e manuais;
2. Criação e configuração do socket UDP associado ao MissionLink;
3. Inicialização do servidor TCP do TelemetryStream;
4. Inicialização do serviço de comunicação com o Ground Control;
5. Carregamento do cenário de execução selecionado.

O cenário ativo é definido através de parâmetros de configuração fornecidos no arranque da Nave-Mãe, permitindo selecionar explicitamente o comportamento desejado sem alterações ao código-fonte.

6.0.2 | Definição de Cenários

Os cenários implementados representam diferentes condições operacionais do sistema e são utilizados tanto para testes funcionais como para avaliação do comportamento do protocolo sob diferentes cargas de trabalho.

6.0.2.1 | Cenário 1 — Sistema sem Missões

No cenário 1, a Nave-Mãe é inicializada sem qualquer missão disponível para atribuição. Este cenário tem como objetivo testar o comportamento do sistema na ausência total de trabalho.

Sempre que um rover envia uma mensagem **READY**, a Nave-Mãe responde com **NOMISSION**. O rover confirma a resposta através de um ACK e entra num período de backoff antes de reenviar **READY**. Este cenário valida:

- o tratamento correto de ausência de trabalho;
- a eficácia do mecanismo de backoff;
- a estabilidade do sistema sob pedidos repetidos sem progresso.

6.0.2.2 | Cenário 2 — Conjunto Finito de Missões

No cenário 2, a Nave-Mãe é inicializada com uma lista finita de missões pré-definidas. Estas missões são armazenadas numa fila interna e atribuídas sequencialmente aos rovers à medida que estes enviam pedidos **READY**.

Cada missão é removida da fila apenas após a confirmação explícita da sua atribuição (ACK da mensagem **MISSION**), garantindo que nenhuma missão é perdida devido a falhas de comunicação. Quando a fila de missões se esgota, a Nave-Mãe passa a responder com **NOMISSION** a pedidos subsequentes.

Este cenário permite avaliar:

- atribuição correta de missões únicas;
- consumo ordenado de tarefas;
- comportamento do sistema quando o trabalho termina.

6.0.2.3 | Cenário 3 — Geração Dinâmica de Missões

No cenário 3, a Nave-Mãe não possui uma lista fixa de missões. Em vez disso, novas missões são geradas dinamicamente sempre que um rover solicita trabalho e não existe nenhuma missão pendente para esse rover.

Este cenário simula um ambiente com carga contínua e potencialmente infinita, permitindo avaliar o comportamento do MissionLink sob condições de operação prolongada. Cada missão gerada recebe um identificador único e parâmetros distintos, garantindo que missões consecutivas podem ser diferenciadas e rastreadas corretamente.

O cenário valida:

- robustez do protocolo em execução contínua;
- correta gestão de sequência e estado ao longo do tempo;
- ausência de degradação do sistema em execuções prolongadas.

6.0.2.4 | Cenário 4 — Missões Manuais via Ground Control

No cenário 4, a Nave-Mãe aceita missões manuais enviadas pelo Ground Control. Estas missões são colocadas numa fila dedicada e têm prioridade sobre missões automáticas.

Quando um rover envia **READY**, a Nave-Mãe verifica primeiro a existência de missões manuais pendentes. Caso existam, atribui a missão manual ao rover. Apenas na ausência de missões manuais é que a Nave-Mãe recorre à lógica de geração automática de missões.

Este cenário permite:

- interação direta do utilizador com o sistema;
- validação da integração entre Ground Control e MissionLink;
- teste de coexistência entre missões manuais e automáticas.

6.0.3 | Interação entre Cenários e MissionLink

Independentemente do cenário ativo, a atribuição de missões segue sempre as mesmas regras de fiabilidade definidas pelo MissionLink. Em particular:

- nenhuma missão é considerada atribuída sem confirmação explícita;
- missões pendentes são reenviadas de forma segura em caso de retransmissão;
- o comportamento do protocolo é consistente entre cenários.

Esta separação entre **política de geração de missões** (cenários) e **mecanismo de comunicação** (MissionLink) simplifica o desenho do sistema e garante que alterações nos cenários não afetam a correção do protocolo de fiabilidade.

7 | Topologia

A infraestrutura de comunicação do sistema foi desenhada para simular um ambiente distribuído realista, refletindo a separação física e lógica entre o centro de controlo, a nave central e os rovers no terreno. A topologia adotada permite testar o comportamento dos protocolos desenvolvidos em cenários com múltiplos saltos de rede, latência adicional e potencial perda de pacotes.

A Figura ilustra a topologia completa utilizada nos testes, implementada no ambiente de simulação CORE.

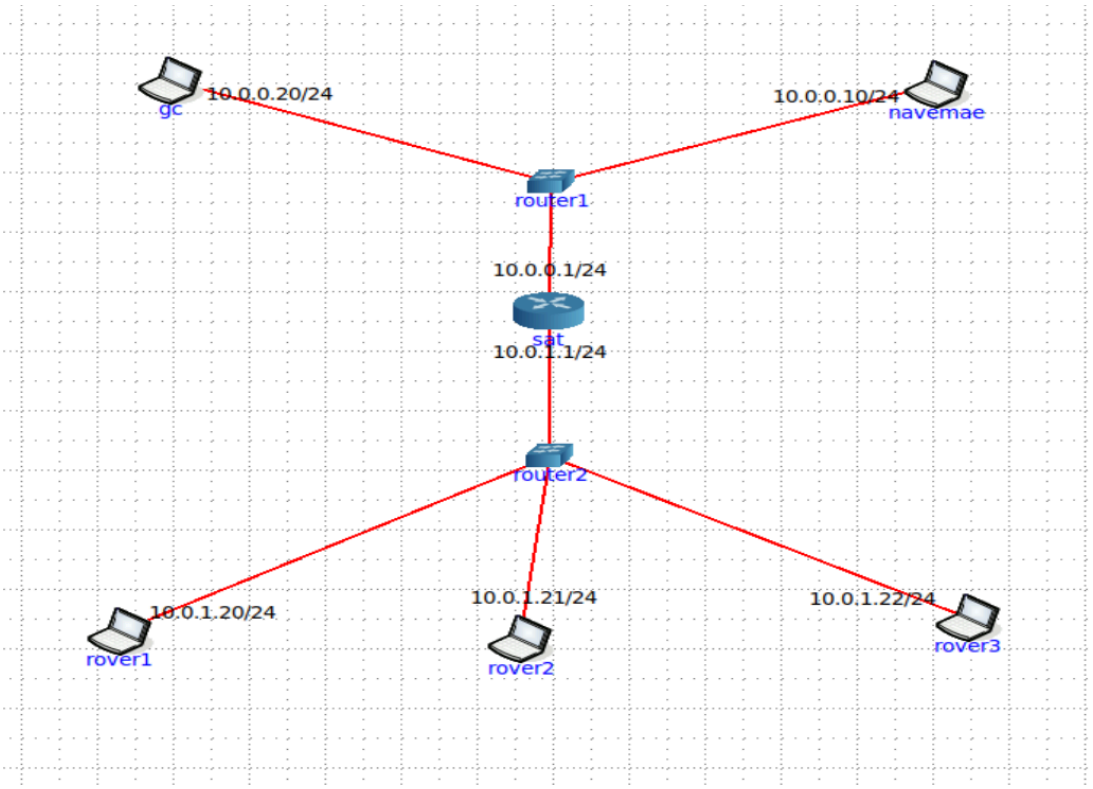


Figura 5 : Topologia de rede do sistema: Ground Control, Nave-Mãe, ligação satélite e rovers

8 | Testes aplicados

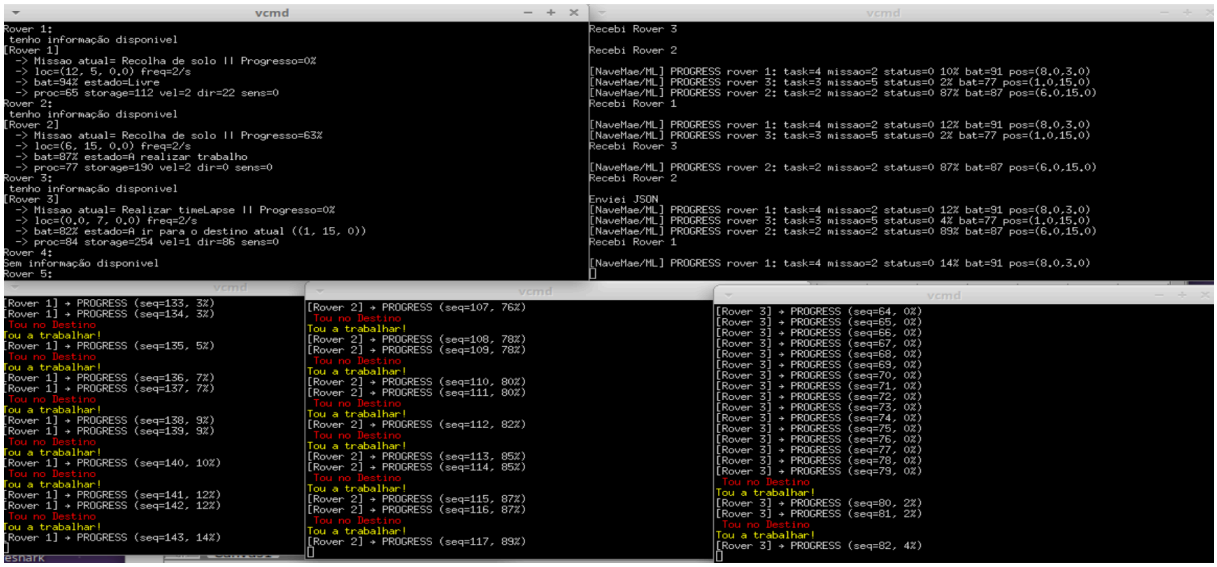
8.1 | Testes e Validação

Esta secção descreve os testes realizados com o objetivo de validar a correção funcional, fiabilidade e robustez do sistema desenvolvido. Os testes foram executados no ambiente de simulação CORE, utilizando a topologia apresentada anteriormente e três rovers em simultâneo. Foram exercitados os diferentes cenários de execução suportados pela Nave-Mãe, incluindo a atribuição de missões automáticas, o caso de ausência de trabalho e a atribuição manual de missões através do Ground Control.

8.1.1 | Testes funcionais com 3 rovers

Para validar o comportamento normal do sistema, foram realizados ensaios com 3 rovers ativos a executar missões em paralelo. Em condições de rede estáveis (sem degradação artificial), confirmou-se que:

- o fluxo READY → MISSION → PROGRESS → DONE ocorre corretamente para cada rover;
- a Nave-Mãe mantém uma visão consistente do estado de cada missão;
- as mensagens críticas do MissionLink são confirmadas por ACK, evitando perdas silenciosas.



8.1.2 | Validação por cenários

Foram testados os cenários de funcionamento disponibilizados pela Nave-Mãe:

- **Cenário 1 (sem missões)**: confirmou-se que a Nave-Mãe responde com `NOMISSION` e que os rovers aplicam backoff antes de reenviar `READY`, evitando flood do canal.
- **Cenário 2/3 (missões automáticas)**: verificou-se a atribuição sucessiva de missões a múltiplos rovers, com atualizações periódicas (`PROGRESS`) e conclusão confirmada (`DONE`).
- **Missões manuais via Ground Control**: foram atribuídas missões manualmente através do GC, confirmando-se a correta integração GC → Nave-Mãe e a entrega fiável ao rover via MissionLink.

8.1.3 | Testes de stress e tolerância a perdas

Para avaliar a robustez do MissionLink em condições adversas, foram realizados testes de stress introduzindo perda de pacotes (packet loss) na ligação satélite. O sistema foi exercitado com perdas progressivamente mais elevadas, incluindo valores até 60%, verificando-se que:

- as retransmissões e confirmações do MissionLink permitem manter a execução correta das missões, ainda que com aumento do tempo total de conclusão;
- em perdas elevadas, observou-se a ocorrência de retransmissões até ao limite configurado (`max_retries`), após o qual a operação é sinalizada como falhada (fail-safe), evitando bloqueios indefinidos.

8.1.4 | Considerações finais

Os testes com três rovers, múltiplos cenários e atribuição manual de missões demonstram que o sistema mantém consistência de estado e comportamento correto em operação normal. Adicionalmente, os testes de stress com perdas elevadas confirmam a eficácia dos mecanismos de fiabilidade implementados no MissionLink (ACKs, timeouts, retransmissões e deteção de duplicados), bem como a presença de limites de segurança (`max_retries`) para lidar de forma controlada com falhas persistentes.

9 | Conclusão

O sistema desenvolvido demonstra que é possível construir comunicação fiável e consistente sobre canais não-fiáveis através de mecanismos aplicacionais bem definidos. A separação clara entre MissionLink e TelemetryStream, aliada a uma topologia de rede realista e a cenários controlados, permitiu validar a robustez da solução em condições adversas. Os resultados obtidos confirmam a correção do desenho adotado e a adequação da arquitetura proposta para sistemas distribuídos com requisitos de fiabilidade e coordenação centralizada.