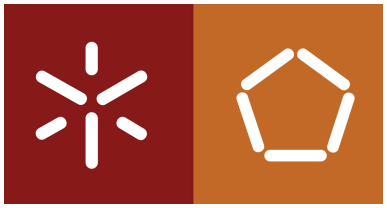


Comunicações por Computadores - Trabalho prático II

NASUM

30.01.2026

PL611: Gabriel Dantas (a107291), José Fernandes (a106937), Simão Oliveira (a107322)
Universidade do Minho - Licenciatura em Engenharia Informática, Comunicações por Computadores

Índice

1. Introdução **1**

2. Idealização protocolar do MissionLink **2**

2.1. Estrutura base do pacote 2

2.2. Comunicação de uma missão 3

2.3. Comunicação de um relatório 4

2.4. Funcionamento de números de sequência e reconhecimento 4

2.5. Início de missões e de conexão 5

2.6. Informação de relatórios 6

2.7. Outras propriedades relevantes do protocolo 6

3. Protocolo TelemetryStream **8**

4. Definição das tarefas **9**

5. Funcionamento da Nave-mãe **11**

5.1. Configurações iniciais 11

5.2. Gestão de missões 12

5.3. Packet Manager 12

5.4. Receção de pacotes 13

5.5. Tratamento de reports 13

5.6. Protocolo TelemetryStream 14

6. Funcionamento do Rover **15**

6.1. Configurações iniciais 15

6.2. Estabelecimento de conexões 15

6.3. Gestão, pedido e execução de missões 15

6.4. Receção de pacotes 16

6.5. TelemetryStream 16

6.6. Gestão de bateria e suspensão 16

6.7. Packet Manager e Window 16

6.8. Concorrência no Rover 17

7. Ground Control **18**

7.1. API de Observação 18

7.2. Nó Ground Control 18

8. Testes e Resultados **20**

8.1. Testes em diferentes topologias 20
8.1.1. Ambiente A - Ideal 20
8.1.2. Ambiente B - Realista 22
8.1.3. Ambiente C - Péssimo 24

9. Conclusão: Observações Finais **27**

1. Introdução

O presente relatório foi how to take printscree linux do trabalho prático II de Comunicações por Computadores. Neste, era pedido um sistema distribuído de comunicação no contexto de uma missão espacial, incidindo especialmente sobre a comunicação protocolar entre os pontos do sistema, a nave-mãe e os *rovers*.

Apresentar-se-ão e explicar-se-ão com maior enfoque as estruturas de dados relevantes para a comunicação e funcionamento interno dos pontos terminais do sistema, o protocolo desenvolvido sobre UDP para suportar uma comunicação fiável e eficiente e a arquitetura geral do sistema. Alguns outros pormenores que consideramos pertinentes serão também abordados ao longo do documento. Apresentar-se-ão ainda testes para comprovar a eficácia e correção do sistema. Por fim, visa-se uma reflexão sobre o trabalho desenvolvido.

2. Idealização protocolar do MissionLink

Nesta seção o objetivo é explicar, de uma forma abstrata, o protocolo que foi implementado sobre UDP. A decisão de fazer esta seção vem da importância que a rigidez da definição deste componente tem no desenvolvimento posterior do trabalho.

2.1. Estrutura base do pacote

O protocolo *MissionLink* utiliza uma estrutura de pacote base *Packet* para encapsular todos os tipos de comunicações que foram definidos. Este pacote é projetado para ser pequeno e robusto sobre UDP.

Este tem os seguintes campos, apresentados abaixo, em Código 1. Para garantir ordenação temos, naturalmente, um número de sequência. Já para garantir a fiabilidade da receção teremos um número de reconhecimento. Apenas o payload é variável.

```
type PacketType uint8
type Packet struct {
  RoverId  uint8 // Identificado único do Rover de origem/destino
  MsgType  PacketType // Tipo de mensagem
  SeqNum   uint32 // Número de sequência da mensagem
  AckNum   uint32 // Número de reconhecimento
  Checksum uint8 // Soma simples para deteção de erros
  Payload  []byte // Conteúdo da mensagem
}
```

Código 1: Estrutura Packet.

O campo *MsgType* define a finalidade do pacote, permitindo ao recetor saber como tratar o pacote e como interpretar o seu *payload*. Os tipos de pacotes existentes na nossa comunicação são os que se mostram no Código 2. Explicam-se abaixo.

- *Request*: Este tipo de pacote é um pedido de missões à nave-mãe por parte de um rover;
- *Mission*: Este tipo de pacote define uma missão e é enviada da nave-mãe para o rover saber onde, como e qual tarefa deve realizar;
- *No_Mission*: Este tipo de pacote é enviado pela nave-mãe ao *rover* quando não existem missões a ser executadas após uma *request*;
- *Report*: Um *report* define a informação enviada pelo *rover* durante e como conclusão da execução de uma tarefa à nave-mãe;
- *ACK*: Um *ACK* é um pacote que tem como único intuíto transmitir o *acknowledgement number* de uma mensagem.

```
const (
    MSG_MISSION    PacketType = iota
    MSG_NO_MISSION
    MSG_ACK
    MSG_REPORT
    MSG_REQUEST
)
```

Código 2: Definição do tipo dos pacotes.

2.2. Comunicação de uma missão

Uma missão usa como *payload* uma estrutura *MissionData* de tamanho fixo de 27 bytes para informar a missão ao rover com toda a informação necessária para este a interpretar.

```
type MissionData struct {
  MsgID uint16 // Identificado único da mensagem [0-65535]
  Coordinate utils.Coordinate //Coordenadas onde a missão deve ser
executada [Latitude:float64, Longitude:float64]
  TaskType uint8 // Tipo de tarefa a executar
  Duration uint32 // Duração de uma missão em segundos
  UpdateFrequency uint32 // Frequência a que o rover deve reportar
atualizações sobre a missão em segundos
  Priority uint8 // Prioridade da missão [1 a 3]
}
```

Código 3: Definição de *MissionData*.

Salienta-se qu,e para os campos *Duration* e *UpdateFrequency*, foi escolhido o tipo *uint32*, pois considerando missões espaciais realistas, podem existir missões com períodos de anos. Fazendo as contas, um *uint32* de segundos, consegue representar, aproximadamente, 136 anos, como se demonstra abaixo:

$\frac{2^{32}}{60*60*24*365} \approx 136.192$ anos

Achamos relevante referir ainda que esta estrutura tem um tamanho fixo de 27 bytes, pois quando é serializada mesclam-se o *TaskType* e a *Priority* no mesmo byte, visto ambos conseguirem cumprir a sua função com apenas 4 bytes e ainda sobrar um byte de folga no caso de modificações futuras que possam envolver expansões do sistema (demonstra-se a função de serialização no Código 4).

```
func (d *MissionData) Encode() []byte {
  data := make([]byte, MissionDataSize)
  binary.BigEndian.PutUint16(data[0:], d.MsgID)
  binary.BigEndian.PutUint64(data[2:],
      math.Float64bits(d.Coordinate.Latitude))
  binary.BigEndian.PutUint64(data[10:],
      math.Float64bits(d.Coordinate.Longitude))
 // taskTypeAndPriority combina TaskType (4 bits superiores) e Priority
  data[18] = (d.TaskType << 4) | (d.Priority & 0x0F)
  binary.BigEndian.PutUint32(data[19:], d.Duration)
  binary.BigEndian.PutUint32(data[23:], d.UpdateFrequency)
  return data
}
```

Código 4: Serialização de *MissionData*.

2.3. Comunicação de um relatório

Um relatório (*report*) tem uma estrutura simples de duas componentes apenas: o *header*, parte fixa da estrutura e necessária para qualquer missão (contém informação do *Id* da missão, do tipo de tarefas e um booleano que informa se este é ou não o último *report* da sequência) e o *payload*, que carregará os dados relativos a cada tipo de missão diferente. A estrutura apresenta-se no Código 5.

```
type ReportHeader struct {
    TaskType     uint8
    MissionID    uint16
    IsLastReport bool
}
    type Report struct {
    Header  ReportHeader
    Payload []byte
}
```

Código 5: Estrutura de *Report*.

Assim, temos uma estrutura versátil, que permite uma desserialização polimórfica eficiente.

2.4. Funcionamento de números de sequência e reconhecimento

Definimos que iremos tratar os números de sequência e de reconhecimento de uma forma similar ao protocolo TCP.

Para garantir entrega ordenada e fiabilidade, temos um número de sequência que funciona cumulativamente e incremental pelo número de bytes do *payload* do pacote enviado no segmento anterior.

O número de reconhecimento serve para confirmar que os dados anteriores foram corretamente recebidos e solicitar novos dados. Implementámos *ack's* cumulativos.

Decidiu-se tratar as retransmissões da mesma maneira que o TCP. Se uma mensagem for perdida no caminho, o manuseamento de números de reconhecimento (*Ack*) garante que, ao enviar repetidamente o *Ack* do último byte recebido com sucesso, o emissor percebe que o pacote seguinte (o que falta) se perdeu e é forçado a retransmitir apenas esse pacote perdido, permitindo que a transmissão de dados continue a partir desse ponto. Já os pacotes duplicados são ignorados.

2.5. Início de missões e de conexão

É sempre um *rover* que envia um pacote de *request* à nave-mãe. Este pacote tem um inteiro que diz o número de missões que o *rover* aceita. A este pedido a nave-mãe responde com o número pedido de pacotes do tipo *mission*, que transmite os dados de uma missão, caso tenha missões em fila. Caso contrário, responde com um pacote do tipo *no mission*, informando o *rover* que não tem missões a distribuir e, neste caso, o *rover* aguarda um tempo até pedir uma nova missão. Na Figura 1 e Figura 2 representam-se estes dois possíveis fluxos de mensagem.

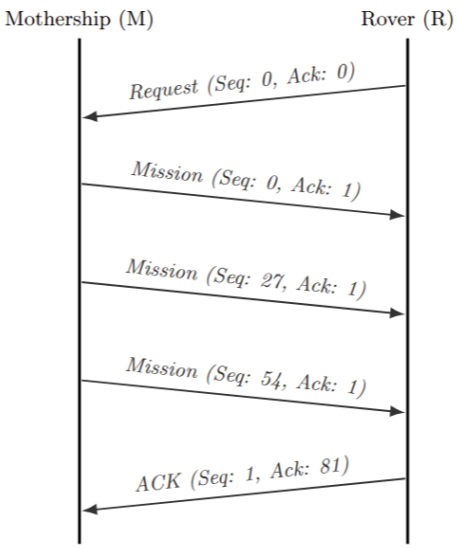


Figura 1: Processo de pedido de missões.

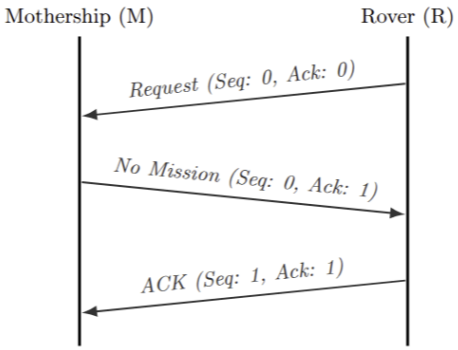


Figura 2: Processo de pedido de missões quando a nave-mãe não contém missões.

Não definimos o *three-way handshake* genérico como no protocolo TCP. Aplicando-o à realidade do nosso tipo de comunicação específico, podemos assumir que esta troca de mensagens, quando começadas com um *request* com número de sequência 0, corresponde ao início da sessão.

É importante notar que existe um tratamento especial de pacotes vazios. Pacotes sem *payload* (como *no mission*) são interpretados como tendo tamanho 1 tanto pelo emissor como pelo recetor. Esta convenção evita ambiguidades no cálculo dos números de sequência e garante sincronização consistente entre nave-mãe e rovers. Sem esta regra, pacotes vazios não avançariam o númerod e sequência, causando colisões de números de sequência em envios subsequentes. Excetua-se deste tratamento o pacote do tipo *ACK*, que não apresenta problema ser tratado

com tamanho de *payload* nulo, visto este não ter que esperar por uma confirmação do recetor.

2.6. Informação de relatórios

Durante a fase de troca de mensagens de relatórios a comunicação é simples de descrever.

Mostra-se na Figura 3 um exemplo abstrato da troca de mensagens entre nave-mãe e *rover* num momento de envio de *reports*.

Assuma-se como verdade os números de sequência iniciais de r e n para o *rover* e a nave-mãe, respetivamente. Assuma-se ainda o próximo valor esperado pelo *rover* como número de reconhecimento de m e, para qualquer missão, t_i o tamanho do *payload* do i-ésimo pacote do tipo *report* enviado pelo *rover*.

É de notar que, devido ao ACK ser um pacote tratado de forma especial e se um *payload*, o *acknowledgement number* do lado do *rover* não se altera.

Esta generalização é aplicável a qualquer tipo de missão com qualquer tamanho de pacotes (desde que contidos no tamanho máximo definido do campo *payload*), com tamanho variável ou não.

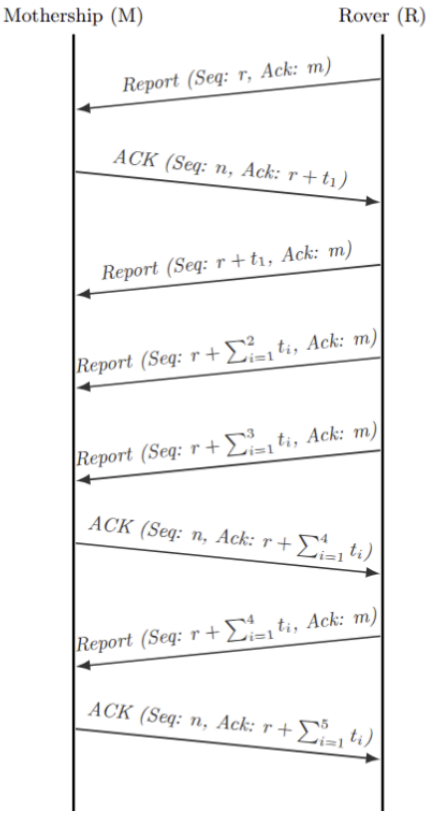


Figura 3: Troca de mensagens de *Report*.

2.7. Outras propriedades relevantes do protocolo

É de importância referir que foi implementado um sistema de RTO (Retransmission Timeout) dinâmico. Este é puramente matemático, seguindo a mesma estratégia do algoritmo TCP Karn's Algorithm.

Querendo optar por uma solução versátil de retransmissão rápida, decidiu-se implementar um mecanismo de fast retransmission, semelhante ao utilizado pelo TCP. Neste sistema, quando o recetor recebe um certo número (configurável) de *ACK's* duplicados, procede ao reenvio imediato do pacote correspondente, evitando assim a espera completa pelo tempo de RTO. Esta abordagem melhora significativamente a fluidez da comunicação e a coerência do processo de retransmissão.

Usou-se um *checksum* para verificação de integridade de pacotes, com uma estratégia modular de 8 bits.

É feito controlo de congestionamento definindo um valor máximo de pacotes "em voo", isto é, enviados, mas que ainda não receberam confirmação. Se este valor for ultrapassado, o envio de novas mensagens é bloqueado por alguns segundos, esperando estabilizar as confirmações.

Por fim, visto que estamos a tratar de um contexto de difícil intervenção técnica e alteração de *software* nos *end points* do sistema, decidiu-se implementar o tratamento de lógica cíclica dos números de sequência.

3. Protocolo TelemetryStream

O protocolo TelemetryStream foi de fácil idealização, visto a comunicação ser feita por via do protocolo TCP, pelo que apenas se necessitou de definir os dados a enviar e a forma de serialização.

Um pacote de telemetria foi definido como se ilustra no Código 6. É de notar que, devido à possibilidade de compactação da informação do estado operacional do *rover* (informação binária) e do estado das rodas, na serialização deste pacote usou-se a mesma técnica já referida no Código 4.

```
type TelemetryPacket struct {
  RoverID      uint8  // ID do rover
  Timestamp    int64 // Timestamp do Unix
  Position     utils.Coordinate // (Latitude, Longitude)
  State        uint8 // Estado operacional (2 bits)
  Battery      uint8 // Percentagem de bateria (0-100%)
  Speed        float32 // Velocidade em m/s
  Temperature  int16 // Temperatura interna (°C * 10)
  WheelStatus  uint8 // Estado das rodas roda1|roda2|roda3|roda4 4 bits
  QueueP1Count uint8 // Número de missões na fila de prioridade 1
  QueueP2Count uint8 // Número de missões na fila de prioridade 2
  QueueP3Count uint8 // Número de missões na fila de prioridade 3
}
```

Código 6: Pacote de Telemetria.

4. Definição das tarefas

Nesta seção definir-se-ão as tarefas, fator relevante e específico do nossos sistema. Assegura-se, assim, na leitura dos próximos capítulos, o correto entendimento do sistema como um todo.

As tarefas têm-lhes associados uma duração e uma frequência de envio de *reports*. Para permitir maior flexibilidade de tipos de tarefas, assumimos que algumas delas poderiam ter estes campos a 0, indicando que são tarefas em que definir durações ou espaçamentos de *updates* constantes não fariam sentido. Têm ainda associadas um ponto de coordenadas, onde se deve executar dada tarefa.

Definiram-se 6 tipos distintos de tarefas a executar pelos *rovers*, visando alcançar uma quantidade diversificada de desafios que metessem à prova o protocolo.

- Análise ambiental: Uma análise ambiental deve enviar espaçadamente no tempo, com uma duração total, dados variáveis do ambiente. Estes dados são todos do tipo *float32*. Esta tarefa é o padrão pedido no enunciado, tendo que enviar vários pacotes de tamanho constante de forma intervalada;
- Mapeamento topográfico: Um mapeamento topográfico é similar à análise ambiental, tendo que enviar repetidamente no tempo, numa duração total, informação da sua altura e de coordenadas em volta do ponto em que se encontra;
- Coleta de amostras: Uma coleta de amostras deve enviar espaçadamente no tempo, com uma duração total, uma lista de componentes químicos. Esta proporciona o desafio de tratar uma lista que pode ter tamanho variável;
- Instalação de instrumentos científicos: Esta tarefa não tem uma frequência de atualização. Deve, na duração determinada, instalar um aparelho científico num local e devolver apenas um pacote de confirmação ou não do sucesso da instalação. Esta desafia-nos a lidar com tipos de missões que operam de forma a ter resoluções determinísticas;
- Captura de imagem: A captura de imagem não tem uma duração nem uma frequência de *updates* determinísticas. Esta tarefa deve carregar para memória (simulação de tirar uma fotografia) uma imagem e devolver vários *chunks* da imagem, sendo possível reconstruí-la posteriormente na nave-mãe;
- Resgate de outros rovers: Inicialmente, esta missão deveria ser enviada a um *rover* sempre que algum outro *rover* fosse dado como inoperacional. Este dever-se-ia deslocar até às coordenadas do *rover* inoperacional e resgatá-lo, voltado o outro a operar. Ora, por impossibilidade de ligação direta entre os *rovers*, a simulação desta tarefa mostrou-se mais complexa do que planejado, pelo que, neste momento, funciona de forma mais simples. O *rover* desloca-se até às coordenadas dadas e numa duração determinada devolve um pacote com uma confirmação da reparação, ou uma negação dessa possibilidade.

Foram ainda implementadas interfaces de dispositivos e implementações de *mockups* que permitem retornar valores para os *reports* usando uma forma de modularidade e abstração mais próxima da realidade. Um destes exemplos é a

Camera, como representado no Código 7. *MockCamera* implementa a interface (em Go esta implementação é implícita) e são-lhe definidos os métodos da interface, que simulam dados reais.

```
// Interface de Camera
type Camera interface {
  ReadImageChunk() []byte
  LoadImage(path string) error
  GetTotalChunks() int
  GetChunk(index int) []byte
}

// MockCamera simula uma câmera para propósitos de teste
type MockCamera struct {
  imageData   []byte // Dados da imagem
  chunkSize   int    // Tamanho de cada chunk
  totalChunks int    // Número de chunks
}
```

Código 7: Interface Camera.

Ainda relativamente aos *devices*, é interessante salientar que as coordenadas são obtidas através de um *mockup* da interface GPS e a bateria do *rover* gerenciada através de um *mockup* da bateria.

5. Funcionamento da Nave-mãe

Comece-se por explorar a implementação e arquitetura da nave-mãe.

5.1. Configurações iniciais

A *nave-mãe* inicia com a leitura de um arquivo de configurações globais do sistema, obtendo portas de conexão e valores de constantes que utilizará ao longo do programa. Este arquivo, `config.json`, foi criado para permitir uma maior flexibilidade na configuração do sistema, facilitando a adaptação a diferentes ambientes operacionais ou alterações nos parâmetros de comunicação, como endereços IP, números de porta, e valores de temporização (*timeouts*) para as retransmissões do protocolo.

É criada e inicializada uma estrutura *MotherShip*, responsável por guardar todos os dados e estruturas relevantes para o funcionamento regular da nave-mãe.

É inicializado também um *Logger*, componente definido para cumprir a função crucial de rastreamento e diagnóstico da aplicação. Este permite um sistema centralizado e multifacetado. Oferece saídas múltiplas simultâneas para múltiplos destinos, sendo estes o Terminal de execução do processo da *nave-mãe*, um ficheiro de gravação persistente de dados (`mothertship.log`) e para o *Ground Control*, interface gráfica do sistema. Existem 4 níveis de *logging*, informação (*INFO*), avisos (*WARN*), erros (*ERROR*) e *debug* (*DEBUG*), que permitem uma diferenciação clara entre diferentes níveis de criticidade e detalhe necessários para a monitorização do sistema.

Após isso, são carregadas do ficheiro `missions.json` as missões que a nave-mãe enviará aos *rovers*. As missões têm o formato apresentado na Código 8.

```
[ {
    "TaskType": 1,
    "Duration": 20,
    "UpdateFrequency": 5,
    "Priority": 3,
    "State": "Pending",
    "Coordinate": {"Latitude": 0.523456, "Longitude": -0.821234}
  } ]
```

Código 8: Exemplo de uma missão em formato JSON.

Após esta configuração inicial, as missões são carregadas para uma fila de missões (*MissionQueue*), implementada como um canal *bufferizado*. Esta estrutura de dados, nativa do Go, garante *thread-safety* automática e permite uma implementação elegante do padrão produtor-consumidor, onde as missões carregadas do ficheiro JSON são enviadas para a fila e posteriormente consumidas à medida que os *rovers* as requisitam.

A estrutura *MotherShip* mantém um registo de todos os *rovers* conectados através de um mapa *Rovers*, onde a chave é o identificador único do *rover* (*RoverId*) e o valor

é uma estrutura *RoverState*. Esta última contém toda a informação necessária para gerir a comunicação com cada rover individualmente. Estas informações são:

- o endereço UDP do *rover* para envio de pacotes;
- o número de sequência para envio de pacotes ao rover (protocolo ML);
- o Número de sequência esperado na receção do próximo pacote enviado pelo *rover*;
- um *buffer* para armazenar pacotes recebidos fora de ordem;
- a estrutura *Window*, responsável pelo controlo de fluxo e gestão de retransmissões;
- o contador de missões atualmente atribuídas ao *rover*.

5.2. Gestão de missões

O sistema utiliza um *MissionManager* para gerir o ciclo de vida completo das missões. Este componente mantém um mapa *thread-safe* de todas as missões ativas (*ActiveMissions*), permitindo adicionar missões, atualizar o seu estado, consultá-las em tempo real e removê-las.

Quando um *rover* se comunica pela primeira vez com a nave-mãe, deve enviar um pedido de *id* para uma porta padronizada em TCP que escuta de todas as origens e trata de atribuir um *id* ao respetivo *rover*. A nave-mãe cria uma nova entrada de *RoverState* no mapa, inicializa as estruturas de controlo e regista o *rover* no sistema de telemetria. Após isso, o *rover* pode enviar pedidos de missões à nave-mãe normalmente por UDP, visto já estar registado no servidor.

Para atribuir missões a fila de missões vai sendo consumida, a missão é adicionada ao *MissionManager* com um estado *Pending* e é atualizado o número de missões que atribuímos a esse *rover*. Após isso faz-se a serialização dos dados da missão e envia-se via *MissionLink*. Quem está encarregue desta função é o *PacketManager*.

5.3. Packet Manager

Decidiu-se, por motivos de adquirir maior eficácia protocolar, sendo-nos permitido devido às *threads* da linguagem Go serem leves e eficientes, criar uma *thread* para tratar cada pacote.

Após criar um pacote e preencher corretamente o seu cabeçalho, uma rotina de Go inicia a função *Packet Manager*, função esta abstrata que tanto se aplica à nave-mãe como ao *rover*.

O *Packet Manager* segue um algoritmo de 3 fases:

1. Registo na *Window*: Antes de enviar um pacote, este é sempre registado na *Window* com uma entrada contendo dois canais de sinalização (um relativo à confirmação do *Ack* e outro ao *fast Retransmit*). Este registo cria um ponto de sincronização que permite aguardar pela confirmação de receção por parte do *rover*, que a rotina *receiver* possa sinalizar os *Ack's* eficientemente e que o mecanismo de *fast retransmit* possa forçar a retransmissão imediatamente;

2. Ciclo de envio e retransmissão: O pacote entra num ciclo controlado que permite um número máximo de tentativas de envio da mensagem. Isto permite um algoritmo de espera adaptativo, podendo haver a receção do *Ack* e retorno gracioso da função, um envio imediato do pacote caso tenhamos recebido 3 *Ack's* duplicados para o mesmo número de sequência ou um *timeout* onde se assume perda de pacote e se tenta reenviar o mesmo pacote. Após o máximo número de tentativas, o pacote é descartado e considerado como falhado no envio;
3. Atualização do RTO dinâmico: Para amostrar de RTT (*round trip time*) limpas, apenas são considerados no cálculo valores de pacotes sucedidos no envio. Aplica-se o algoritmo de Karn e atualiza-se o valor do RTO.

Os pacotes do tipo *ACK* são considerados especiais. Estes são enviados uma única vez, não aguardam retransmissão e são processados de forma não bloqueante.

5.4. Receção de pacotes

Paralelamente ao envio de pacotes, a nave-mãe mantém uma rotina dedicada (*receiver*) que escuta continuamente a porta UDP do protocolo ML. Esta rotina é responsável por receber todos os pacotes provenientes dos *rovers*, sejam eles pedidos de missões, relatórios de progresso ou confirmações (ACKs).

O processamento de pacotes recebidos segue um *pipeline* de validação e ordenação:

1. Validação de Integridade: A primeira verificação realizada é a validação do *checksum* modular de 8 *bits*. Os pacotes que não verifiquem esta integridade, são descartados;
2. Processamento ordenado: Para garantir que os dados são processados na ordem correta (essencial para reconstituição de imagens *multi-chunk*, por exemplo), implementa-se um sistema de *buffering* de pacotes fora de ordem. Um pacote é processado imediatamente apenas se tem o número de sequência esperado, caso contrário é guardado no *buffer*. Sempre que é recebido um pacote, é também feito a verificação de processamento consecutivo, visto poder-se processar em ordem mais pacotes previamente armazenados;
3. Envio de *Ack's*: No fim desta lógica, é enviado um pacote do tipo *ACK* com o próximo *byte* esperado (soma do número de sequência atual com o tamanho do *payload* do pacote).

Após este tratamento, o pacote é tratado por tipo.

5.5. Tratamento de reports

Os *rovers* enviam *reports* periodicamente sobre o progresso das missões. A nave-mãe tem a função de desserializar os dados enviados. Ao receber os dados o *MissionManager* atualiza o estado da missão. Para qualquer missão, se receber um pacote do *report* com o campo *isLastReport* do header interno de *report* a *true*, marca

a missão como concluída. Para o caso das imagens, tem que armazenar os *chunks* e reconstruir a imagem.

5.6. Protocolo TelemetryStream

A lógica envolvendo o protocolo *TelemetryStream* é bastante mais simples. Cada *rover* envia pacotes de telemetria a cada 2 segundos. Existe uma rotina responsável por receber e tratar estes pacotes.

É feita uma pequena deteção de falhas. Caso sejam perdidos 3 pacotes de telemetria seguidos, o *rover* é dado como inoperacional.

6. Funcionamento do Rover

Tratar-se-á agora de explicar como o *Rover* é atuante no sistema.

6.1. Configurações iniciais

O *rover* inicia de forma similar à nave-mãe. Lê configurações globais do sistema do arquivo `config.json`, obtendo portas de conexão e valores de constantes que utilizará ao longo do programa.

É criada uma estrutura *Rover* que contém o *RoverBase*, uma estrutura base com informações do estado do *rover*, os *devices* simulados deste, as conexões e o *Logger* de informações.

O *rover* é inicializado numa posição aleatória dentro dos limites definidos (por defeito assumimos um mapa quadrangular de valores de latitude e latitude variáveis entre -1 e 1).

6.2. Estabelecimento de conexões

Ao contrário da nave-mãe, o *rover* é o iniciador da comunicação em ambos os protocolos.

Inicia por enviar um pedido do seu *id* para uma porta padronizada TCP da nave-mãe. A nave-mãe guarda-o no sistema dela, passando a conseguir gerenciar os contactos com o *rover*. Envia de volta uma comunicação com o *id* do *rover* e tempo de frequência para as missões. Assim, o identificador dele é-lhe atribuído dinamicamente neste primeiro contacto.

Relativamente ao protocolo MissionLink, o *rover* troca mensagens pela porta padrão *ML (MissionLink)* da nave-mãe.

Já o protocolo *TS (TelemetryStream)* estabelece conexão TCP persistente com a nave-mãe e inicia uma rotina de envio periódico de telemetria.

6.3. Gestão, pedido e execução de missões

Uma das diferenças fundamentais do *rover* em relação à nave-mãe é a gestão de múltiplas missões com sistema de prioridades. O rover mantém três filas independentes de prioridades.

Quando o *rover* recebe uma missão via *ML*, esta é adicionada à fila correspondente baseada no campo *Priority* (1, 2 ou 3). O *rover* executa missões seguindo uma política de prioridade estrita.

Quando todas as filas estão vazias, o *rover* envia um *mission request* à nave-mãe, solicitando um número fixo de missões definido inicialmente no `config.json`.

Cada missão segue um fluxo de três fases:

1. Movimentação: O rover move-se em linha reta da posição atual até às coordenadas da missão, calculando a distância, o tempo de viagem aproximado e atualizações de posição. A telemetria, durante a deslocação, reflete o estado

"Pending", com velocidade simulada alterável e uma posição atualizada continuamente;
2. Início de missão: Ao chegar ao local, o *rover* inicia a missão, decidindo o seu fluxo dependente do tipo de missão;
3. Envio de relatórios: À medida que vai executando a missão, o *rover* envia os *reports* relativos desta missão.

6.4. Receção de pacotes

O *rover* tem uma rotina *receiver* que escuta continuamente os pacotes da nave-mãe.

Caso receba uma *mission* deve desserializar a *MissionData* e adicioná-la à fila de prioridade correspondente, enviando o *ACK* desta. Já se receber um *no mission*, regista-o em *log* e aguarda um tempo determinado antes de tentar pedir novamente uma missão. Por fim, se receber um *ACK*, sinaliza o *PacketManager* e liberta o *slot* correspondente da *Window*.

O processamento de pacotes segue a mesma lógica da nave-mãe. Existe uma validação inicial do *checksum*, um processamento ordenado com *buffering* e um envio automático de *Ack's* para os pacotes que o necessitem.

6.5. TelemetryStream

Paralelamente à execução de missões, o *rover* mantém um rotina dedicada a enviar periódicamente os pacotes de telemetria.

Os pacotes deste tipo estabelecem o estado do *rover*, que pode ser *Idle*, ou seja, sem missões a executar, de *In Mission*, quando se está a deslocar para o local de uma missão ou já a executá-la e *Suspend*, quando está com bateria baixa e está a recarregar ou há alguma falha.

6.6. Gestão de bateria e suspensão

O *rover* inclui um monitor de bateria que suspende operações quando o nível é crítico. A função desta rotina apresenta-se no Código 9.

Durante a suspensão do *rover* as missões em execução são pausadas. Após a recarga as operações são normalmente retomadas. Esta operação é interessante para provar a correção de estado do *rover*, conseguindo retomar com normalidade às suas ações após uma paragem prolongada de troca de mensagens.

6.7. Packet Manager e Window

O rover utiliza exatamente a mesma implementação de *PacketManager* e *Window* da nave-mãe, garantindo simetria no protocolo. A única diferença é que a nave-mãe mantém uma *Window* por *rover* (múltiplas) e rover mantém uma única *Window* (comunicação apenas com nave-mãe).

```
func (rover *Rover) batteryMonitor() {
  ticker := time.NewTicker(config.BATTERY_MONITOR_INTERVAL)
  defer ticker.Stop()

  for range ticker.C {
    level := rover.Devices.Battery.GetLevel()

    // Aviso de bateria abaixo de 20%
    if level <= 20 && level > 5 {
      rover.Logger.Warnf("Battery", "Low battery level: %d%%", level)
    }

    // Estado crítico - abaixo de 5% - suspende imediatamente se não
foi ainda suspensa
   if level <= config.CRITICAL_BATTERY_LEVEL && !rover.IsSuspended() {
        rover.Logger.Errorf("Battery", "Critical battery level: %d%
%", level)
        rover.Logger.Warn("Battery", "Suspending all operations for
recharge", nil)
      go rover.SuspendForLowBattery()
    }
  }
}
```

Código 9: Monitor de bateria.

6.8. Concorrência no Rover

O rover executa múltiplas *goroutines* simultaneamente:

- Main loop: Execução sequencial de missões das filas;
- ML Receiver: Receção contínua de pacotes UDP;
- TS Sender: Envio periódico de telemetria (2 s);
- PacketManagers: Uma goroutine por pacote enviado (reports, pedidos);
- Battery Monitor: Verificação periódica de bateria (1 s).

Esta arquitetura concorrente permite que o rover execute missões enquanto recebe novas, envie telemetria sem bloquear operações, processe múltiplos *reports* em paralelo e responda rapidamente a missões de resgate.

Esta arquitetura modular e concorrente permite que cada *rover* opere de forma autónoma, gerindo múltiplas missões com prioridades, adaptando-se a falhas, e mantendo comunicação confiável com a nave-mãe através de protocolos robustos sobre UDP e TCP.

7. Ground Control

7.1. API de Observação

A comunicação entre a nave-mãe e aplicações externas (como o *dashboard* do controlo terrestre) é realizada através de uma API genérica e extensível que suporta tanto *endpoints* REST, como conexões *WebSocket* em tempo real.

A API foi desenhada com um padrão de *Data Provider*, onde cada *endpoint* é registado dinamicamente com uma função que fornece os dados. Esta função apresenta-se no Código 10.

```
type DataProvider func() interface{}

func (api *APIServer) RegisterEndpoint(path string, method string,
provider DataProvider)
```

Código 10: Função que fornece os dados da API.

Esta abordagem permite que qualquer módulo da aplicação exponha dados sem modificar o código do servidor API, bastando registar um novo *endpoint*. Mostra-se, no Código 11 um exemplo de registo de *endpoints*.

```
api.RegisterEndpoint('/api/rover', 'GET', ms.handleListRovers)
api.RegisterEndpoint('/api/missions', 'GET', ms.handleListMissions)
```

Código 11: Endpoints do sistema.

A API disponibiliza *endpoints* REST para consulta de estado dos rovers e missões, um canal *WebSocket* (/ws/telemetry) para envio de atualizações em tempo real através do método `PublishUpdate()`, e suporte CORS para permitir o acesso de aplicações *web* externas.

7.2. Nó Ground Control

O *dashboard* conecta-se à nave-mãe através de chamadas HTTP aos *endpoints* REST, que instancia os dados recebidos em classes JavaScript (Código 12).

```
const loadData = async () => {
  const r = await fetch(`${API_BASE}/rovers`);
  rovers.value = (await r.json()).map(obj => new Rover(obj));

  const m = await fetch(`${API_BASE}/missions`);
  missions.value = (await m.json()).map(obj => new Mission(obj));
};
```

Código 12: Fetch feito pelo GC para obter os dados na Nave-Mãe.

Os dados são atualizados automaticamente a cada 2 segundos via *polling*, complementado por uma conexão *WebSocket* para notificações em tempo real.

Em termos de interface, optamos pela utilização da *framework* VUE, dado estarmos a utilizar a mesma para o trabalho prático da UC de Interface Pessoa-Máquina.

Disponibilizam-se abaixo algumas imagens que mostram todos os seus estados:

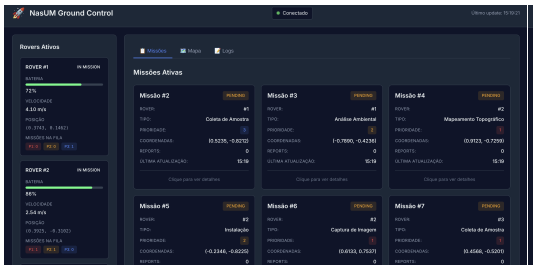


Figura 4: Ecrã Principal.

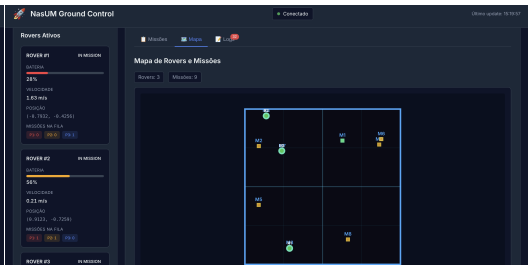


Figura 5: Mapa.

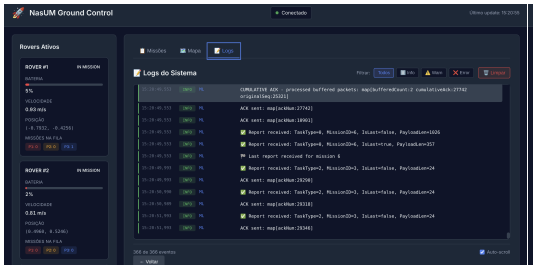


Figura 6: Logs.

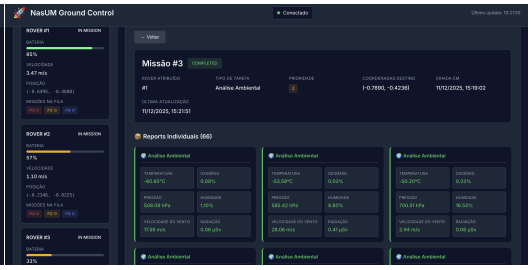


Figura 7: Missão.

8. Testes e Resultados

8.1. Testes em diferentes topologias

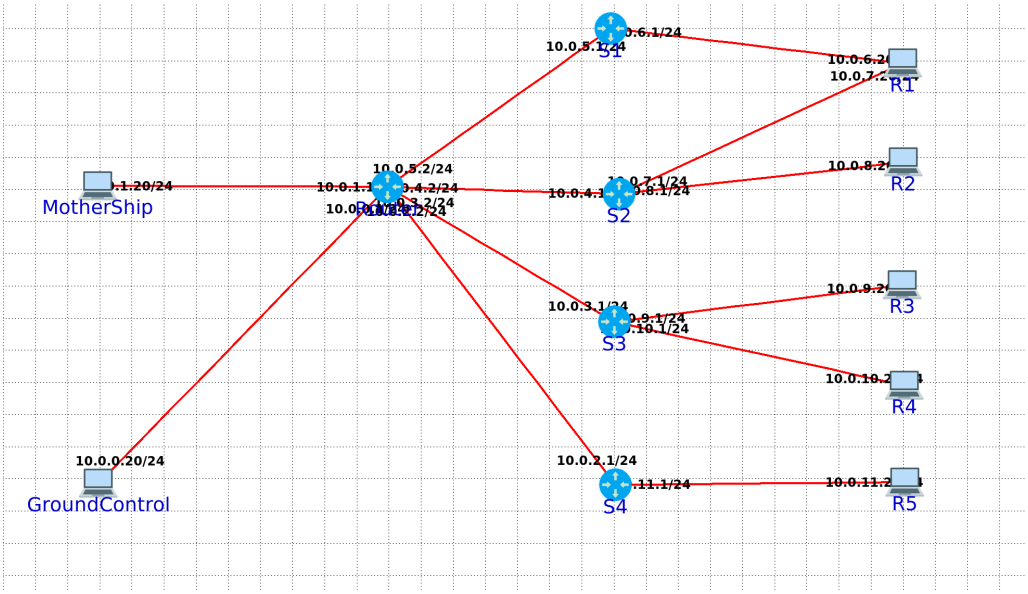
Para avaliar o desempenho do nosso protocolo e do programa, considerámos três topologias distintas, cada uma representando condições progressivamente mais adversas.

Para analisar o comportamento do programa em cada topologia, criámos uma forma de executar o programa em modo teste, onde são guardadas métricas acumuladas de acontecimentos do sistema. Estas métricas permitem verificar de forma detalhada o envio e receção de pacotes, confirmações (ACK's), retransmissões, *checksums* incorretos e outros eventos relevantes para a avaliação do protocolo.

Para cada topologia, deixámos o programa executar durante um período significativo e, posteriormente, analisámos as métricas resultantes para identificar quaisquer inconsistências ou anomalias no funcionamento do sistema.

8.1.1. Ambiente A - Ideal

Nesta topologia, simulamos um ambiente ideal para o funcionamento do programa, sem perdas de conexão e com um delay constante de 150 ms aplicado a todas as ligações (embora pouco realista).



Após deixarmos o sistema em execução durante um período prolongado, recolhemos as seguintes métricas:

Resultados Obtidos

```
{
  "uptime": "1m43s",
  "packets_sent": 15,
  "packets_received": 173,
  "acks_sent": 153,
  "acks_received": 15,
  "checksums_failed": 0,
  "retransmissions": 0,
  "packets_lost": 0,
  "duplicates_received": 0,
  "out_of_order_received": 41,
  "bytes_sent": 510,
  "bytes_received": 96039,
  "avg_rtt": "600.373ms",
  "min_rtt": "600.216ms",
  "max_rtt": "600.713ms",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent": 0,
  "duplicate_rate_percent": 0,
  "throughput_sent_bps":
4.949869810173724,
  "throughput_recv_bps":
932.118719018185,
  "packet_types_sent": {
    "MSG_MISSION": 15
  },
  "packet_types_received": {
    "MSG_ACK": 15,
    "MSG_REPORT": 153,
    "MSG_REQUEST": 5
  }
}
```


Código 13: Estatísticas da comunicação no Ambiente A - Nave-Mãe.


```
{
  "uptime": "1m29s",
  "packets_sent": 131,
  "packets_received": 132,
  "acks_sent": 3,
  "acks_received": 130,
  "checksums_failed": 0,
  "retransmissions": 0,
  "packets_lost": 0,
  "duplicates_received": 0,
  "out_of_order_received": 0,
  "bytes_sent": 95071,
  "bytes_received": 1005,
  "avg_rtt": "600.819ms",
  "min_rtt": "600.196ms",
  "max_rtt": "602.308ms",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent":
0,
  "duplicate_rate_percent": 0,
  "throughput_sent_bps":
1069.4739850482217,
  "throughput_recv_bps":
11.305459656188141,
  "packet_types_sent": {
    "MSG_REPORT": 130,
    "MSG_REQUEST": 1
  },
  "packet_types_received": {
    "MSG_ACK": 129,
    "MSG_MISSION": 3
  }
}
```


Código 14: Estatísticas da comunicação no Ambiente A - Rover1.

- **Análise dos resultados**

Tanto na Nave-Mãe como no Rover1, verificamos métricas perfeitas: zero retransmissões, zero pacotes perdidos, zero duplicados e zero checksums falhados. O RTT médio mantém-se consistente em aproximadamente 600 ms em ambos os nós, com variação mínima (inferior a 1 ms no Rover1 e cerca de 0.5 ms na Nave-Mãe), refletindo a estabilidade do *delay* constante de 150 ms configurado e a ausência de congestionamento ou perturbações na rede.

Um aspeto interessante a notar é que a Nave-Mãe regista 41 pacotes recebidos fora de ordem, representando cerca de 23.7% dos pacotes recebidos, enquanto o Rover1 não apresenta nenhum pacote fora de ordem. Esta assimetria deve-se à natureza assíncrona da implementação: utilizamos *goroutines* separadas para processar cada pacote, o que pode resultar em reordenação quando múltiplos pacotes são enviados rapidamente. Este fenómeno é particularmente evidente durante o envio de capturas de imagem pelos rovers, onde múltiplos pacotes são

transmitidos em sequência rápida até atingir o limite de *packets in flight*, aumentando a probabilidade de ultrapassagem entre *goroutines*. O facto de o Rover1 não apresentar pacotes fora de ordem reflete o baixo volume e frequência de pacotes recebidos (apenas missões da nave-mãe). Apesar desta reordenação, o protocolo lida eficazmente com a situação sem necessitar de retransmissões.

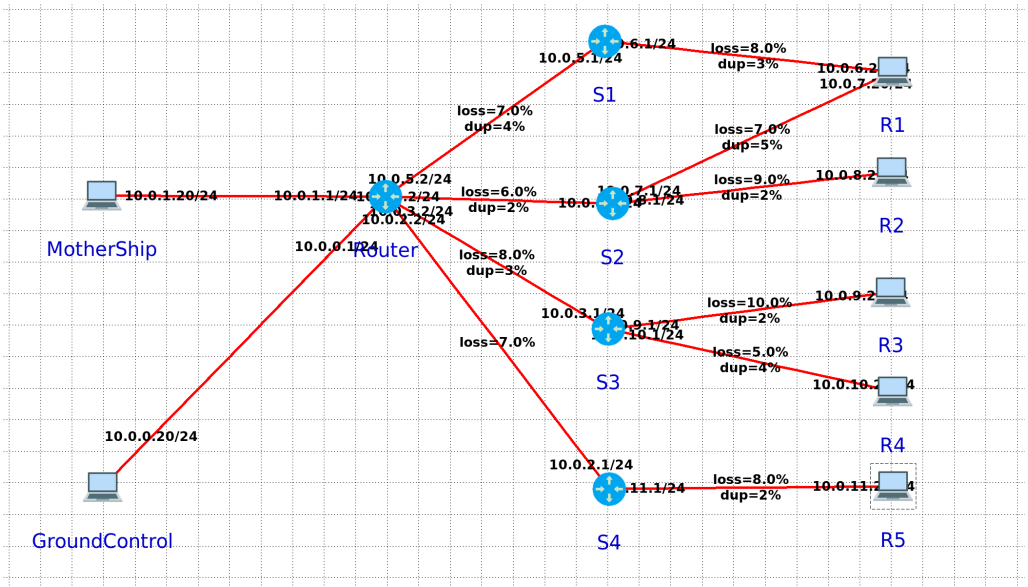
A simetria observada entre pacotes enviados e recebidos (131 enviados e 132 recebidos no Rover1, por exemplo) confirma a comunicação bidirecional sem perdas. Os tipos de mensagens trocadas evidenciam o funcionamento normal do protocolo: a nave-mãe envia missões e recebe *reports*, enquanto os *rovers* enviam *reports* periódicos e recebem as confirmações correspondentes.

O *throughput*, embora modesto devido à natureza do teste, mantém-se estável e proporcional ao volume de dados transmitidos. A Nave-Mãe apresenta maior *throughput* de receção (≈ 932 bps) devido aos múltiplos reports recebidos dos rovers, enquanto o Rover1 apresenta maior throughput de envio (≈ 1069 bps) ao transmitir os seus relatórios detalhados.

Concluímos assim que o nosso protocolo opera de forma eficiente e fiável em condições ideais, estabelecendo uma *baseline* de desempenho para comparação com topologias mais adversas.

8.1.2. Ambiente B - Realista

Dado estarmos a simular uma comunicação interespacial, é evidente que é inconcebível ter uma conexão perfeita, sem perdas, pelo que esta topologia reflete isso. Aplicámos entre 5% a 10% de perda de pacotes a todas as conexões, tal como um delay de 200 ms e uma probabilidade de duplicação de 0% a 5% em algumas conexões.



Resultados Obtidos

```
{
  "uptime": "1m27s",
  "packets_sent": 12,
  "packets_received": 200,
  "acks_sent": 186,
  "acks_received": 9,
  "checksums_failed": 0,
  "retransmissions": 3,
  "packets_lost": 0,
  "duplicates_received": 47,
  "out_of_order_received": 97,
  "bytes_sent": 408,
  "bytes_received": 159515,
  "avg_rtt": "1.000974s",
  "min_rtt": "1.000551s",
  "max_rtt": "1.001276s",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent":
25,
  "duplicate_rate_percent":
19.02834008097166,
  "throughput_sent_bps":
4.703061737826609,
  "throughput_recv_bps":
1838.747287032871,
  "packet_types_sent": {
    "MSG_MISSION": 12
  },
  "packet_types_received": {
    "MSG_ACK": 11,
    "MSG_REPORT": 185,
    "MSG_REQUEST": 4
  }
}
```


Código 15: Estatísticas da comunicação no Ambiente B - Nave-Mãe.


```
{
  "uptime": "1m19s",
  "packets_sent": 199,
  "packets_received": 154,
  "acks_sent": 5,
  "acks_received": 105,
  "checksums_failed": 0,
  "retransmissions": 90,
  "packets_lost": 0,
  "duplicates_received": 2,
  "out_of_order_received": 2,
  "bytes_sent": 175913,
  "bytes_received": 1213,
  "avg_rtt": "966.725ms",
  "min_rtt": "372.495ms",
  "max_rtt": "1.00238s",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent":
45.22613065326633,
  "duplicate_rate_percent":
1.282051282051282,
  "throughput_sent_bps":
2236.515882404933,
  "throughput_recv_bps":
15.421792393724079,
  "packet_types_sent": {
    "MSG_REPORT": 197,
    "MSG_REQUEST": 2
  },
  "packet_types_received": {
    "MSG_ACK": 149,
    "MSG_MISSION": 5
  }
}
```


Código 16: Estatísticas da comunicação no Ambiente B - Rover1.

- **Análise dos resultados**

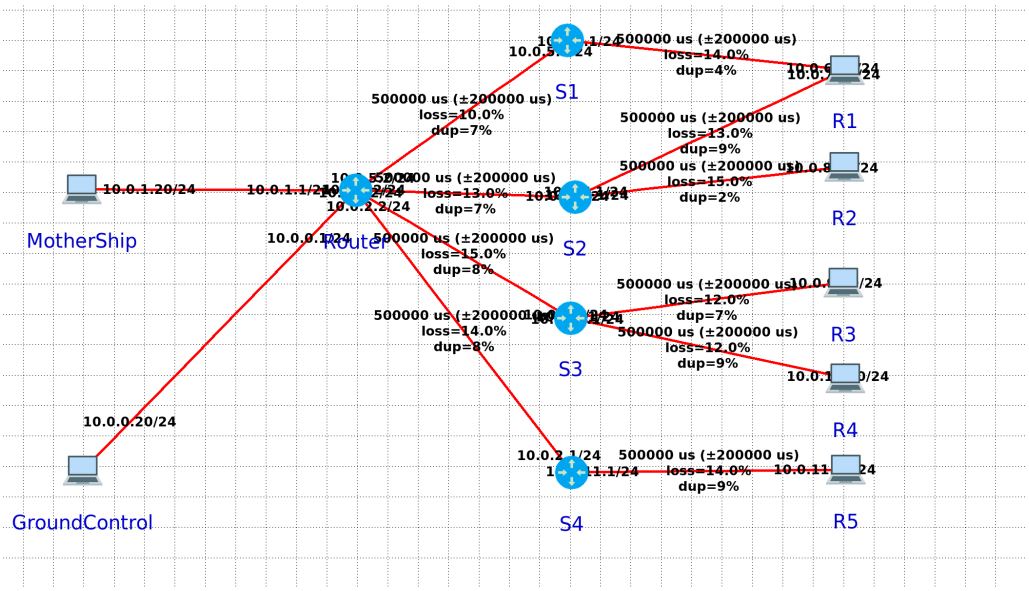
Na Nave-Mãe, verificamos uma taxa de retransmissão de 25% e uma taxa de duplicados de aproximadamente 19%, reflexo direto da perda e duplicação de pacotes configuradas na topologia. Um aspeto particularmente significativo é a elevada taxa de pacotes fora de ordem, com 97 pacotes recebidos fora de sequência, representando cerca de 48.5% dos pacotes recebidos. O RTT médio de aproximadamente 1 segundo é significativamente superior ao ambiente ideal (aumento de cerca de 67%), resultado do maior *delay* configurado (200 ms vs 150 ms) e das retransmissões necessárias. A variação do RTT mantém-se mínima (entre 1.000551s e 1.001276s), demonstrando estabilidade apesar das condições adversas. Não foram registadas falhas de *checksum*, confirmando a integridade dos dados transmitidos.

No Rover1, a situação é mais crítica, com uma taxa de retransmissão substancial de 45.2%, quase o dobro da nave-mãe. O RTT médio situa-se em aproximadamente 967 ms, consistente com o *delay* configurado, mas apresenta uma variação muito mais ampla, com valores entre 372 ms e 1.002 s. Esta amplitude de variação reflete o impacto combinado das perdas de pacotes e dos *timeouts* de retransmissão. Curiosamente, este nó regista apenas 2 pacotes duplicados (1.28%) e 2 pacotes fora de ordem.

De forma geral, o protocolo demonstra resiliência face às adversidades, conseguindo manter a comunicação operacional sem perda efetiva de pacotes, graças aos mecanismos de retransmissão e confirmação implementados. No entanto, o custo desta robustez manifesta-se nas elevadas taxas de retransmissão (especialmente no Rover1 com 45%) e no aumento do RTT comparativamente ao ambiente ideal.

8.1.3. Ambiente C - Péssimo

Para testar verdadeiramente as capacidades do nosso protocolo, falta avaliar o seu funcionamento numa topologia extremamente instável. Colocámos um delay de 500 ms em todos os *links*, com um *jitter* de 200 ms, perdas entre 10% e 15%, e probabilidades de duplicação mais elevadas.



Resultados Obtidos

```
{
  "uptime": "1m37s",
  "packets_sent": 18,
  "packets_received": 374,
  "acks_sent": 355,
  "acks_received": 15,
  "checksums_failed": 0,
  "retransmissions": 3,
  "packets_lost": 0,
  "duplicates_received": 77,
  "out_of_order_received": 204,
  "bytes_sent": 612,
  "bytes_received": 325501,
  "avg_rtt": "961.018ms",
  "min_rtt": "800.654ms",
  "max_rtt": "1.001891s",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent":
16.666666666666664,
  "duplicate_rate_percent":
17.073170731707318,
  "throughput_sent_bps":
6.340226218482263,
  "throughput_recv_bps":
3372.140480951299,
  "packet_types_sent": {
    "MSG_MISSION": 18
  },
  "packet_types_received": {
    "MSG_ACK": 14,
    "MSG_REPORT": 355,
    "MSG_REQUEST": 5
  }
}
```


Código 17: Estatísticas da comunicação no Ambiente C - Nave-Mãe.


```
{
  "uptime": "1m22s",
  "packets_sent": 209,
  "packets_received": 178,
  "acks_sent": 3,
  "acks_received": 107,
  "checksums_failed": 0,
  "retransmissions": 101,
  "packets_lost": 0,
  "duplicates_received": 0,
  "out_of_order_received": 0,
  "bytes_sent": 192322,
  "bytes_received": 1327,
  "avg_rtt": "928.064ms",
  "min_rtt": "23.292ms",
  "max_rtt": "1.35784s",
  "packet_loss_rate_percent": 0,
  "retransmission_rate_percent":
48.32535885167464,
  "duplicate_rate_percent": 0,
  "throughput_sent_bps":
2342.7024618821733,
  "throughput_recv_bps":
16.164381437992763,
  "packet_types_sent": {
    "MSG_REPORT": 208,
    "MSG_REQUEST": 1
  },
  "packet_types_received": {
    "MSG_ACK": 175,
    "MSG_MISSION": 3
  }
}
```


Código 18: Estatísticas da comunicação no Ambiente C - Rover1.

- **Análise dos resultados**

Na Nave-Mãe, observamos uma taxa de retransmissão relativamente moderada de aproximadamente 16.7%, o que sugere que o protocolo está a lidar de forma eficaz com as perdas configuradas. A taxa de duplicados atinge os 17%, evidenciando o impacto da duplicação de pacotes na rede. O aspeto mais notável é a elevada taxa de pacotes fora de ordem, com 204 pacotes recebidos fora de sequência, representando cerca de 54.5% dos pacotes recebidos - este valor é consistente com o jitter de 200 ms configurado, que causa variações substanciais nos tempos de chegada. O RTT médio situa-se em aproximadamente 961 ms, substancialmente inferior ao *delay* base de 500 ms somado com o *jitter*, o que indica que o RTT reflete principalmente o caminho de ida e volta mais rápido. A variação do RTT entre 800 ms e 1.002 s demonstra alguma instabilidade, embora menos dramática do que seria expectável dado o *jitter* configurado.

No Rover1, a situação é significativamente mais crítica, com uma taxa de retransmissão alarmante de 48.3%, quase metade de todos os pacotes enviados necessitando de retransmissão. O RTT médio de aproximadamente 928 ms apresenta uma variação muito mais ampla, com valores entre 23 ms e 1.358 s. O valor mínimo de 23 ms é claramente uma anomalia (possivelmente relacionado com medições de ACKs locais ou erros de cálculo), enquanto o máximo de 1.358 s reflete o impacto combinado do *delay* base, *jitter* e múltiplas tentativas de retransmissão. Curiosamente, este nó não regista nenhum pacote duplicado nem pacotes fora de ordem, contrastando fortemente com os valores observados na nave-mãe. Esta assimetria é expectável dado que os únicos pacotes recebidos pelo *rover* são missões da nave-mãe, um volume muito menor que o fluxo contrário de *reports*.

Apesar das condições catastróficas, o aspeto mais notável é que não se registaram perdas efetivas de pacotes em nenhum dos nós (estamos a utilizar um limite de retransmissões de 5). Isto demonstra a robustez fundamental do protocolo: mesmo com 48% de retransmissões necessárias no *rover*, o sistema consegue, eventualmente, entregar todos os pacotes através dos mecanismos de *timeout* e retransmissão implementados. Os *checksums* mantêm-se todos válidos, confirmando a integridade dos dados mesmo nas piores condições.

No entanto, o custo desta resiliência é evidente na degradação do *throughput* e no aumento dramático dos tempos de resposta. O protocolo mantém a fiabilidade à custa da eficiência, o que é apropriado para um sistema onde a integridade dos dados é prioritária sobre a velocidade de transmissão.

9. Conclusão: Observações Finais

Para finalizar, deixamos algumas conclusões que achamos pertinentes.

A estrutura abstrata e modular do código desenvolvido permitiu, com relativa facilidade, fazer alterações protocolares e de estrutura/estado interno dos *endpoints* do sistema. Podemos dar como exemplo a implementação do RTO dinâmico, que foi bastante prática de executar, tal como uma alteração tardia de tratamento dos números de sequências de incrementais por um, para incrementais pelo tamanho do *payload* do pacote. Foi desenvolvida uma arquitetura modular e escalável, com clara separação de responsabilidades e código reutilizável.

A robustez do protocolo desenvolvido sobre UDP, com técnicas usadas no TCP, como o *fast retransmit*, o RTO dinâmico e o tratamento de números de sequência com aritmética modular, é também um ponto forte do nosso projeto.

Ainda a arquitetura de concorrência eficiente, evitando bloquear ações críticas e um paralelismo real elevado através de utilizar uma rotina por pacote é um ponto de orgulho da nossa parte.

Apesar dos pontos fortes no que toca à parte essencial do trabalho, temos alguns fatores de melhoria para trabalho futuro. A principal deve basear-se numa documentação mais profissional e gestão coerente de erros. Há vários momentos em que os comentários são pouco detalhados e dever-se-ia documentar extensivamente a comunicação protocolar e a gestão de erros muitas vezes retorna apenas *nil*, não sendo explícito no sei tratamento e informação. Expandir testes unitários e de integração seria também benéfico para a comprovação constante da correção do programa. Entretanto, consideramos que todos estes fatores que visam melhoria são pontos menos cruciais para a demonstração de conhecimentos relativos à unidade curricular e, por limitação de tempo, podem considerar-se secundários e para realizar futuramente.

O sistema cumpre os objetivos propostos, tendo, os estudantes envolvidos, desfrutado do trabalho realizado. As competências adquiridas deste trabalho são uma mais valia em contexto real de engenharia e pensamento crítico, tendo sido benéfico o esforço aplicado.