Universidade do Minho

Licenciatura em Engenharia Informática

Comunicações por Computador

2025/2026

Trabalho Prático 2

a106804, Alice Soares

a106853, Beatriz Freitas

a106927, Tiago Martins

Dezembro 2025

Índice

List of Figures

Chapter 1

Introdução

No contexto da unidade curricular 'Comunicações por Computadores' inserida no plano curricular do 3º ano letivo da Licenciatura de Engenharia Informática, elaboramos este relatório com o intuito de expor os resultados e raciocínios realizados durante a execução do trabalho prático 2, disponibilizado pelos docentes.

Este trabalho tem como principal objetivo o desenvolvimento de um ecossistema distribuído inspirado num cenário espacial composto por três tipos de entidades principais: *Rovers*, Nave-Mãe e *Ground Control*. Cada componente comunica através de múltiplos protocolos e expõe informação operacional de forma estruturada. O projeto exige a implementação de dois canais de comunicação distintos entre a Nave-Mãe e os *rovers*:

- *MissionLink* (ML), baseado em UDP, utilizado para a troca de mensagens críticas para execução de missões.
- *TelemetryStream* (TS), baseado em TCP, responsável pela transmissão contínua de telemetria dos *rovers* para a Nave-Mãe.

Para além da comunicação entre estes dois agentes, a Nave-Mãe disponibiliza uma API de Observação, acessível via HTTP, que permite a consulta externa do estado do sistema. Esta API deve fornecer a lista de *rovers* ativos e respetivo estado, a lista de missões ativas ou concluídas, a última telemetria recebida e outros.

O módulo *Ground Control* funciona como cliente desta API, consumindo periodicamente os seus *endpoints* e apresentando a informação de forma compreensível.

Este relatório descreve o desenho, implementação e justificação das decisões tomadas ao longo do desenvolvimento de cada etapa, assegurando conformidade com todos os requisitos especificados no enunciado.

Chapter 2

Descrição Geral do Sistema

1 Topologia

A topologia do sistema segue um modelo centralizado, em que a Nave-Mãe atua como núcleo de coordenação. Todos os *rovers* comunicam (passando pelos satélites intermédios) com a Nave-Mãe, que por sua vez disponibiliza o estado do sistema ao *Ground Control*. Não existe comunicação direta entre *rovers* nem entre o *Ground Control* e os *rovers*.

A figura seguinte ilustra esta organização, destacando a posição central da Nave-Mãe e os fluxos de informação estabelecidos com os restantes módulos:

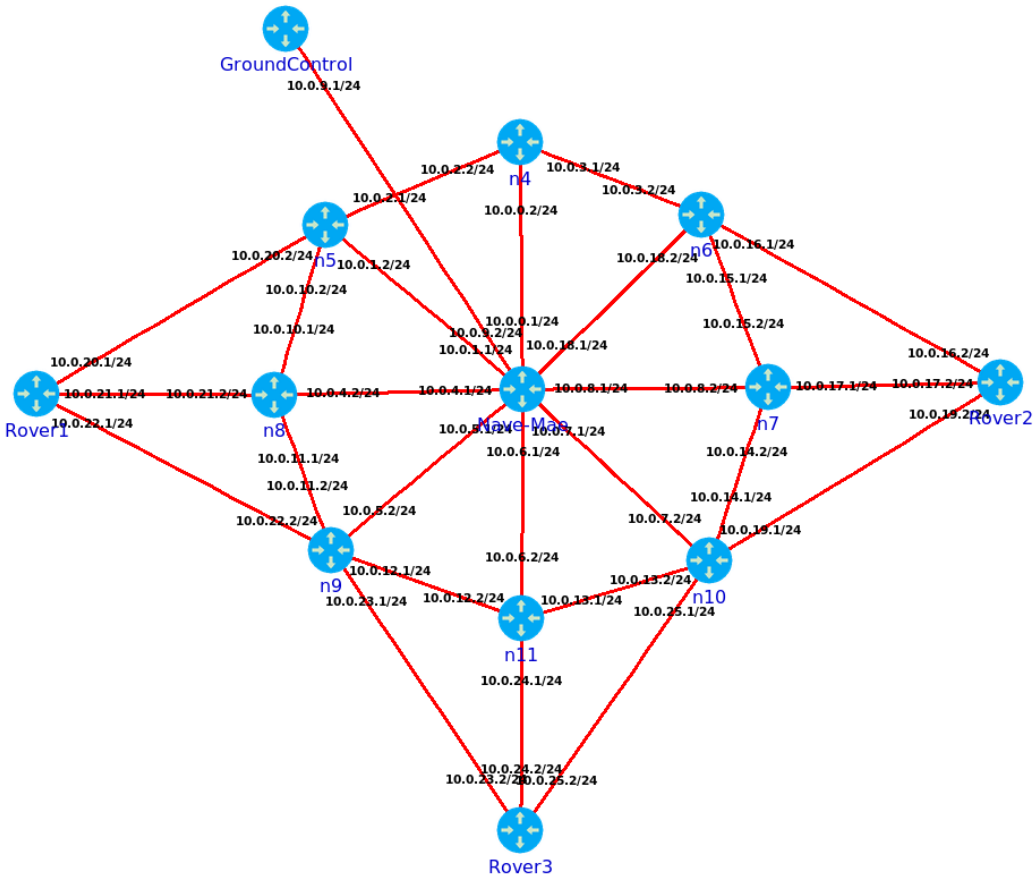

Figure 2.1: Topologia do Sistema

2 Descrição do Fluxo de Dados

O sistema foi concebido para permitir a coordenação centralizada de um conjunto de *rovers* distribuídos, garantindo que a Nave-Mãe dispõe de informação atualizada sobre o estado de cada unidade e consegue atribuir-lhes novas tarefas quando necessário. O fluxo de dados desenvolvido assenta na circulação contínua de informação entre três entidades: *Rovers*, Nave-Mãe e *Ground Control*. Cada uma destas desempenha um papel distinto no ciclo operacional.

O fluxo inicia-se nos *rovers*, que fornecem periodicamente informação sobre o seu estado interno e progresso das operações em execução. Esses dados são enviados para a Nave-Mãe, que funciona como centro lógico do sistema. A Nave-Mãe agrega, interpreta e atualiza o estado global com base na informação recebida, permitindo-lhe decidir, quando necessário, a atribuição de novas tarefas ou a atualização das condições de operação do *rover*.

Para além de receber dados, a Nave-Mãe atua também como emissor de instruções, enviando ordens e parâmetros operacionais necessários ao cumprimento das missões. Desta forma, estabelece-se um ciclo bidirecional: a nave-mãe coordena os *rovers*, enviando-lhes missões, e eles fornecem informação sobre o seu progresso.

O estado consolidado na Nave-Mãe é disponibilizado externamente através de uma interface de observação. Este ponto de acesso permite ao *Ground Control* consultar e apresentar o estado atual das missões, a situação operacional dos *rovers* e a informação mais recente recebida do terreno.

3 Comunicação entre módulos

A comunicação entre os módulos do sistema foi estruturada de forma a refletir as funções específicas de cada entidade e a garantir um fluxo de informação contínuo e coerente ao longo de toda a operação. Para tal, foram definidos três canais conceptualmente distintos, cada um deles responsável por um tipo particular de interação.

O primeiro canal (TS) estabelece ligação entre os *rovers* e a Nave-Mãe, permitindo que os *rovers* transmitam informação sobre o seu estado. Esta comunicação ocorre de forma regular, possibilitando à Nave-Mãe manter uma visão atualizada das condições operacionais de cada *rover*.

O segundo canal (ML) também envolve os *rovers* e a Nave-Mãe, mas com objetivo inverso: é a partir dele que a Nave-Mãe envia instruções e orientações que definem as ações a executar pelos *rovers*. Este canal garante que a coordenação das missões é centralizada e que todos os *rovers* seguem os parâmetros definidos no centro de controlo.

O terceiro canal estabelece a ligação entre a Nave-Mãe e o *Ground Control*. Neste caso, a comunicação é de natureza consultiva. Assim, o *Ground Control* acede ao estado consolidado que a Nave-Mãe mantém sobre o sistema. Este canal permite apresentar informação atualizada ao utilizador ou operador, oferecendo uma visão global sobre missões, *rovers* e evolução das operações.

Esta estrutura garante uma divisão clara de responsabilidades, mantendo a Nave-Mãe como ponto central do sistema e assegurando que cada módulo troca apenas o tipo de informação para o qual foi concebido.

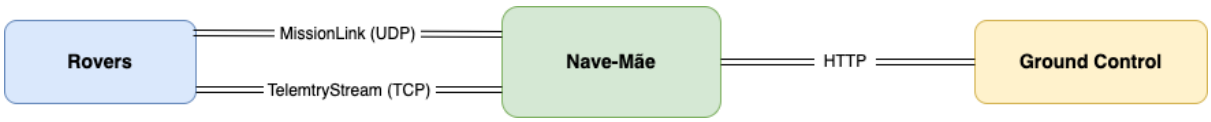

Figure 2.2: Canais de comunicação

Chapter 3

Responsabilidades do Sistema

1 *TelemetryStream*

O *TelemetryStream* é responsável pela transferência contínua de informação operacional dos *rovers* para a Nave-Mãe. No sistema implementado, esta funcionalidade é assegurada por um canal dedicado cuja única responsabilidade é garantir que a Nave-Mãe recebe, em tempo útil, o estado atualizado de cada *rover* durante a execução das suas missões.

Cada *rover* mantém uma ligação ativa à Nave-Mãe através de um cliente dedicado, que envia periodicamente um conjunto estruturado de dados de telemetria contendo posição, velocidade, estado operacional, nível de bateria e, quando aplicável, o identificador da missão em curso. Esta informação é preparada localmente pelo *rover* e enviada após um período estipulado de tempo ou quando existem novos dados a reportar derivados de um evento relevante, evitando sobrecarga desnecessária na comunicação.

Do lado da Nave-Mãe, o *TelemetryStream* é gerido por um servidor que aceita ligações individuais de cada *rover* e cria uma *thread* dedicada por conexão. A responsabilidade deste servidor é receber as mensagens de telemetria e atualizar o estado interno associado a cada *rover*.

Para além da atualização contínua do estado dos *rovers*, o módulo de telemetria assegura ainda a deteção de desconexões. Sempre que uma ligação termina inesperadamente, o servidor identifica o *rover* correspondente e regista a perda de contacto, permitindo que o sistema mantenha uma visão precisa e atualizada da disponibilidade dos dispositivos.

2 *MissionLink*

O *MissionLink* constitui o módulo responsável pela comunicação fiável de missões entre a Nave-Mãe e os *rovers*. Opera sobre UDP, mas encapsula um conjunto de mecanismos que permitem garantir que ordens críticas são entregues, reconstruídas e confirmadas de forma íntegra, mesmo num meio de transporte sujeito a perdas, atrasos e desordenação de pacotes. A sua função central é assegurar que toda a informação necessária à execução de uma missão é transmitida de forma completa, interpretável e confirmada antes do *rover* iniciar qualquer ação operacional.

O processo inicia-se com o estabelecimento explícito de contacto entre a Nave-Mãe e o *rover*. Sempre que existe uma missão pendente e um *rover* disponível, o *MissionLink* é responsável por iniciar uma sessão de comunicação dedicada, enviando uma mensagem inicial que identifica a missão e solicita confirmação da disponibilidade do *rover*. Apenas após esta validação explícita é iniciado o processo de transferência da missão, evitando atribuições incorretas ou situações de concorrência.

Segue-se a transmissão fiável dos dados da missão. Dada a dimensão variável da missão e a necessidade de garantir integridade, quando necessário, o *MissionLink* assume a responsabilidade de fragmentar a

missão em blocos coerentes e de os transmitir sequencialmente, assegurando que nenhum campo é dividido de modo inconsistente. Durante este processo, o módulo acompanha a receção do lado do *rover*, recolhendo confirmações parciais e finais, e reagindo a perdas através de retransmissões seletivas. A entrega da missão é considerada válida apenas quando todos os fragmentos foram reconhecidos e confirmados.

Uma vez concluída a transferência, o *MissionLink* continua ativo durante toda a execução da missão. É através deste canal que o *rover* comunica o seu progresso, permitindo à Nave-Mãe atualizar continuamente o estado da missão e acompanhar a evolução temporal e percentual da tarefa em curso. O módulo é responsável por monitorizar esta troca de informação, detetar ausência de mensagens dentro dos intervalos esperados e sinalizar eventuais falhas ou atrasos anómalos.

Finalmente, o *MissionLink* assegura também a conclusão formal da missão. Quando o *rover* termina a sua tarefa, envia uma mensagem explícita de finalização, acompanhada de um indicador de sucesso ou insucesso. O módulo recebe esta notificação, confirma a sua receção e desencadeia o encerramento da sessão, atualizando o estado global do *rover* e da missão. Esta etapa marca o fim lógico da interação, permitindo que o sistema retome o ciclo de planeamento e atribuição.

3 API de Observação

A API de Observação corresponde ao módulo responsável por expôr, de forma estruturada e acessível, o estado interno mantido pela Nave-Mãe relativamente aos *rovers*, missões e telemetria. A sua função é tornar o sistema observável a partir do exterior, permitindo a integração com interfaces como o *Ground Control*. A API concretiza-se com um servidor HTTP independente, baseado no HttpServer embutido no JDK, garantindo baixo acoplamento relativamente aos restantes módulos comunicacionais.

Do ponto de vista funcional, esta API desempenha três responsabilidades fundamentais. Em primeiro lugar, fornece uma forma consistente de consultar o estado agregado do sistema. Para isso, opera exclusivamente sobre a camada de gestão de estado, à qual acede através de uma interface dedicada, ObservacaoAPI. Esta separação permite isolar a lógica de transporte HTTP das operações de leitura e estruturação dos dados, assegurando que a API não altera o estado nem interfere na lógica central da Nave-Mãe. Assim, todas as consultas, como listagem de rovers, obtenção de uma missão específica, progresso de execução ou última telemetria, são tratadas como operações idempotentes de leitura que refletem diretamente o estado interno mantido pela Nave-Mãe.

A segunda responsabilidade consiste em organizar e serializar estes dados para um formato interoperável. Como a implementação assume uma dependência zero de bibliotecas externas, a serialização para JSON é realizada manualmente através do módulo CriarJson, que converte objetos internos, como Rover, Missao, PayloadTelemetria ou PayloadProgresso, em estruturas de texto devidamente formatadas. Esta escolha obriga a controlar explicitamente processos como estruturação hierárquica e representação consistente de tipos complexos, mas garante simultaneamente total controlo sobre o formato final e evita dependências externas.

Por fim, é assumida a responsabilidade de disponibilizar estes dados via HTTP, organizando-os em *endpoints* claros e semanticamente estruturados. O servidor HTTP define contextos como `/rovers`, `/missoes` e `/telemetria`, garantindo uma correspondência direta entre a estrutura interna do estado e o modelo de observação externo. Cada pedido é tratado de forma síncrona, de acordo com o método GET, e resulta tipicamente em respostas JSON representando coleções, objetos individuais ou históricos temporais. Os *endpoints* inválidos são tratados com respostas 404 definidas explicitamente, o que reforça a robustez da API perante pedidos incorretos.

4 *Ground Control*

O *Ground Control* constitui o módulo responsável pela apresentação visual e organizada do estado do sistema, funcionando como interface externa de monitorização operacional. Ao contrário dos restantes componentes, este módulo não realiza qualquer comunicação direta com os *rovers* nem participa nos mecanismos internos de coordenação da Nave-Mãe.

Todo o acesso à informação é mediado pela API de Observação, o que garante que o *Ground Control* permanece desacoplado da lógica interna e substituível sem impacto na execução das missões. O *Ground Control* não mantém estado próprio persistente. Este limita-se a refletir o estado que a Nave-Mãe expõe, assegurando que a informação apresentada corresponde à versão mais atual conhecida pelo sistema central.

Para cumprir esta função, o módulo executa operações de consulta sobre os vários *endpoints* HTTP disponibilizados pela Nave-Mãe, interpretando as respostas JSON retornadas. O processamento inclui a conversão destes dados para estruturas locais, a validação de campos essenciais e a deteção de alterações relevantes, como variações súbitas de bateria, actualizações de progresso de missão ou mudanças do estado operacional dos rovers. A componente visual (UI) utiliza estes dados para construir *dashboards* adequados ao acompanhamento em tempo real.

Além da função de monitorização, o Ground Control assume também a responsabilidade de isolar o utilizador final da complexidade técnica dos módulos comunicacionais. Através da sua representação gráfica, o operador pode observar o comportamento global do sistema sem necessidade de compreender ou interagir com os canais TCP ou UDP subjacentes. Este isolamento garante segurança, reduz o risco de interferência acidental e facilita a extensibilidade futura do sistema.

Chapter 4

Decisões Implementadas

1 Estruturação da Topologia

A topologia desenvolvida simula uma infraestrutura de comunicação complexa e distribuída, integrando a Nave-Mãe, múltiplos rovers, uma rede de satélites intermédios e um nó dedicado ao *Ground Control*. Para além de cumprir os requisitos mínimos definidos, a arquitetura foi pensada de forma a garantir múltiplos caminhos de comunicação e possibilidade de introduzir condições adversas nos links.

A topologia é composta pelos seguintes elementos principais:

- Nó da Nave-Mãe - está diretamente ligada a vários routers da constelação de satélites, garantindo *multipath* e reduzindo dependência de um único ponto de falha.
- Nós dos Rovers - Foram introduzidos três Rovers (Rover1, Rover2 e Rover3) e cada um deles está associado a diferentes satélites da rede intermédia, garantindo caminhos distintos e permitindo analisar o comportamento da comunicação em cenários diferenciados de latência e carga.
- Nós dos Satélites - A rede intermédia é constituída por um conjunto alargado de routers (n4 a n11), que representam satélites ou nós de retransmissão. Esta rede adota uma topologia parcialmente em malha, permitindo múltiplos saltos entre origem e destino e redundância em caso de falha ou degradação de um *link*
- Nó do *Ground Control* - O *Ground Control* está ligado diretamente à Nave-Mãe. Esta ligação direta foi uma decisão intencional para garantir comunicação estável e imediata entre estes dois componentes.

A estrutura final adota uma topologia híbrida. Esta abordagem foi escolhida para aumentar a resiliência do sistema e permitir o estudo mais detalhado do comportamento dos protocolos em cenários com diferentes rotas e métricas dinâmicas. Entre a nave-mãe e os satélites temos uma espécie de topologia em estrela, sendo resiliente e isolando falhas em satélites, isto é, a falha de um satélite não compromete o resto da rede, no entanto se o ponto central falhar (a nave-mãe) tudo para de funcionar. A nível dos satélites (routers intermédios), estes encontram-se interligados formando vários caminhos possíveis entre a Nave-Mãe e os *rovers*, formando uma rede em malha parcial que visa simular constelações de satélites reais, onde nem todos os satélites mantêm visibilidade mútua constante. Cada *rover* liga-se também a mais do que um satélite.

É de referir ainda que, embora os valores desta topologia base definidos nos *links* para latência e perdas estejam a zero, a topologia foi desenhada precisamente para permitir a futura introdução de cenários adversos. Esta estrutura é também escalável, ou seja, permite adicionar novos satélites e *rovers* facilmente, não comprometendo o seu bom-funcionamento.

2 *TelemetryStream*

O TelemetryStream é o protocolo de aplicação responsável pela transmissão contínua de dados de telemetria dos rovers para a Nave-Mãe, permitindo a monitorização em tempo real do estado operacional de cada *rover*. Ao contrário do MissionLink, este protocolo assenta sobre transporte TCP, beneficiando das garantias nativas de fiabilidade, ordenação e controlo de fluxo da camada de transporte. O protocolo implementa um modelo de comunicação híbrido, combinando envios periódicos regulares com envios reativos desencadeados por eventos significativos, garantindo que a Nave-Mãe mantém uma visão atualizada e de baixa latência do estado de cada *rover*. A comunicação estabelece-se através de uma conexão persistente que permanece ativa durante todo o ciclo de vida operacional do *rover*, com mecanismos automáticos de reconexão para garantir robustez perante falhas de rede temporárias.

2.1 Formato das Mensagens Trocadas

As mensagens utilizadas pelo *TelemetryStream*, são do tipo MSG_TELEMETRY e são todas constituídas por um CabeçalhoTCP comum a todas as mensagens e um *PayloadTCP* que poderá ser do tipo *PayloadTelemetria*.

CabeçalhoTCP

O cabeçalho TCP é constituído pelos seguintes campos:

- **tipo (TipoMensagem)** - identifica o tipo de Mensagem referido anteriormente.
- **idEmissor (int)** - identifica o emissor da mensagem
- **idRecetor (int)** - identifica o recetor da mensagem.
- **idMissao (int)** - identifica a missão.
- **timestamp (Time)** - identifica o tempo aquando o envio da missão.

PayloadTelemetria

A telemetria do *rover*, enviada por mensagens do tipo `MSG_TELEMERY`, inclui:

- **posicaoX, posicaoY (float)** - coordenadas da posição atual do *rover*;
- **estadoOperacional (enum)** - estado interno atual do *rover*;
- **bateria (float)** - bateria atual em percentagem;
- **velocidade (float)** - velocidade instantânea.

2.2 Protocolo

1ª Fase - Iniciação do *Rover*

Ao arrancar, tenta estabelecer uma conexão TCP com a Nave-Mãe na porta 5001. O processo utiliza um *timeout* de conexão de 5 segundos.

2ª Fase - Aceitação Nave-Mãe

Quando a Nave-Mãe aceita uma conexão, cria uma *thread* específica para lidar com o *Rover*.

3ª Fase - Identificação Automática

A primeira mensagem de telemetria enviada pelo *rover* identifica-o através do campo *idEmissor* no cabeçalho. A Nave-Mãe regista o *rover* no sistema de gestão de estado, associando-o ao endereço IP da conexão.

Transmissão de Telemetria

O *rover* envia dados de telemetria de forma contínua através da conexão TCP estabelecida, seguindo um modelo híbrido que combina envios periódicos com envios reativos.

1. **Envio Periódico Regular -** O *rover* transmite mensagens `MSG_TELEMETRY` a cada **10 segundos**, independentemente do seu estado operacional. Cada mensagem contém um resumo completo do estado atual do *rover* (*payloadTelemetria*). O campo `idMissao` no cabeçalho identifica a missão ativa; se o *rover* não estiver a executar nenhuma missão, este campo é definido como `-1`.
2. **Envio por Ocorrência Significativa -** Além dos envios periódicos, o *rover* transmite telemetria imediatamente sempre que ocorre em evento relevante, garantindo que a Nave-Mãe é notificada sem atraso de eventos críticos como, por exemplo:
    - **Início de Missão:** Quando o *rover* aceita uma nova missão (transição de *DISPONIVEL* para *EM_EXECUCAO*), envia telemetria com o novo `idMissao` e estado atualizado.
    - **Conclusão de Missão:** Ao concluir uma missão com sucesso (transição para *CONCLUIDO*), o *rover* envia telemetria imediata com o novo estado.
    - **Falha Operacional:** Em caso de erro crítico (transição para *FALHA*), transmite imediatamente o estado de falha à Nave-Mãe.
    - **Bateria Crítica:** Quando a bateria atinge níveis críticos (< 10%), dispara o envio imediato desta informação.

Este modelo híbrido (periódico + reativo) garante que a Nave-Mãe tenha sempre uma visão atualizada do *rover*, com latência máxima de 10 segundos em operação normal e resposta instantânea a eventos importantes.

Processamento no Servidor

A Nave-Mãe processa cada mensagem de telemetria recebida para manter o estado global atualizado em tempo real. Assim, todos os campos de telemetria (posição, bateria, velocidade, estado operacional) são imediatamente refletidos no registo interno do *rover*.

O campo `idMissao` permite à Nave-Mãe sincronizar automaticamente a missão no *rover*:

- Se `idMissao > 0`: o *rover* está a executar uma missão → atualiza `temMissao = true` e `idMissaoAtual`.
- Se `idMissao <= 0`: o *rover* não tem missão ativa → atualiza `temMissao = false` e `idMissaoAtual = -1`.

2.3 Mecanismos de Controlo

Mecanismo de Reconexão Automática

A comunicação entre o *rover* e a nave depende de uma ligação TCP persistente, necessária para o envio periódico de telemetria. Para garantir robustez mesmo perante falhas de rede, o módulo ClienteTCP implementa um mecanismos de reconexão automática.

Quando a ligação é perdida, o cliente tenta restabelecer a comunicação com o servidor. O mecanismo tem um limite definido de tentativas, que é incrementado a cada falha. Ao atingir esse limite, o cliente desiste de tentar reconectar, prevenindo ciclos infinitos. Além disso, após cada falha, o cliente aguarda um intervalo fixo de tempo, evitando tentativas sucessivas demasiado rápidas. Sempre que a ligação é restabelecida com sucesso, o contador de tentativas é reposto a zero. Isto é importante para permitir que o cliente volte a ter todo o número de tentativas disponíveis caso ocorra uma nova falha mais tarde. Antes de qualquer nova tentativa, o *socket* anterior é fechado explicitamente. Isto evita problema, como por exemplo, portas ocupadas, que possam interferir com a reconexão.

Encerramento da Conexão

O encerramento da conexão TCP pode ocorrer de forma planeada ou devido a falhas de comunicação:

- **Encerramento Normal -** Quando o *rover* desliga (através do método `stop()`), fecha explicitamente o *socket* TCP, sinalizando à Nave-Mãe o fim da comunicação. A Nave-Mãe deteta o encerramento através de uma exceção no *stream* de entrada e marca o *rover* como desconectado.
- **Encerramento por Falha de Reconexão -** Se o *rover* não consegue restabelecer a conexão após esgotar as 10 tentativas de reconexão (intervalo de 5 segundos entre cada tentativa), o cliente TCP termina definitivamente. Neste cenário, o rover fica inoperacional até ser reiniciado manualmente.
- **Shutdown do Servidor -** Quando a Nave-Mãe é desligada, o `ServerSocket` é fechado, provocando o encerramento de todas as conexões ativas com os *rovers*. Os *rovers* detetam a perda de conexão e iniciam automaticamente o processo de reconexão, aguardando que o servidor volte a ficar disponível.

Garantias de Fiabilidade

Ao contrário do protocolo MissionLink (UDP), o TelemetryStream beneficia das garantias nativas do TCP:

- **Entrega Garantida:** Todas as mensagens de telemetria são garantidamente entregues, ou a conexão é considerada perdida.
- **Ordenação:** As mensagens chegam sempre na ordem em que foram enviadas, eliminando a necessidade de números de sequência.
- **Controlo de Fluxo:** O TCP ajusta automaticamente a taxa de transmissão conforme a capacidade da rede e do recetor.
- **Deteção de Erros:** *Checksums* do TCP garantem integridade dos dados, dispensando validações adicionais ao nível da aplicação.

3 *MissionLink*

3.1 Formato das Mensagens Trocadas

O protocolo aplicacional desenvolvido assenta num conjunto estruturado de mensagens transmitidas sobre UDP. Cada mensagem contém um cabeçalho comum e um *payload* específico (PayloadAck, PayloadErro, PayloadMissao ou PayloadProgresso), dependente do tipo de mensagem. O formato das mensagens foi projetado para ser simples, compacto e com informações úteis aos mecanismos de controlo, respeitando integralmente os requisitos definidos no enunciado do projeto.

TipoMensagem

Definimos que uma mensagem utilizada pelo *MissionLink* pode ser destes tipos:

- `MSG_HELLO` - mensagem associada ao envio de pacotes HELLO.
- `MSG_RESPONSE` - mensagem de resposta a um pacote recebido.
- `MSG_MISSION` - mensagem para atribuição ou envio de dados de missão.
- `MSG_ACK` - mensagem de confirmação de receção (acknowledgement).
- `MSG_PROGRESS` - mensagem de atualização do estado/progresso da missão.
- `MSG_COMPLETED` - mensagem indicando conclusão da missão.
- `MSG_ERROR` - mensagem de notificação de erro ou falha.

CabeçalhoUDP

Todas as mensagens transmitidas através do protocolo de aplicação desenvolvido sobre UDP utilizam um cabeçalho comum, representado pela classe `CabecalhoUDP`. Este cabeçalho garante a identificação correta da mensagem, a associação à missão relevante e garante apoio aos mecanismos de controlo.

- **tipo (TipoMensagem)** - identifica o tipo de mensagem.
- **idEmissor (int)** - identifica o emissor da mensagem.
- **idRecetor (int)** – identifica o recetor da mensagem.
- **idMissao (int)** - identifica a missão.
- **timestamp (Time)** - identifica o tempo aquando o envio da missão.
- **seq (int)** - identifica o numero do fragmento.
- **totalFragm (int)** - identifica o numero total de fragmentos.
- **flagSucesso (boolean)** - flag a verificar o sucesso da mensagem.

PayloadMissao

Uma missão é uma mensagem enviada pela Nave-Mãe para um *rover* do tipo `MSG_MISSION` e contém toda a informação necessária para que o *rover* a compreenda e execute, incluinda nos seguintes campos:

- **idMissao (int)** - cada missão tem uma identificação única, garantindo a distinção entre diferentes missões.
- **x1, y1, x2, y2 (float)** - representa a área geográfica de operação da missão, sendo definida por um retângulo usando duas coordenadas (canto superior esquerdo e canto inferior direito). A escolha da representação retangular deve-se à sua simplicidade, formato compacto, de fácil de serialização e compatibilidade com diversas tarefas.
- **tarefa (String)** - texto que define a operação a executar pelo *rover*.
- **duracaoMissao (long)** - tempo máximo permitido para execução em segundos;
- **intervaloAtualizacao (long)** - frequência com que o *rover* deverá enviar mensagens de progresso, sendo este um intervalo em segundos.
- **inicioMissao (long)** - instante a partir do qual a missão é considerada ativa.
- **prioridade (int)** - valor entre 1 e 5 que determina a prioridade da missão, suporta a gestão de múltiplas missões.

PayloadProgresso

Durante a execução da missão, o *rover* envia mensagens do tipo `MSG_PROGRESS` com:

- **idMissao (int)** - identifica a missão ativa a ser reportada;
- **tempoDecorrido (long)** - tempo decorrido desde o início da missão em segundos;
- **progressoPercentagem (float)** - valor em percentagem do progresso da missão;

PayloadAck

Durante toda a comunicação é usada várias vezes a mensagem do tipo `MSG_ACK` por ambos, constituida por:

- **missingCount (int)** - informa, caso aplicável, o número de fragmentos perdidos da mensagem recebida ao qual está a responder. Se for maior do que 0 o recetor sabe que terá mensagens a retransmitir;
- **misssing (int[])** - lista dos índices dos fragmentos efetivamente perdidos, para os quais é pedida a retransmissão;
- **finalAck (boolean)** - flag que indica que este será o `ACK` final, usado na resposta a mensagens do tipo `COMPLETED/ERROR`

PayloadErro

Usado pelos *rovers* para avisar a nave que não conseguirão completar a missão devido a erro. Para posteriormente a nave gerir o estado sobre o *rover*, este *payload* é constituído por um relatório completo sobre o erro:

- **idMissao (int)** - identifica a missão que falhou;
- **codigoErro (int)** - como o erro pode ser de vários tipos, criou-se um código de erro númerico, pois teria um uso mais prático na identificação do erro. O códigoErro pode ser:
  1. `ERRO_BATERIA_CRITICA` - Bateria crítica (<10%)
  2. `ERRO_BATERIA_BAIXA` - Bateria baixa (<20%)
  3. `ERRO_OBSTACULO` - Obstáculo intransponível pelo *rover*
  4. `ERRO_HARDWARE` - Falha de hardware do *rover*
  5. `ERRO_TIMEOUT` - Tempo limite para concluir a missão chegou ao fim antes do seu término
  99. `ERRO_DESCONHECIDO` - Erro desconhecido ou código de erro inválido
- **descricao (String)** - descrição textual detalhada do erro recebido;
- **progressoAtual (float)** - percentagem de progresso alcançada até à ocorrência do erro;
- **bateria (float)** - nível de bateria do *rover* no momento da falha;
- **posicaoX, posicaoY (float)** - coordenadas da posição do *rover* quando ocorreu o erro;
- **timestampErro (long)** - instante temporal em que o erro foi detetado

3.2 Fragmentação das mensagens

A comunicação baseada em UDP impõe uma limitação quanto ao tamanho máximo das mensagens. Para garantir o funcionamento robusto do módulo de comunicação, optou-se por implementar um mecanismo próprio de fragmentação ao nível da aplicação, controlado pelo módulo `SerializadorUDP`. Dado os mecanismos de controlo implementados, poderíamos ter optado por uma fragmentação "ás cegas" em que apenas serializávamos a mensagem e dividíamos pelo tamanho máximo. No entanto, achamos vantajoso implementar uma fragmentação por campos, para haver a possibilidade de reconstruir campos individualmente, podendo ser utilizada para aproveitar informações dos campos recebidos, se não chegassem todas as mensagens ao devido destino, em vez de descartar logo uma mensagem inteira.

Unidade de Fragmentação - Campo Serializado

Ao contrário de soluções que fragmentam a mensagem completa, esta implementação fragmenta a unidade lógica mínima: o *campo*, representado pela classe `CampoSerializado`. Cada campo contém:

- o nome do campo,
- os seus dados binários,
- `indiceParte` — posição deste fragmento
- `totalPartes` — número total de fragmentos pertencentes ao mesmo campo.

`indiceParte` e `totalPartes` apenas são utilizados quando um campo é grande demais para um único fragmento.

Esta abordagem permite:

- reconstruir o campo independentemente da ordem de chegada;
- suportar campos com tamanhos muito superiores ao limite de fragmentação;
- manter compatibilidade com vários tipos de payload.

Empacotamento Otimizado em `FragmentoPayload`

Após a fragmentação individual dos campos grandes, o sistema realiza um processo de **empacotamento best-fit**, cujo objetivo é maximizar o aproveitamento do espaço disponível em cada fragmento UDP. O algoritmo aplica:

1. fragmentação de campos que excedem o limite definido;
2. cálculo do espaço restante em cada fragmento UDP;
3. seleção do campo (ou fragmento de campo) que melhor se ajusta ao espaço disponível, minimizando desperdício;
4. criação de novos fragmentos quando não existe qualquer campo que caiba no atual.

Esta estratégia reduz o número total de fragmentos enviados, o que diminui:

- o overhead de transmissão;
- a probabilidade global de perda de fragmentos;
- o custo total de reconstrução no recetor.

Agregação e Reconstrução no Recetor

O recetor mantém um estado interno através de um mapa `camposPorNome`, no qual:

- cada campo recebe as suas partes à medida que chegam;
- é possível verificar se um campo está completo comparando o número de partes recebidas com `totalPartes`;
- permite recontruir o objeto usando os campos recebidos.

3.3 Protocolo

O *Mission Link* é o protocolo de aplicação responsável pela comunicação fiável entre a Nave-Mãe e os *rovers* para o envio e monitorização de missões. Embora assente sobre transporte UDP o protocolo adiciona mecanismos explícitos de controlo de fiabilidade, ordenação, deteção de perdas e confirmação de estado. Todas as interações pertencem a um único protocolo, mas este está dividido em quatro fases lógicas: ”handshake”, envio da missão, monitorização do progresso e finalização.

1ª Fase - Handshake

O objetivo desta fase é estabelecer uma sessão entre a Nave-Mãe e o *rover*, garantindo que este está disponível para receber uma missão, funcionando como um pré-acordo simples antes de iniciar a transferência da missão. A primeira fase do protocolo consiste num ”handshake” leve:

1. A Nave-Mãe inicia sempre a comunicação enviando uma mensagem **HELLO**, que contém um identificador de missão a atribuir e um número de sequência inicial (seq=1). A perda do pacote é tratada com um mecanismo simples de retransmissão: caso não receba resposta dentro do *timeout*, a Nave-Mãe pode reenviar o HELLO até ao limite máximo configurado.
2. O *rover* responde com **RESPONSE**, indicando explicitamente no cabeçalho:
   - suc=true → está disponível para aceitar a missão.
   - suc=false → encontra-se ocupado ou indisponível. Neste caso, a sessão termina imediatamente e a Nave-Mãe pode tentar outro *rover* sem necessidade de retransmissões adicionais.

O seguinte diagrama de sequência espelha a primeira fase do protocolo implementado.

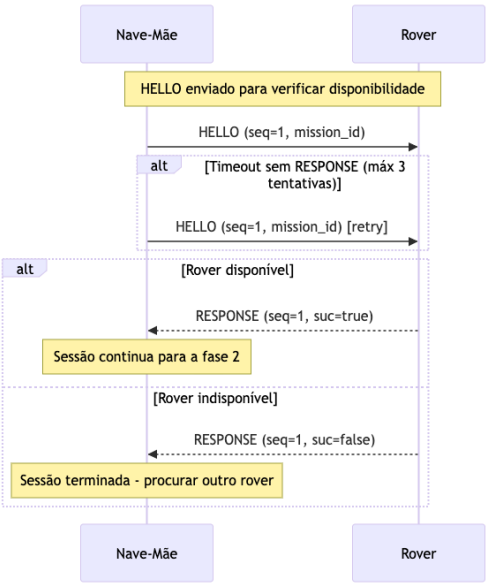

Figure 4.1: 1ª Fase

2ª Fase - Envio da Missão

As missões podem variar significativamente de tamanho. Para manter o protocolo eficiente e compatível com os limites máximos de transmissão, o envio é adaptativo.

1. A nave-mãe começa o processo de envio da missão para o *rover* com a sessão previamente estabelecida, tendo em conta o tamanho da missão:
   - Missões Pequenas - Se o payload for inferior ao limite de 512 bytes, a Nave-Mãe envia uma única mensagem **MISSION** contendo, no cabeçalho, seq=2, tot=1 e o payload completo.
   - Missões Grandes - Quando o payload excede o limite, a missão é fragmentada. Cada fragmento é enviado como uma mensagem **MISSION(seq=i+1, tot=N)**, sendo:

  - seq o número de sequência do fragmento;
  - tot o número total de fragmentos.

2. Após receber todos os pacotes (ou após o timeout interno), o *rover* verifica a completude da missão:

   - Se faltarem fragmentos, envia **ACK(missing=[lista de seq])**, com lista de seq= aos números de sequência dos fragmentos em falta.
   - Se tudo estiver correto, envia **ACK(missing=[])**.

3. A Nave-Mãe retransmite apenas os fragmentos perdidos, seguindo o mesmo processo, até receber um **ACK(missing=[])** por parte do *rover*.

4. Depois de receber todos os fragmentos, o *rover* reconstrói a missão localmente e inicia a execução.

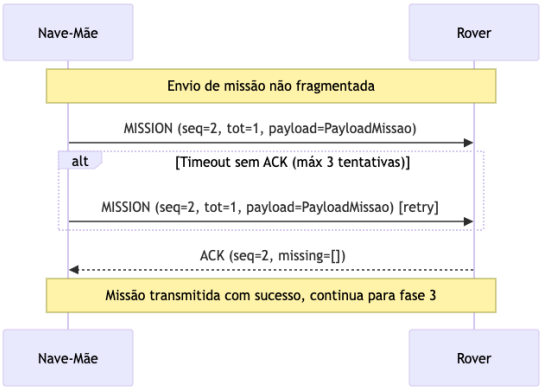

Figure 4.2: 2ª Fase - Caso Missão Pequena

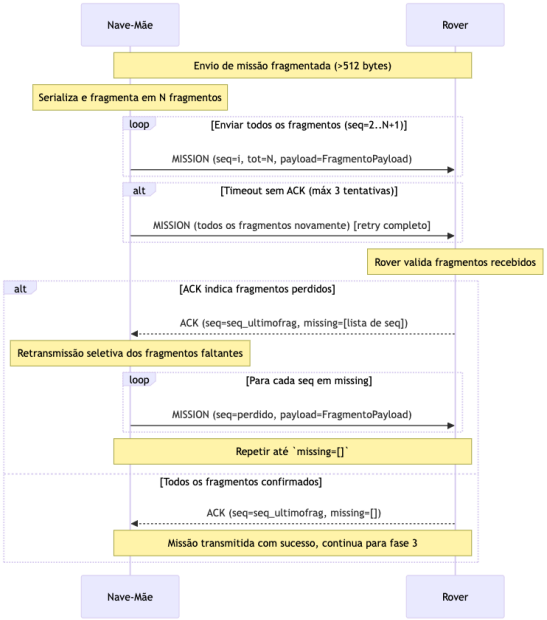

Figure 4.3: 2ª Fase - Caso Missão Fragmentada

3ª Fase - Monitorização da Missão

Durante a execução, o *rover* envia periodicamente mensagens **PROGRESS**, para manter a nave-mãe atualizada do estado da missão. O *timeout* nesta fase é dinâmico e depende do intervalo esperado para atualizações de cada missão, evitando falsos positivos de falha.

- Para cada mensagem **PROGRESS** enviada, o *rover* espera receber um **ACK**.
- Caso o **ACK** indique sequências perdidas, o *rover* reenvia seletivamente os relatórios de progresso em falta.
- Do lado da Nave-Mãe, cada **PROGRESS** recebido é imediatamente refletido no estado interno da missão.

O seguinte diagrama de sequência espelha a primeira fase do protocolo implementado.

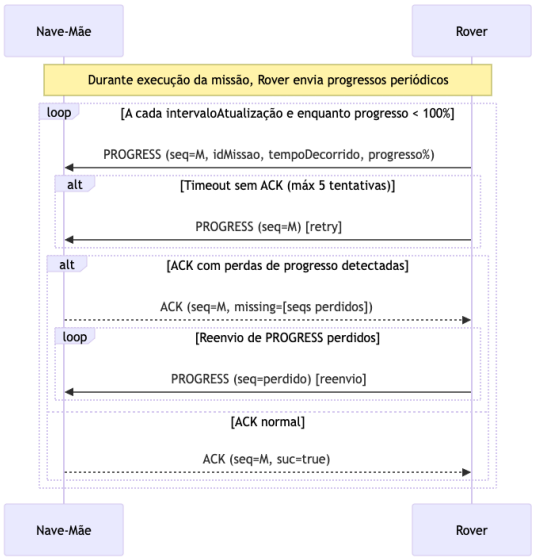

Figure 4.4: 3ª Fase

4ª Fase - Finalização da Missão

A conclusão da missão acontece quando o *rover* atinge 100% de progresso ou quando ocorre um erro irreversível.

- Conclusão com Sucesso:

  1. O *rover* envia **COMPLETED (seq=M+1, suc=true)** e retransmitirá caso não chegue nenhum **ACK** da nave-mãe.
  2. Para garantir máxima robustez, a Nave-Mãe envia vários ACK's de confirmação final **(finalAck=true)** de uma vez.
  3. Caso nenhum dos ACK's chegue, o *rover* retransmitirá a mensagem de COMPLETED.
  4. Após este passo, a sessão é considerada como concluída, a Nave-Mãe marca a missão como CONCLUIDA e o estado do *Rover* é atualizado para DISPONIVEL.

O seguinte diagrama de sequência espelha esta fase do protocolo implementado.

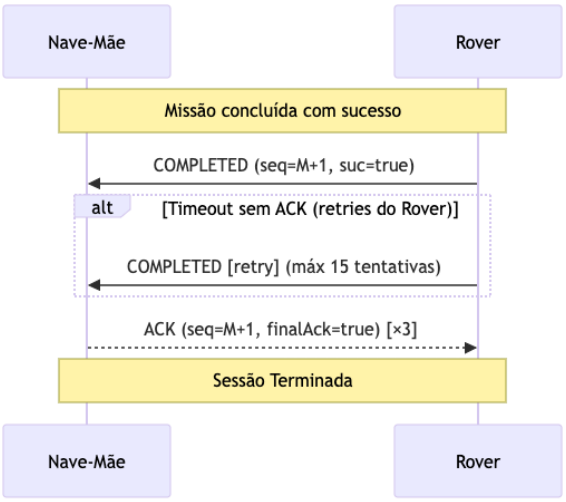

Figure 4.5: 4ª Fase - Caso COMPLETED

- Conclusão com Erro:

  1. Em caso de falha operacional, o *rover* envia mensagem **ERROR**.
  2. Segue-se um processo idêntico ao da mensagem COMPLETED: retransmissões do *rover* em caso de timeout e envio de múltiplos ACK finais pela Nave-Mãe.
  3. Depois desta última troca, a sessão é considerada como concluída, a Nave-Mãe marca a missão como FALHADA e o *rover* é atualizado para o estado FALHA.

  O seguinte diagrama de sequência espelha esta fase do protocolo implementado.

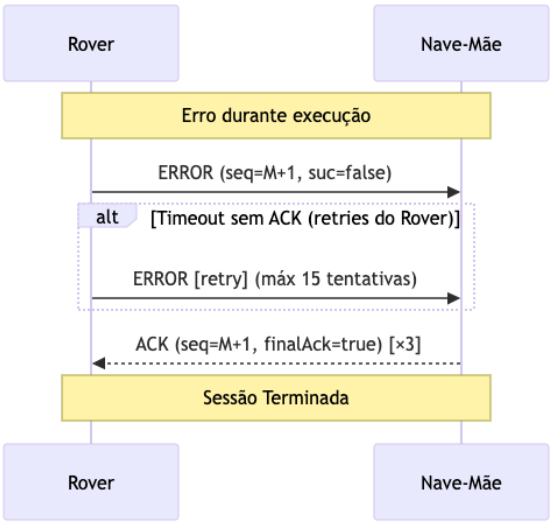

Figure 4.6: 4ª Fase - Caso ERRO

À exceção da conclusão de missão com erro, que pode acontecer a qualquer momento durante a execução da missão pelo *rover*, todas as fases são encadeadas.

3.4 Mecanismos de controlo

Uma vez que o protocolo de MissionLink assenta sobre UDP, tornou-se necessário aplicar vários mecanismos de controlo, de forma a garantir uma maior fiabilidade na comunicação.

Controlo de Sessão

Introduzimos um mecanismo explícito de controlo de sessão. Este permite ao servidor da nave-mãe e ao *rover* manterem um estado consistente da comunicação, assegurando que é recuperável, ordenada e monitorizável, compensando limitações inerentes ao protocolo subjacente.

Para tal, cada entidade (Nave-Mãe e *Rover*) mantém uma sessão independente:

- no lado do servidor, existe um `HashMap` que associa cada `idRover` à sua `SessaoServidorMissionLink`, permitindo gerir múltiplos *rovers*, isto é, múltiplas sessões de comunicação em paralelo;
- no lado do cliente, a classe `SessaoClienteMissionLink` representa uma sessão isolada, refletindo o estado corrente da sua comunicação com a nave-mãe.

A existência de duas estruturas de sessão distintas é essencial devido às responsabilidades diferentes. Esta separação evita estados inconsistentes e impede que a falha de um *rover* ou missão comprometa outras operações. Cada sessão armazena um conjunto de parâmetros essenciais, distintos para servidor e cliente mas complementares no seu papel:

- **Endereço e porta**: permitem identificar inequivocamente o *endpoint* remoto associado à missão. No servidor, estes valores permitem reenviar fragmentos ou mensagens ao *rover* correto.
- **Número de Sequência**: como o UDP não garante ordenação, cada sessão mantém informação sobre os números de sequência dos cabeçalhos das mensagens recebidas.
  No caso do `SessaoClienteMissionLink`, guarda a sequência atual da mensagem recebida. Na `SessaoServidorMissionLink`, guarda o último número de sequência recebido. Este controlo é fundamental para:
  - tratar pacotes duplicados;
  - detetar perdas de fragmentos: Como o UDP não retransmite dados, com estas sessões é permitido identificar pacotes desaparecidos através de lacunas de sequência e solicitar retransmissões, evitando que fragmentos incompletos invalidem toda a missão;
  - reconstruir corretamente as mensagens fragmentadas;
  - validar mensagens (ACKs) recebidas: o cliente mantem um *ACK esperado* e último confirmado, que permite ignorar ACKs fora de ordem, saber exatamente que mensagem foi confirmada e retransmitir apenas a que falhou.
- **Progresso da missão**: mantém o estado atual do *rover* relativamente ao tempo decorrido e percentagem concluída, permitindo ao servidor identificar falhas, atrasos e inconsistências e, se necessário, reemitir pedidos ou sinalizar erro.
- **Fragmentos enviados e perdidos**: Ambos os lados incluem estruturas explícitas para armazenar:
  - fragmentos recebidos/enviados
  - fragmentos em falta (na receção)
  - número total de fragmentos previstos.

Esta informação é indispensável para reconstruir mensagens fragmentadas e, igualmente, para implementar retransmissão seletiva. Mensagens longas são divididas em múltiplos fragmentos (`FragmentoPayload`). Com esta informação armazenada localmente na sessão armazena-se localmente, permite-se uma reconstrução correta e detetar omissões de fragmentos.

- **Serializador**: ambas as sessões possuem uma instância do serializador que trata da serialização e desserialização das mensagens.
- **Estado da comunicação**: cada sessão mantém a informação do estado da comunicação. Na `SessaoServidorMissionLink`, tem informação da recessão e/ou sucesso de mensagens `RESPONSE`, `PROGRESS`, `COMPLETED` e `ERROR`, se ainda está a enviar o finalAck, assim como *timestamp* de última atividade e um valor limite para considerar a sessão como inativa. Já na `SessaoClienteMissionLink` tem informação sobre os intervalos em que tem que mandar mensagens de `PROGRESS`, e até quando o deve fazer, assim como informação para a confirmação de mensagens `ACK` recebidas e se ainda está ou não à sua espera. Tem também informação sobre se a sessão está ou não a decorrer, para não começar uma sessão nova que entre em conflito.
- **Missão e *Rover***: A `SessaoServidorMissionLink` contém ainda informação sobre a missão a ser enviada e o *Rover* que a receberá. Na `SessaoClienteMissionLink` apenas tem o idMissao que é enviado no cabeçalho das mensagens para confirmação na receção das mensagens.

  Esta abordagem garante isolamento entre envios de missões e persistência da informação necessária para deteção de perdas, reconstrução de fragmentos e controlo de retransmissões, garantindo uma certa integridade ao protocolo. Além disso, ao manter o estado numa sessão persistente, evita-se incluir nas mensagens metadados redundantes, tornando o protocolo mais leve e modular.

Controlo de Sequência e Duplicação

Todas as mensagens do protocolo *MissionLink* incluem um campo `seq` no cabeçalho.

Para implementação de fragmentação, foi decidido utilizar duas variáveis, `seq` e `totalFragm`. A variável `seq` identifica o número de sequência de cada fragmento, enquanto `totalFragm` indica o número total de fragmentos da mensagem. Esta abordagem permite ao recetor verificar a integridade de cada fragmento recebido e determinar a sua posição na mensagem completa, possibilitando a deteção de fragmentos perdidos ou recebidos fora de ordem, características essenciais para assegurar a fiabilidade numa comunicação baseada em UDP.

Tanto no servidor como no cliente, é implementado a:

- verificação de sequência incremental;
- deteção de duplicados, assim, mensagens duplicadas nunca produzem efeitos secundários e apenas são ignoradas ou originam ACK coerente;
- tratamento distinto das sequências para:
  - fragmentos da missão;
  - progressos;
  - mensagens finais.

Ainda como mecanismo contra a duplicação, a estrutura que guarda os fragmentos perdidos da missão na sessão é um `Set` evitando repetições de retransmissão por ACK's duplicados.

Timeouts e Retransmissões

Para diferentes fases do protocolo é feita a retransmissão de mensagens quando a resposta não é recebida num dado intervalo de tempo (*timeout*). Os valores foram definidos segundo a seguinte tabela:

Tipo Mensagem	Timeout	Comportamento
HELLO	5 s	Retransmissão até 3 vezes
Fragmentos da missão	5 s	Retransmissão seletiva
PROGRESS	Dinâmico (2 * tempoDeAtualização)	Retransmite até 5 vezes

Gestão de Perdas

O protocolo identifica perdas de mensagens nas seguintes situações:

- fragmentos em falta na recessão da missão;
- quebras na sequência de PROGRESS;

Assim, implementamos um mecanismo de pedido de retransmissão, em que o recetor envia, no ACK imediatamente a seguir, uma lista de sequências dos fragmentos perdidos, para que o emissor saiba quais os fragmentos ou mensagens a retransmitir. Esta recuperação é sempre selectiva, evitando transmitir informação redundante. Este mecanismo de ACK seletivo é fundamental para reduzir tráfego desnecessário, preservando largura de banda e diminuindo latências, algo crítico em cenários onde o *rover* pode ter conectividade limitada.

Deteção e Envio de Erros

Durante a execução da missão, o *rover* verifica continuamente condições críticas que podem impedir a conclusão bem-sucedida da tarefa. A deteção de erros ocorre no método `verificarCondicoesErro()` executado antes de cada envio de `PROGRESS`.

As condições monitorizadas incluem:

- **Bateria Crítica**: Se o nível de bateria cair abaixo de 10%, o *rover* considera que não possui energia suficiente para completar a missão e retornar à base. Neste caso, dispara imediatamente um erro do tipo `ERRO_BATERIA_CRITICA`.
- **Falha de Hardware**: Se a máquina de estados interna do *rover* transita para o estado `ESTADO_FALHA` devido a qualquer anomalia operacional, é enviado um erro `ERRO_HARDWARE`.

Isto é muito importante para a gestão interna das missões na nave-mãe.

Dado que as mensagens de erro são críticas para a coordenação do sistema, implementa-se um mecanismo de retransmissão mais agressivo que o utilizado para `PROGRESS`. O número máximo de tentativas é triplicado (`MAX_RETRIES * 3`), resultando em aproximadamente 15 tentativas com intervalos de 3 segundos entre cada uma. Este comportamento garante que, mesmo em condições severas de perda de pacotes, a Nave-Mãe é notificada da falha operacional do *rover*.

Após enviar a mensagem `ERROR`, se esgotar todas as tentativas de retransmissão (com ou sem confirmação da Nave-Mãe), o *rover* finaliza a sessão de comunicação, transita o seu estado interno para `ESTADO_FALHA` e permanece indisponível para novas missões até que o sistema de recuperação automática o restaure ao estado. Já a nave-mãe, se receber uma mensagem destas, gere o estado interno da missão e finaliza também a sessão de comunicação. Este mecanismo garante que falhas operacionais são comunicadas de forma fiável e que o estado do sistema permanece consistente.

Encerramento Fiável da Sessão

O encerramento correto da sessão de missão é crítico para manter a consistência do estado do sistema. O protocolo implementa mecanismos específicos para garantir sincronização entre Nave-Mãe e *rover*, mesmo em condições adversas de comunicação.

- **Confirmação Tripla do ACK Final**

  Para garantir que o *rover* recebe a confirmação de encerramento, a Nave-Mãe envia três mensagens `ACK` consecutivas, espaçadas por 100ms, todas marcadas com a *flag* `finalAck = true`. Esta redundância aumenta drasticamente a probabilidade de pelo menos uma mensagem chegar ao destino, mesmo em cenários de perda elevada de pacotes.

  Do lado do *rover*, ao receber qualquer ACK com `finalAck = true`, este:

  - Interrompe imediatamente todas as retransmissões pendentes;
  - Marca a sessão como confirmada (`aguardandoAck = false`);
  - Atualiza o estado interno e finaliza a sessão localmente.

- **Encerramento Unilateral por Timeout**

  Se, após esgotar todas as tentativas de retransmissão da mensagem final (`COMPLETED` ou `ERROR`), o *rover* ainda não recebeu qualquer ACK da Nave-Mãe, este assume autonomamente que a comunicação foi perdida e encerra unilateralmente a sessão. Esta decisão impede que o *rover* fique bloqueado indefinidamente à espera de uma confirmação que pode nunca chegar.

  O número elevado de tentativas (15 para mensagens críticas) garante que apenas em falhas de comunicação severas e prolongadas o *rover* procede ao encerramento unilateral.

- **Gestão de Sessões Órfãs no Servidor**

  Do lado da Nave-Mãe, existe um mecanismo complementar de deteção e limpeza de sessões órfãs que previne inconsistências quando o *rover* encerra a sessão mas a Nave-Mãe não recebeu a mensagem final.

  O método `limparSessoesOrfas()`, executado periodicamente a cada 10 segundos, verifica:

  1. **Sessões Inativas**: Se uma sessão não apresenta atividade (mensagens recebidas ou enviadas) há mais de 30 segundos, é considerada órfã e removida automaticamente.
  2. **Inconsistências de Estado**: Se um *rover* está marcado como `DISPONIVEL` mas ainda possui uma sessão ativa registada, esta inconsistência é detetada e a sessão é removida, evitando bloqueios na atribuição de novas missões.
  3. ***Rovers* Desconectados**: Se o *rover* associado a uma sessão não existe mais no registo de *rovers* ativos (desconexão do *TelemetryStream*), a sessão correspondente é imediatamente removida.

  Este mecanismo de limpeza garante que, mesmo quando a comunicação UDP falha completamente durante o encerramento da sessão, o sistema recupera automaticamente dentro de um intervalo máximo de 30 segundos, libertando recursos e permitindo que o *rover* (se ainda operacional) receba novas missões.

Cenários de Encerramento

O protocolo suporta três cenários distintos de encerramento:

- **Encerramento Normal**: O *rover* envia `COMPLETED/ERROR`, recebe ACK final da Nave-Mãe, e ambos removem a sessão sincronizadamente. Este é o caso mais comum e eficiente.
- **Encerramento por Timeout do *Rover***: O *rover* esgota todas as retransmissões sem receber ACK, encerra unilateralmente a sessão, mas a Nave-Mãe deteta a inatividade e remove a sessão órfã dentro de 30 segundos.
- **Encerramento por Falha Total**: O *rover* deixa de responder completamente (falha crítica ou desconexão). A Nave-Mãe deteta ausência de telemetria (via TCP), marca o *rover* como desconectado e remove todas as sessões associadas, revertendo missões pendentes para o estado `PENDENTE`.

Esta arquitetura multi-camada de encerramento garante robustez do sistema mesmo em condições extremas de falha de comunicação, prevenindo *deadlocks* e garantindo que o estado global permanece eventualmente consistente.

4 API de Observação

4.1 Modelo de Exposição Estado

A API de Observação foi estruturada como uma camada exclusivamente dedicada à leitura do estado interno da Nave-Mãe. A classe `ObservacaoAPI` funciona como fachada sobre a classe `GestaoEstado`, garantindo que o *Ground Control* acede apenas a métodos controlados e nunca às estruturas internas diretamente. Esta decisão assegura isolamento, evita acoplamento e preserva a consistência do estado durante a execução da simulação.

4.2 Servidor HTTP

Optou-se pelo servidor nativo do Java, `HttpServer`, configurado para ouvir no endereço `0.0.0.0`. Esta decisão técnica foi essencial para a flexibilidade dos testes uma vez que permite que o sistema funcione em *localhost* durante o desenvolvimento e permite acesso via rede simulada (10.0.0.1) dentro do ambiente CORE, sem alterar o código.

Os *handlers* foram definidos manualmente, `handleRovers`, `handleTelemetria`, para manter o código explícito.

4.3 Serialização Manual para JSON

A construção das respostas JSON foi deliberadamente implementada manualmente através da classe `CriarJson`, utilizando *StringBuilder*. Esta decisão fundamentou-se na necessidade de controlo total sobre o formato final das respostas e da eficiência na serialização de estruturas simples e estáticas.

A implementação inclui normalização explícita de estados passando pela remoção de prefixos internos, pelo *escaping* de caracteres especiais e pela gestão rigorosa da estrutura dos objetos `Rover, Missao, PayloadProgresso, PayloadTelemetria`. Este processo assegura que o formato exposto ao *Ground Control* permanece estável e coerente, independentemente da representação interna.

4.4 *Parsing* Manual de Missões

O *endpoint* responsável pela criação de novas missões exigiu a implementação de um *parser* JSON manual. Optou-se por:

- remover chaves estruturais do JSON recebido,
- dividir pares chave-valor respeitando conteúdos entre aspas,
- validar tipos e valores,
- construir um objeto `Missao` internamente consistente.

Esta abordagem mantém a uniformidade com a restante estrutura minimalista da API e evita bibliotecas externas. Além disso, permite validar rigorosamente cada campo antes da inserção no estado global, prevenindo dados inválidos ou inconsistentes.

4.5 Integração com a UI (CORS e Ficheiros Estáticos)

Para que a interface Web (*Ground Control*) funcione sem falhas, implementaram-se duas funcionalidades críticas no servidor:

Suporte a CORS

O servidor injeta cabeçalhos (`Access-Control-Allow-Origin: *`) em todas as respostas. Isto permite que o *browser* aceite dados da API mesmo que a página esteja aberta num contexto diferente.

Servidor de Ficheiros Estáticos

A classe `StaticFileHandler` carrega a interface diretamente dos recursos do JAR (`src/main/resources/ui`). Ela deteta automaticamente o tipo de ficheiro (MIME *type*), diferenciando CSS de JavaScript — para que o navegador renderize a página corretamente sem necessidade de instalar um servidor web externo.

5 *Ground Control*

Para a concretização das responsabilidades atribuídas ao módulo Ground Control, foram tomadas decisões técnicas específicas visando a autonomia da aplicação, a redução de dependências externas e a facilidade de validação dos dados.

5.1 Interface de Interação em Modo Texto (CLI/TUI)

Optou-se, inicialmente, pela implementação de uma interface baseada no terminal, estruturada através de menus interativos, em detrimento de uma interface gráfica (GUI). Esta decisão, visível na classe `GroundControlApp`, permitiu focar o desenvolvimento na correção da lógica de comunicação e na visualização imediata dos dados brutos recebidos.

A interação é síncrona: o operador solicita uma ação, como listar *rovers* ou criar missão, e a aplicação bloqueia até obter a resposta da API, garantindo que a informação apresentada é sempre o resultado da transação mais recente

5.2 Cliente HTTP

Para a camada de comunicação com a Nave-Mãe, explicita na classe `GroundControlAPI`, decidiu-se utilizar exclusivamente as bibliotecas padrão do Java (java.net.HttpURLConnection e java.io), evitando o uso de *frameworks* externos.

Esta abordagem reduz a complexidade de compilação e execução do projeto, eliminando a necessidade de gestores de dependências adicionais para este módulo.

Foram, então, criados métodos utilitários genéricos, como GET e POST, que encapsulam a abertura de conexões, a configuração de cabeçalhos e o tratamento de erros HTTP, lançando exceções controladas que são tratadas na camada de interface.

Assim, o módulo `GroundControlAPI` encapsula todas as interações com o servidor, isto é, a obtenção de *rovers* e missões, a consulta de progresso e de telemetria (*live* e histórico) e a criação de novas missões via POST.

5.3 *Parsing* JSON

Uma das decisões mais distintivas desta implementação foi a criação de um *parser* JSON manual, classe `ParserJson`, em vez de recorrer a bibliotecas de serialização robustas como Jackson ou Gson.

O *parser* foi desenhado especificamente para a estrutura plana dos modelos do projeto: RoverModel, MissaoModel, TelemetriaModel e ProgressoModel. Utiliza manipulação direta de strings para extrair pares chave-valor e converte-os para os tipos de dados Java apropriados.

5.4 Modelo de Criação de Missões

Embora o enunciado refira o *Ground Control* primariamente como uma interface de observação, decidiu-se implementar também a capacidade ativa de comando através da criação de missões.

Desta forma, a aplicação recolhe os parâmetros da missão como coordenadas, prioridade, tarefa, ... através de inputs sequenciais do utilizador. De seguida instancia o `MissaoModel` localmente e serializa-o manualmente para uma string JSON antes de efetuar o POST para o *endpoint* /missoes. Isto centraliza a validação do formato dos dados no cliente antes do envio para a Nave-Mãe.

5.5 Interface Gráfica Web

Complementarmente à interface de consola, desenvolveu-se uma interface gráfica Web para permitir uma visualização panorâmica do estado da missão. Esta decisão visou colmatar a dificuldade de leitura de múltiplas linhas de texto em tempo real na consola.

Optou-se deliberadamente pela não utilização de *frameworks* de *frontend*. A aplicação foi construída utilizando HTML, CSS e JavaScript puro. Esta abordagem reduziu drasticamente o *overhead* de recursos e eliminou a necessidade de processos de *build*, facilitando a implantação direta no ambiente emulado CORE, onde os recursos são limitados e a simplicidade de configuração é prioritária.

Estratégia de Atualização por Polling

Em alternativa aos *WebSockets*, a aplicação utiliza um ciclo de atualização baseado em `setInterval`, definido para 5000ms em `app.js`, executando pedidos *fetch* assíncronos aos *endpoints* da API.

Embora os *WebSockets* permitissem latência menor, a arquitetura REST da Nave-Mãe favorece o modelo pedido-resposta. A estratégia de *polling* revelou-se eficiente para a cadência de atualização dos *Rovers* e simplificou o tratamento de erros de conexão.

Deteção Automática de Ambiente

Foi implementado um mecanismo de deteção de *host* no arranque da aplicação. O *script* verifica se está a correr num ambiente local, `localhost`, numa rede emulada ou se existe um *override* via parâmetros de URL.

Isto permite utilizar exatamente o mesmo código fonte tanto durante o desenvolvimento na máquina hospedeira como dentro dos nós virtuais do CORE, sem necessidade de reconfiguração manual de IPs.

5.6 Gestão de Estado no Cliente

Tendo em conta que a API é *stateless* e apenas devolve o estado atual, o *frontend* mantém uma cópia local do estado anterior. Isto permite comparar o ciclo atual com o anterior para gerar notificações de eventos, como "*Rover* iniciou missão" ou "Bateria crítica", e alimentar o "Registo de Atividades" visual, algo que a API não fornece diretamente.

6 Ambiente de Testes

6.1 Ferramentas de Diagnóstico

Para complementar a validação funcional no ambiente emulado, foi utilizada uma ferramenta de diagnóstico dedicada, `test-api.html`. Trata-se de um cliente HTTP minimalista que permite a invocação direta e isolada dos *endpoints* da Nave-Mãe (`/rovers`, `/missoes`, `/telemetria`).

A sua utilização no ambiente de testes serviu dois propósitos essenciais de validação:

1. **Inspeção de Dados em Bruto**: Permitiu visualizar a resposta JSON crua e os cabeçalhos HTTP exatos retornados pelo servidor, validando a correção do *parser* manual sem a interferência da lógica de renderização do *Dashboard*.
2. **Verificação de Conectividade e CORS**: Ao ser executada em diferentes nós da topologia, a ferramenta validou o comportamento das políticas de segurança do navegador (CORS) e a acessibilidade da API através da rede simulada

6.2 Testes unitários

Para testar o módulo do Serializador, criamos uma série de testes unitários que validam a fragmentação. Foram úteis para encontrar erros ao fragmentar e reconstruir as mensagens, assim como ajudou no desenvolvimento do módulo. Foram implementados testes organizados em cinco categorias:

- **Serialização de Payloads** - Verifica que todos os campos de `PayloadMissao`, `PayloadProgresso`, `PayloadAck` e `PayloadErro` são corretamente convertidos.
- **Fragmentação**- Valida que mensagens grandes são divididas em fragmentos e que payloads pequenos não são fragmentados desnecessariamente.
- **Agregação e Reconstrução** - Garante que fragmentos recebidos fora de ordem são corretamente recompostos, preservando a integridade dos dados originais.
- **Verificação de Completude** - Assegura a deteção de mensagens incompletas antes da reconstrução.
- **Operações Auxiliares** - Testa serialização binária, tratamento de erros e limpeza de estado.

6.3 Testes em ambiente CORE

Para testar os protocolos, utilizamos a topologia no ambiente CORE, simulando diversos cenários com diferentes valores de perdas, latência e duplicados. Durante a implementação estes testes ajudaram a afinar os mecanismos de controlo, mostrando casos vulneráveis que iam sendo corrigidos. Por fim, são também uma boa ferramenta para validar o protocolo implementado, mostrando a sua fiabilidade. Para relatórios mais completos sobre estes testes, existe ainda um módulo que guarda as métricas sobre as comunicações durante a execução e guarda em ficheiro no fim, permitindo uma análise posterior.

Para estes testes foram usadas, além da topologia base, várias variantes da mesma, nomeadamente: test-dup.xml (10% duplicados); test-low-loss.xml (5% perda, 50ms *delay*); test-medium-loss.xml (10% perda, 100ms *delay*); test-high-loss.xml (20% perda, 200ms *delay*); test-only-loss.xml (5% perda)

Chapter 5

Resultados dos Testes

Os testes realizados foram agrupados por tipo de comportamento, nomeadamente concorrência, duplicação de pacotes, perdas de mensagens e mecanismos de recuperação e robustez.

1 Testes de Concorrência e Execução em Paralelo

Este conjunto de testes valida a capacidade da Nave-Mãe em lidar com múltiplos *rovers* a operar em simultâneo, garantindo que missões e telemetria são processadas corretamente sem interferências entre sessões paralelas.

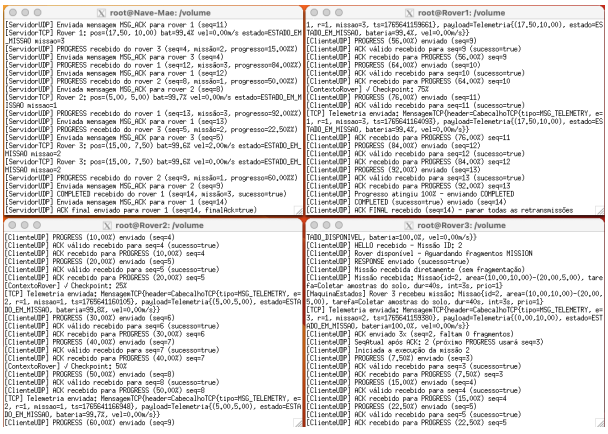

Figure 5.1: Execução simultânea de múltiplos *rovers*

2 Testes com Duplicação de Pacotes (DUP = 10%)

Nestes testes é introduzida duplicação de pacotes em todos os links para verificar se o sistema mantém consistência, ignora mensagens redundantes e preserva a sequência correta de comunicação.

```
[ClienteUDP] ACK recebido para PROGRESS (76.00%) seq=11
[ClienteUDP] PROGRESS (84.00%) enviado (seq=12)
[ClienteUDP] ACK válido recebido para seq=12 (sucesso=true)
[ClienteUDP] ACK inesperado para seq=12 (esperado=12)
[ClienteUDP] ACK recebido para PROGRESS (84.00%) seq=12
[ClienteUDP] PROGRESS (92.00%) enviado (seq=13)
[ClienteUDP] ACK válido recebido para seq=13 (sucesso=true)
[ClienteUDP] ACK recebido para PROGRESS (92.00%) seq=13
[ClienteUDP] Progresso atingiu 100% - enviando COMPLETED
[ClienteUDP] COMPLETED (sucesso=true) enviado (seq=14)
[ClienteUDP] ACK FINAL recebido (seq=14) - parar todas as retransmissões
[ClienteUDP] ACK recebido para COMPLETED (sucesso=true) seq=14
[ClienteUDP] Missão 3 concluída
[ContextoRover] Rover #1 agora disponível
```

Figure 5.2: Duplicação de ACK sem quebra da sequência

```
[ServidorUDP] COMPLETED recebido para rover 1
[ServidorTCP] Rover 1: pos=(15.00, 7.50) bat=98.8% vel=0.00m/s estado=ESTADO_DIS
PONIVEL missao=-1
[ServidorUDP] Enviada mensagem MSG_ACK para rover 1 (seq=15)
[ServidorUDP] ACK final enviado para rover 1 (seq=15, finalAck=true)
[ServidorUDP] Enviada mensagem MSG_ACK para rover 1 (seq=15)
[ServidorUDP] ACK final enviado para rover 1 (seq=15, finalAck=true)
[ServidorUDP] Sessão do rover 1 removida
[ServidorUDP] Sessão de missão 2 para rover 1 concluída com sucesso
```

Figure 5.3: Receção duplicada de COMPLETED sem impacto no sistema

```
[ServidorUDP] Enviada mensagem MSG_HELLO para rover 1 (seq=1)
[ServidorUDP] RESPONSE recebido do rover 1 (sucesso=true, seq=1)
[ServidorUDP] RESPONSE duplicado ignorado do rover 1
[ServidorUDP] RESPONSE duplicado ignorado do rover 1
[ServidorUDP] Missão 2 não precisa de fragmentação - enviando diretamente
[ServidorUDP] Enviada mensagem MSG_MISSION para rover 1 (seq=2)
[ServidorUDP] ACK recebido do rover 1 (faltam 0 fragmentos, seq=2)
[ServidorUDP] Missão 2 enviada com sucesso para rover 1
```

Figure 5.4: Servidor a ignorar respostas duplicadas

3 Testes de Perda de Mensagens e Retransmissão

Aqui é avaliada a robustez do sistema perante perdas de mensagens, verificando se os mecanismos de retransmissão permitem a conclusão correta das missões e o progresso adequado do estado.

Figure 5.5: Perda da mensagem COMPLETED

```
[ServidorUDP] Enviada mensagem MSG_HELLO para rover 1 (seq=1)
[ServidorUDP] RESPONSE recebido do rover 1 (sucesso=true, seq=1)
[ServidorUDP] Missão 1: 748 bytes em 2 fragmentos (máx 512 bytes cada)
[ServidorUDP] Enviada mensagem MSG_MISSION para rover 1 (seq=2)
[ServidorUDP] Enviada mensagem MSG_MISSION para rover 1 (seq=3)
[ServidorUDP] Timeout aguardando ACK - retransmitindo missão completa (tentativa
 2/3)
[ServidorUDP] Missão 1: 748 bytes em 2 fragmentos (máx 512 bytes cada)
[ServidorUDP] Enviada mensagem MSG_MISSION para rover 1 (seq=2)
[ServidorUDP] Enviada mensagem MSG_MISSION para rover 1 (seq=3)
[ServidorUDP] ACK recebido do rover 1 (faltam 0 fragmentos, seq=3)
[ServidorUDP] Missão 1 enviada com sucesso para rover 1
```

Figure 5.6: Perda e reenvio de uma missão completa

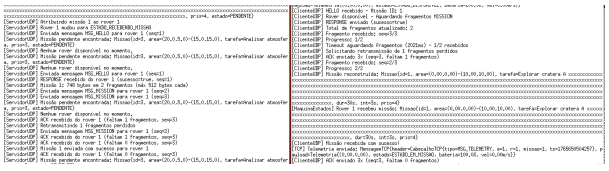

Figure 5.7: Perda e reenvio de fragmentos de uma missão

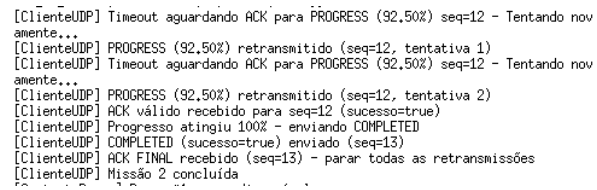

Figure 5.8: Retransmissão de PROGRESS após perda

4 Testes de Gestão de Sessões

Estes testes focam-se na correta gestão do ciclo de vida das sessões, assegurando que sessões órfãs não são removidas prematuramente.

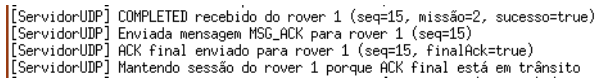

Figure 5.9: Gestão correta de sessões órfãs

5 Testes de Reconexão

Este teste demonstra a capacidade de reconexão no protocolo TelemetryStream.

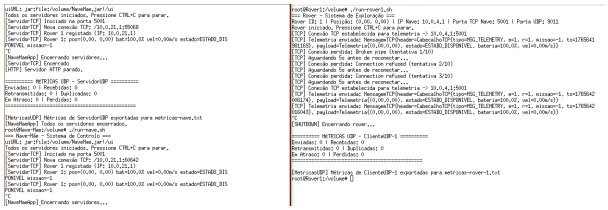

Figure 5.10: Reconexão TCP sem perda de estado

Chapter 6

Desafios Encontrados e Soluções

A decisão de não utilizar *frameworks* web nem bibliotecas de serialização impôs desafios técnicos significativos na construção do servidor HTTP:

Processamento de JSON (*Parsing*) sem Bibliotecas

O Java não possui mecanismos nativos simples para converter uma string JSON num objeto. Receber uma nova missão via POST tornava-se complexo, pois era necessário extrair campos como coordenadas e prioridade de um texto ”bruto”, lidando com aspas, vírgulas e formatação variável.

Desta forma, implementou-se um método de *parsing* manual na classe `ServidorHTTP`. A solução limpa caracteres estruturais como chavetas e quebras de linha, e divide o conteúdo por vírgulas, iterando sobre os pares chave-valor. Foram adicionadas validações defensivas para garantir que erros de formato ou números inválidos não interrompem a execução do servidor, devolvendo mensagens de erro HTTP 400 claras ao cliente.

Restrições de Segurança do Navegador (CORS)

Durante o desenvolvimento e testes, ao tentar aceder à API a partir de uma interface web alojada localmente (ou num porto diferente), os navegadores bloqueavam os pedidos devido às políticas de segurança *Cross-Origin Resource Sharing* (CORS), impedindo a comunicação entre o *Ground Control* e a Nave-Mãe.

Injeção manual de cabeçalhos de permissão em todas as respostas HTTP. O servidor foi configurado para adicionar *Access-Control-Allow-Origin:* * em cada resposta e, especificamente, para intercetar e responder com sucesso (código 204) a todos os pedidos do tipo OPTIONS, que os navegadores enviam automaticamente antes de fazerem o pedido real.

A implementação da interface Web impôs desafios específicos relacionados com a natureza *stateless* da comunicação e o desempenho no navegador:

Ausência de Eventos em API *Stateless*

Como a API não suporta *push notifications*, era difícil detetar transições de estado, como conclusão de missão, em tempo real.

Assim, implementou-se um algoritmo de comparação de estado no cliente, em app.js. O sistema compara, em `detectRoverChanges` e `detectMissionChanges`, o JSON atual com o anterior e gera eventos sintéticos locais quando deteta alterações relevantes nos dados.

Sobrecarga de Renderização - Telemetria

A visualização contínua de dados de telemetria tendia a degradar a performance do navegador devido ao crescimento infinito dos elementos no DOM.

Assim, criou-se uma *sliding window* que limita a renderização aos últimos 20 registos, descartando automaticamente dados antigos para manter a fluidez da interface

Feedback de Conectividade

Em um ambiente simulado com perda de pacotes, a interface não distinguia claramente entre ”ausência de dados novos” e ”falha de conexão”.

Desta forma, desenvolveu-se um sistema visual de monitorização de rede. O `httpWrapper` interceta falhas nas *Promises* e altera imediatamente o estado da interface para ”Offline”, revertendo apenas quando um pedido é bem-sucedido.

Ambiguidade entre Erros de Renderização e Falhas de Rede

Durante a integração da interface gráfica com a Nave-Mãe, ocorriam frequentemente situações em que os dados não apareciam no ecrã. Era complexo determinar se a falha residia na lógica de apresentação do *Dashboard* ou se a API estava a devolver dados corrompidos/inacessíveis. O *feedback* visual do Dashboard era demasiado abstrato para diagnósticos técnicos precisos.

Desta forma, decidimos desenvolver uma Ferramenta de Diagnóstico Isolada, exposta em *test-api.html*.

Assim, criamos uma página HTML minimalista, isenta de qualquer lógica de negócio ou estilos complexos, contendo apenas botões para invocar diretamente os *endpoints* /rovers, /missoes e /telemetria/historico.

A ferramenta exibe o texto bruto recebido do servidor antes de tentar qualquer processamento. Isto permitiu detetar imediatamente problemas invisíveis no *Dashboard* principal, como JSON malformado, cabeçalhos HTTP incorretos ou respostas vazias.

Sendo um ficheiro único sem dependências externas, pôde ser facilmente transferido e executado dentro dos diferentes nós da rede emulada (CORE) para testar a conectividade local de cada componente.

Durante a implementação do MissionLink, enfrentamos alguns desafios difíceis:

Sincronização de Mensagens

Foi necessário garantir que as mensagens entre os módulos fossem entregues de forma ordenada e sem perdas. Para isso, implementamos mecanismos de controlo para confirmar a recessão e retransmitir pacotes. Estes mecanismos são de extrema importância para o bom funcionamento do protocolo, encontrando-se mais explorados em detalhe no capítulo das decisões implementadas.

Gestão de Conexões

A manutenção das várias conexões UDP em paralelo exigiu um tratamento diferente. Solucionamos isso com a estrutura de sessão para cada comunicação onde se guardam as informações necessárias ao bom-funcionamento de cada comunicação do ínício ao fim. Esta solução demonstrou-se também útil ao longo da implementação como apoio à adição de mecanismos de controlo do protocolo.

Serialização e Fragmentação de Dados

A troca de dados entre diferentes módulos exigiu a padronização dos formatos de mensagens. Utilizamos classes específicas para serializar e desserializar os dados, garantindo compatibilidade e integridade das informações. Além disso a própria fragmentação das mensagens foi bastante repensada, tendo passado por diversas tentativas diferentes de implementação até chegar á solução encontrada que temos no momento.

Identificação de problemas

Para facilitar a identificação de problemas durante a implementação, incluimos logs detalhados, tendo sido de grande ajuda e permitindo uma depuração mais eficiente durante o desenvolvimento e testes.

Execução no CORE

No início tivemos alguma dificuldade em executar o trabalho no CORE, por exemplo, não conseguíamos executar Java. Para ultrapassar esse obstáculo, tivemos que modificar o docker-compose e adicionámos um DockerFile para que tivéssemos disponíveis as ferramentas de que precisávamos dentro do CORE.

Chapter 7

Possíveis Melhorias Futuras

Embora o sistema desenvolvido cumpra os requisitos estabelecidos e demonstre robustez nas comunicações, foram identificadas várias oportunidades de melhoria que poderiam aumentar a sua eficiência, escalabilidade e funcionalidades.

No âmbito das otimizações de protocolo, seria vantajoso implementar um mecanismo de controlo de congestão adaptativo, com ajuste dinâmico dos *timeouts* e intervalos de retransmissão baseados nas condições da rede, à semelhança do TCP *Congestion Control*. Atualmente, o envio de mensagens PROGRESS utiliza um mecanismo *stop-and-wait*, no qual cada mensagem aguarda confirmação antes de enviar a seguinte.

Relativamente à fiabilidade do sistema, quando o número máximo de retransmissões é atingido, o *rover* assume a perda de conexão, mas não verifica se a Nave-Mãe continua operacional. A implementação de um mecanismo de *heartbeat* permitiria distinguir entre perdas de pacotes pontuais e falhas totais do servidor, melhorando a gestão de erros. A persistência de estado seria igualmente uma adição valiosa, permitindo recuperar o estado das missões em caso de reinício inesperado e garantindo a continuidade operacional. Embora o UDP forneça verificação básica de integridade, a adição de *checksums* complementares aos *payloads* críticos aumentaria a deteção de corrupção de dados não identificada pela camada de transporte.

No que respeita à interoperabilidade, a implementação atual utiliza serialização nativa de Java, tornando o protocolo dependente da JVM e incompatível com clientes implementados noutras linguagens de programação. A adoção de formatos de serialização independentes de linguagem, facilitaria a integração do sistema em ambientes heterogéneos onde os diversos componentes podem ser implementados em tecnologias distintas. Esta alteração, embora exija o redesenho do módulo `SerializadorUDP`, tornaria o sistema significativamente mais flexível, permitindo que diferentes equipas desenvolvam módulos de forma independente e garantindo a interoperabilidade entre componentes implementados em linguagens diversas.

Quanto a funcionalidades adicionais, a priorização de mensagens por nível de criticidade (ERRO > COMPLETED > PROGRESS) permitiria um tratamento diferenciado em situações de congestionamento de rede. Atualmente, as mensagens PROGRESS não utilizam fragmentação devido ao seu tamanho reduzido (16 bytes), mas o suporte para fragmentação destas mensagens permitiria, no futuro, o envio de informação enriquecida. A implementação de encriptação das comunicações e de mecanismos robustos de autenticação e autorização seria essencial em ambientes onde a segurança das missões fosse crítica.

Estas melhorias, embora não essenciais para o funcionamento básico do sistema, aumentariam significativamente a sua robustez, eficiência e adequação a cenários de produção reais com requisitos mais exigentes de desempenho, segurança e fiabilidade.

Chapter 8

Conclusão

O presente trabalho permitiu desenvolver e implementar com sucesso um ecossistema distribuído de comunicações, cumprindo os objetivos propostos no âmbito da unidade curricular de Comunicações por Computadores. Através da conceção e implementação das três entidades principais *Rovers*, Nave-Mãe e *Ground Control* foi possível consolidar conhecimentos teóricos e práticos relativos a protocolos de comunicação distintos, nomeadamente TCP e UDP.

A implementação dos dois canais de comunicação entre a Nave-Mãe e os *rovers*, o *MissionLink* (ML) baseado em UDP e o *TelemetryStream* (TS) baseado em TCP, demonstrou a importância da escolha adequada de protocolos consoante os requisitos específicos de cada funcionalidade. O UDP com mecanismos de fiabilidade (ACK, retransmissões e deteção de perdas) revelou-se eficaz para a transmissão de missões e atualizações de progresso, oferecendo um equilíbrio entre controlo de fluxo e baixa latência. Por sua vez, o TCP mostrou-se apropriado para o fluxo contínuo de telemetria através de *streams*, garantindo a entrega ordenada e fiável dos dados de monitorização.

A comunicação entre o *Ground Control* e a Nave-Mãe através de HTTP/REST permitiu uma interface de controlo intuitiva e acessível, facilitando a gestão centralizada das missões e visualização do estado do sistema em tempo real.

As decisões de implementação tomadas ao longo do desenvolvimento foram fundamentadas nos requisitos especificados no enunciado e nas boas práticas de engenharia de *software*, privilegiando a modularidade, escalabilidade e manutenibilidade do código. A arquitetura modular com separação clara entre camadas de comunicação (UDP/TCP), serialização de dados e lógica de aplicação facilitou o desenvolvimento incremental do sistema. Os desafios encontrados durante a implementação nomeadamente a gestão de fragmentação de mensagens, sincronização de estados entre entidades distribuídas, tratamento robusto de perdas de pacotes e garantia de entrega de mensagens críticas (COMPLETED/ERROR) contribuíram significativamente para o aprofundamento dos conhecimentos na área das comunicações por computador.

Este projeto constituiu uma oportunidade valiosa para aplicar conceitos teóricos em contexto prático, desenvolvendo competências essenciais para a futura atividade profissional enquanto engenheiros informáticos, nomeadamente no que respeita ao desenho e implementação de protocolos robustos e eficientes.