

# **Trabalho Prático**

CC3040 – Programação Concorrente (2024/2025)

Grupo 3

Ana Duarte Amorim – up202207213

Tomás dos Reis Leitao Torres Fontes – up202107382

# The Alternating Bit Protocol (ABP)

## Exercício 1.1 - Implementação do ABP em CCS

O objetivo deste exercício foi implementar o Alternating Bit Protocol (ABP) em CCS, abstraindo-nos do conteúdo das mensagens e focando-nos apenas no bit de controlo (0 ou 1).

Assim, fizemos esta implementação com os 4 processos principais: *Sender*, *Receiver*, *Trans* e *Ack*.

O **Sender** começa no estado **Sender0**, onde aceita uma nova mensagem (*'accept*) e envia-a com bit 0 (*'send0*). Depois pode ficar á espera da confirmação *rcv\_ack0* que faz com que vá para **Sender1**, ou volta a tentar enviar. O mesmo acontece em **Sender1**, mas com bit 1.

Já no **Receiver**, temos os estados **Receiver0** e **Receiver1**, que recebem mensagens com o bit correspondente (*receive0* ou *receive1*), fazem *'deliver* da mensagem e enviam o acknowledgment (*'snd\_ack0* ou *'snd\_ack1*), alternando também o estado.

O processo **Trans** representa o canal de comunicação do **Sender** para o **Receiver** e pode comportar-se de três formas:

- entregar a mensagem (*'receive0* ou *'receive1*),
- perdê-la ou
- duplicá-la (*'receive0* . *'receive0* ou *'receive1* . *'receive1* )

O processo **Ack** recebe uma acknowledgment do **Receiver** (*snd\_ack0/snd\_ack1*) e envia-o para o **Sender** (*'rcv\_ack0/'rcv\_ack1*).

Por fim, restringimos todas as ações internas usando o operador de restrição, deixando apenas visível *'accept* e *'deliver*.

Toda esta especificação foi implementada e testada na ferramenta **CAAL**, que permitiu simular e visualizar o comportamento do protocolo.

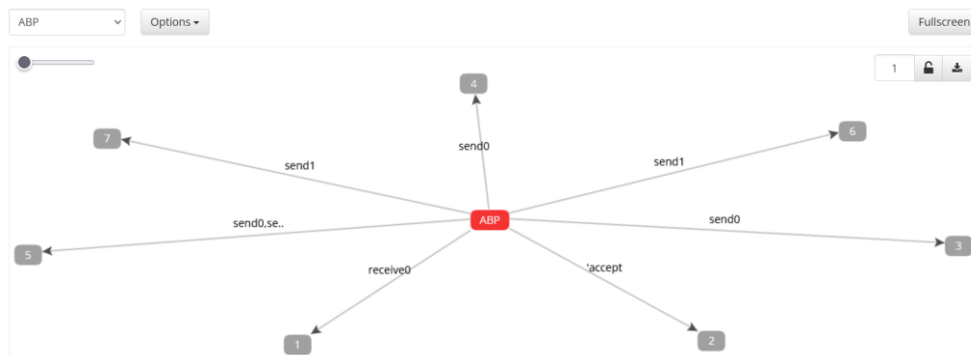


Fig. 1 Visualização parcial do grafo de estados do processo ABP no CAAL, com profundidade de expansão definida como 1.

## Exercício 1.2 – Implementação de SPEC e Comparação com o ABP

Neste exercício, o objetivo foi criar uma especificação alternativa do protocolo ABP em CCS, modelando-o como um único processo sequencial, sem composição paralela.

Assim, implementamos o processo **SPEC** apenas com as ações visíveis ‘accept’ e ‘deliver’, de forma sequencial e repetida:

$$\text{SPEC} = \overline{\text{'accept'}} . \overline{\text{'deliver'}} . \text{SPEC};$$

Fig. 2 Processo SPEC que implementamos

Fizemos desta forma pois o objetivo da especificação é representar apenas o comportamento externo observado pelo ambiente, ignorando totalmente os detalhes internos do protocolo, como reenvios, perdas de mensagem ou acknowledgments.

Depois, fizemos a comparação com a versão completa do protocolo ABP utilizando **bissimulação fraca** no CAAL. Optámos pela bissimulação fraca porque o nosso objetivo era comparar apenas o comportamento observável do sistema. A bissimulação forte, exige que os dois processos coincidam exatamente em todos os passos, incluindo os internos, o que não era o que pretendíamos.

Obtivemos o seguinte resultado:


| Status  | Time  | Property           |
|---|-------|--------------------|
|  | 29 ms | ABP $\approx$ SPEC |

Fig. 3 Resultado do teste de equivalência entre ABP e SPEC, usando bissimulação fraca no CAAL

Apesar de termos usado bissimulação fraca, que ignora os passos internos do sistema, podemos observar que o ABP e o SPEC continuam a **não ser** considerados **equivalentes**.

Isto acontece porque no ABP, pode haver situações onde são aceites várias mensagens seguidas sem que a entrega aconteça logo a seguir, por exemplo: se houver perdas ou reenvios. Já no SPEC cada ‘accept’ é imediatamente seguido de ‘deliver’.

## Exercício 1.3 – Reimplementação do ABP

Neste exercício, o objetivo foi reimplementar o protocolo *ABP* incluindo valores das mensagens e o bit de controlo enviados entre os processos.

Optamos por fazer esta implementação na ferramenta **PseuCo** e não no CAAL, porque o CAAL não permite enviar valores concretos como parâmetros nas ações.

Para além da adaptação do código para a sintaxe do PseuCo, as outras alterações que fizemos foram:

- Adicionámos a passagem da mensagem, com valores entre 0 e 9;
- O **Sender0** e **Sender1** agora recebem uma mensagem **m** via *accept?m* e enviam-na respetivamente por *send0!m* ou *send1!m*;
- Os **Receiver0** e **Receiver1** também recebem a mensagem **m** com *receive0?m* ou *receive1?m* e fazem *deliver!m* dessa mesma mensagem;
- O comportamento do **Trans** e **Ack** foi mantido de forma semelhante ao ABP anterior, mas agora com a transmissão dos valores;

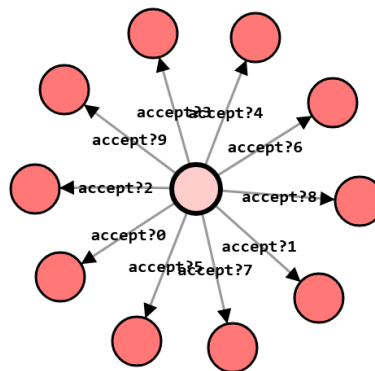


Fig 4. Transições iniciais do sistema ABP com passagem de valores

## Run processes in a server

### Exercício 2.1 - Implementação de mecanismos de sincronização no servidor

O objetivo deste exercício foi implementar um servidor que permite a execução de comandos do sistema operativo através de pedidos HTTP, dado que não é possível executar processos OS diretamente no browser via JavaScript. O

servidor recebe pedidos de um cliente web, executa os comandos solicitados e retorna os resultados.

O sistema é composto por:

- **Cliente web:** Uma interface HTML/JavaScript que permite ao utilizador inserir comandos e visualizar os resultados
- **Servidor:** Implementado em Scala utilizando a biblioteca http4s, responsável por processar os pedidos e executar os comandos

Os passos para a execução de cada versão do servidor são idênticos:

- Abrir o terminal no diretório “\*Mecanismo escolhido\*/server “
- Executar o comando ‘sbt run’
- Abrir o ficheiro client/index.html num browser
- Inserir comandos no campo "Input program" (por exemplo: pwd, ls, echo)
- Clicar em "Run the commands" para enviar os pedidos
- Verificar o estado do servidor clicando em "Status"

Para testar a concorrência e os limites de processos simultâneos:

- Executar múltiplos comandos de longa duração (ex: sleep 10)
- Observar que apenas 3 processos executam simultaneamente (limite definido)
- Verificar que comandos adicionais ficam em fila de espera

## Synchronized Blocks

Os **synchronized blocks** são usados neste contexto para proteger estruturas de dados partilhadas contra o acesso concorrente. O servidor gere várias estruturas de dados partilhadas:

- **counter:** Mantém o registo do número total de pedidos.
- **runningProcesses:** Uma lista dos comandos atualmente em execução, os seus IPs de utilizador e os números de processo.
- **pendingQueue:** Uma fila de comandos à espera de serem executados.

Sem a sincronização adequada, o acesso concorrente a estas estruturas de dados pode levar a condições de corrida, onde a ordem das operações de diferentes threads afeta o resultado. Isto pode resultar em:

- Contagens de pedidos incorretas.
- Corrupção de dados nas listas e filas.
- Estado do servidor inconsistente.

Também garantem que apenas uma thread pode aceder a estas estruturas de dados de cada vez, mantendo a integridade e a consistência dos dados. Isto é crucial para o correto funcionamento do servidor, especialmente ao lidar com vários pedidos de clientes ao mesmo tempo.

Todos os métodos que acedem ao estado usam ***this.synchronized***, como por exemplo:

- **incrementAndGetCounter**: Este método incrementa o counter e retorna o valor atualizado. O bloco `this.synchronized` garante que apenas uma thread pode executar esta operação de cada vez, prevenindo condições de corrida quando múltiplas threads tentam incrementar o contador.
- **canStartProcess**: Este método verifica se o número de processos em execução é inferior ao máximo permitido (`MAX_RUNNING`). O bloco `this.synchronized` protege a lista `runningProcesses` contra o acesso concorrente.

## Lock-Free programming

A programação lock-free é usada como uma abordagem alternativa para gerir o acesso concorrente a estruturas de dados partilhadas, com o objetivo de evitar a sobrecarga e os potenciais problemas associados aos locks. Em vez de usar locks para serializar o acesso, os algoritmos lock-free dependem de operações atómicas para garantir que múltiplas threads podem modificar dados partilhados sem interferir na correção umas das outras.

Nesta implementação, são utilizadas técnicas de programação lock-free para gerir o estado partilhado do servidor. As seguintes estruturas de dados são geridas utilizando operações atómicas:

- **counter**: Um ***AtomicInteger*** é usado para manter o número total de pedidos.
- **runningProcesses**: Um ***AtomicReference*** é usado para guardar um `ListBuffer` dos processos atualmente em execução.
- **pendingQueue**: Um ***AtomicReference*** é usado para guardar uma `Queue` de comandos à espera de serem executados.

As operações atómicas fornecem garantias sobre a indivisibilidade das operações, garantindo que, mesmo num ambiente concorrente, estas estruturas de dados são atualizadas de forma consistente.

A programação lock-free é implementada utilizando **AtomicInteger** e **AtomicReference** para gerir o estado partilhado de forma thread-safe sem depender de locks. Como por exemplo:

- **incrementAndGetCounter:** Este método utiliza **AtomicInteger.incrementAndGet()** para incrementar atomicamente o contador e obter o novo valor. Isto elimina a possibilidade de condições de corrida quando múltiplas threads tentam incrementar o contador.
- **removeRunningProcess:** Semelhante a **addRunningProcess**, este método utiliza **compareAndSet()** num loop para remover atomicamente um processo da lista **runningProcesses**. Encontra o índice do processo a remover, cria uma nova lista sem esse processo e, em seguida, atualiza atomicamente a referência **runningProcesses**.

## Volatile Variables

As variáveis **volatile** são utilizadas para garantir a visibilidade das alterações a variáveis partilhadas entre várias threads. Ao contrário da utilização de locks, as variáveis **volatile** fornecem uma forma mais fraca de sincronização, garantindo que cada leitura de uma variável **volatile** acontece antes de qualquer escrita subsequente, e vice-versa. Isto significa que as threads vão ver o valor mais atualizado de uma variável **volatile** sem necessariamente terem acesso exclusivo à mesma.

Nesta implementação, as variáveis **volatile** são usadas para gerir o estado partilhado do servidor. As seguintes estruturas de dados são declaradas como **volatile**:

- **counter:** Uma variável **Int** que mantém o registo do número total de pedidos.
- **runningProcesses:** Um **ListBuffer** que armazena os comandos atualmente em execução, os seus IPs de utilizador e os números de processo.
- **pendingQueue:** Uma **Queue** que contém os comandos à espera de serem executados.

## Exercício 2.2 - Implementação de Bad Synchronized Blocks

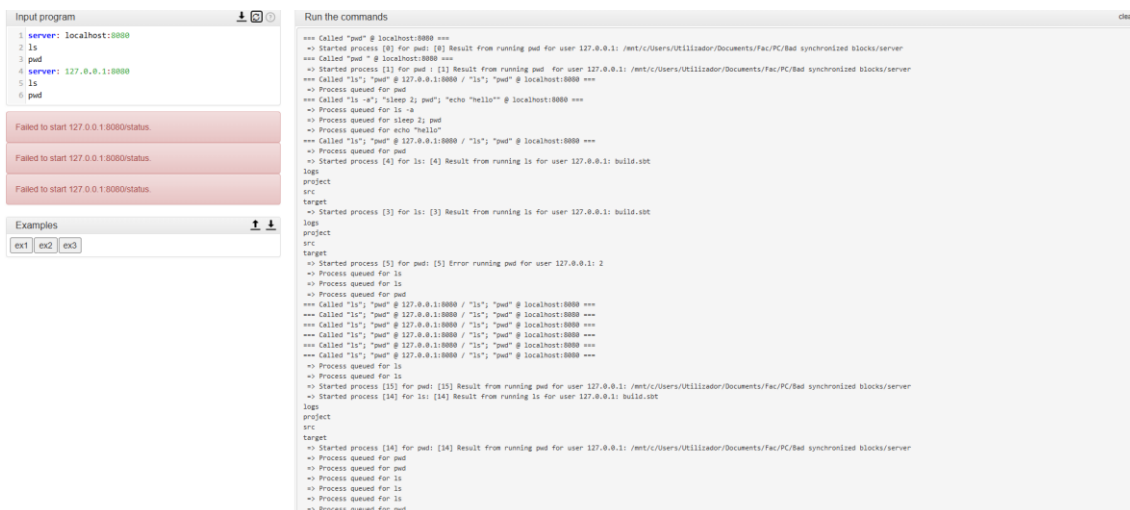
Esta “má” implementação do **ServerState** apresenta várias condições de corrida (data races) que podem levar a comportamentos inconsistentes e inesperados do servidor. Como por exemplo:

- O método **incrementAndGetCounter** apresenta uma condição de corrida clássica, entre a leitura do valor atual (**val current = counter**) e o incremento (**counter += 1**), outra

thread pode modificar o valor de counter, resultando em: contadores duplicados, perda de incrementos e estados inconsistentes do contador.

- As operações sobre **runningProcesses** e **pendingQueue** não são nem atômicas nem sincronizadas, logo várias threads podem, ao mesmo tempo, verificar se podem inciar um processo (**canStartProcess**), adicionar processos à listas (**addRunningProcess**) e fazer com que o limite de processos em execução seja excedido (**MAX\_RUNNING**)

O processo de replicação da ‘data race’ é relativamente simples, para tal basta enviar uma quantidade significativa de comandos para o servidor em simultâneo. O que faz com que o servidor acuse um erro e acabe por \*crashar\*.



The screenshot shows a terminal window with two main sections. The left section, titled 'Input program', contains a list of commands: 1 server: localhost:8080, 2 ls, 3 pwd, 4 server: 127.0.0.1:8080, 5 ls, 6 pwd. Below this, there are three red error messages: 'Failed to start 127.0.0.1:8080/status.', 'Failed to start 127.0.0.1:8080/status.', and 'Failed to start 127.0.0.1:8080/status.'. At the bottom of the left section, there is an 'Examples' section with buttons for 'ex1', 'ex2', and 'ex3'. The right section, titled 'Run the commands', displays a detailed log of system events, including process creation, execution, and termination. The log shows that the program is running on a server at localhost:8080 and is attempting to execute various commands. The log also shows that the program is running on a server at 127.0.0.1:8080 and is attempting to execute various commands. The log ends with a 'Process queued for pwd' message.



Se verificarmos os logs deparamo-nos com esta mensagem de erro:

```
java.lang.IndexOutOfBoundsException: 6
    at scala.collection.mutable.ListBuffer.remove(ListBuffer.scala:347)
    at cp.serverPr.ServerState.removeRunningProcess(ServerState.scala:28)
    at cp.serverPr.Routes$.anonfun$runProcess$1(Routes.scala:65)
    at cats.effect.IOFiber.runLoop(IOFiber.scala:413)
    at cats.effect.IOFiber.execR(IOFiber.scala:1362)
    at cats.effect.IOFiber.run(IOFiber.scala:112)
    at cats.effect.unsafe.WorkerThread.run(WorkerThread.scala:743)
2025-05-30 17:25:19 - ♦ Starting process (25) for user 127.0.0.1: ls
2025-05-30 17:25:19 - [25] Error running ls for user 127.0.0.1: 6
java.lang.IndexOutOfBoundsException: 6
    at scala.collection.mutable.ListBuffer.remove(ListBuffer.scala:347)
    at cp.serverPr.ServerState.removeRunningProcess(ServerState.scala:28)
    at cp.serverPr.Routes$.anonfun$runProcess$1(Routes.scala:65)
    at cats.effect.IOFiber.runLoop(IOFiber.scala:413)
    at cats.effect.IOFiber.execR(IOFiber.scala:1362)
    at cats.effect.IOFiber.run(IOFiber.scala:112)
    at cats.effect.unsafe.WorkerThread.run(WorkerThread.scala:634)
```

**Explicação:** O método *removeRunningProcess* tem uma race condition crítica:

1. Thread A encontra o índice 6 para remover um processo
2. Thread B remove outro elemento, encurtando a lista para 6 elementos (índices 0-5)
3. Thread A tenta remover o índice 6, mas este já não existe
4. **Resultado:** Crash da aplicação

Este crash demonstra perfeitamente como ‘data races’ podem causar falhas catastróficas no sistema, não apenas inconsistências nos dados.

## Simulating ABP with actors

### ***Exercício 3.1 – Descrever a estrutura e o comportamento dos atores***

Nesta parte do projeto, implementamos o protocolo ABP utilizando o modelo de atores da biblioteca *Akka*. O sistema é composto pelos seguintes Atores:

- **SenderActor:** Envia mensagens numeradas, alternado o bit de controlo (0 ou 1). Fica a aguardar um ack correspondente. Se este não for recebido dentro de 15 segundos, reenvia a mensagem até um máximo de 5 tentativas. Quando o ack correto, é recebido, avança para a próxima mensagem.
- **TransActor:** Simula o canal de transmissão entre o emissor e o recetor. Dependendo do modo escolhido pode:
  - Modo 1 – entregar sempre a mensagem
  - Modo 2 – perder ou duplicar mensagens, mas entregando-as sempre
  - Modo 3 – perder ou duplicar mensagens, não conseguindo entregá-las sempre

- **ReceiverActor:** Recebe mensagens do transmissor e valida o bit. Se for o esperado envia um ack para confirmar a receção e alterna o bit.
- **AckActor:** Recebe os acks do receiver e reencaminha-os para o SenderActor.

A hierarquia é:

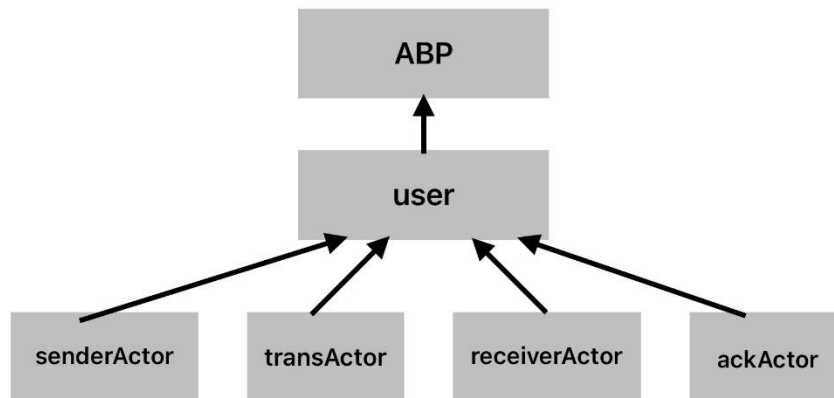


Fig. 5 Diagrama da hierarquia dos atores

### **Exercício 3.2**

O projeto encontra-se estruturado em vários ficheiros .scala na pasta akka-abp/src/main/scala.

Assim, o user deve abrir o terminal, ir para o diretório akka-abp e correr o comando sbt, que irá compilar o sistema.

Depois deve fazer run.

O sistema pedirá ao utilizador que diga um número de mensagens que quer enviar e de que modo as quer enviar. Se o user quiser um cenário sem perdas nem duplicações deverá escolher o Modo 1.

Por exemplo se escolher enviar 3 mensagens irá ver algo deste género:

```

Protocolo ABP iniciado
Quantas mensagens pretende enviar?
Numero: 3
Por favor, escolha um modo:
1 - Comunicacao com sucesso (sem perda, nem duplicacao)
2 - Comunicacao com algumas falhas (mensagens perdidas e duplicadas mas todas entregues)
3 - Comunicacao com falhas graves (mensagens nao entregues)
Modo: 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Receiver] Recebeu mensagem 1 com bit 0
[Ack] Recebido ack para bit 0
[Ack] A enviar ack para bit 0

[Sender] A enviar mensagem 2 com bit 1
[Trans] A processar a mensagem 2 com bit 1
[Receiver] Recebeu mensagem 2 com bit 1
[Ack] Recebido ack para bit 1
[Ack] A enviar ack para bit 1

[Sender] A enviar mensagem 3 com bit 0
[Trans] A processar a mensagem 3 com bit 0
[Receiver] Recebeu mensagem 3 com bit 0
[Ack] Recebido ack para bit 0
[Ack] A enviar ack para bit 0

Transmissao concluida. A encerrar o sistema
[sucess] Total time: 10 s, completed 30/05/2025, 16:35:27

```

Fig. 5 PrintScreen do output para 3 mensagens no modo 1

### Exercício 3.3

O user deve abrir o terminal, ir para o diretório akka-abp e correr o comando sbt, que irá compilar o sistema.

Depois deve fazer run.

O sistema pedirá ao utilizador que diga um número de mensagens que quer enviar e de que modo as quer enviar.

- Se o user quiser um cenário com perdas e duplicação de mensagens, mas onde as mensagens são sempre entregues deverá escolher o Modo 2.

Por exemplo se escolher enviar 3 mensagens irá ver algo deste género:

```
Quantas mensagens pretende enviar?
Numero: 3
Por favor, escolha um modo:
1 - Comunicacao com sucesso (sem perda, nem duplicacao)
2 - Comunicacao com algumas falhas (mensagens perdidas e duplicadas mas todas entregues)
3 - Comunicacao com falhas graves (mensagens nao entregues)
Modo: 2
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 duplicada, a enviar 2 vezes
[Receiver] Recebeu mensagem 1 com bit 0
[Receiver] Recebeu mensagem 1 com bit 0
[Ack] Recebido ack para bit 0
[Ack] A enviar ack para bit 0

[Sender] A enviar mensagem 2 com bit 1
[Trans] A processar a mensagem 2 com bit 1
[Trans] Mensagem 2 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 2
[Sender] A enviar mensagem 2 com bit 1
[Trans] A processar a mensagem 2 com bit 1
[Receiver] Recebeu mensagem 2 com bit 1
[Ack] Recebido ack para bit 1
[Ack] A enviar ack para bit 1

[Sender] A enviar mensagem 3 com bit 0
[Trans] A processar a mensagem 3 com bit 0
[Receiver] Recebeu mensagem 3 com bit 0
[Trans] Mensagem 3 duplicada, a enviar 2 vezes
[Receiver] Recebeu mensagem 3 com bit 0
[Ack] Recebido ack para bit 0
[Ack] A enviar ack para bit 0

Transmissao concluida. A encerrar o sistema
[sucesss] Total time: 34 s, completed 30/05/2025, 16:43:17
```

Fig. 6 PrintScreen do output para 3 mensagens no modo 2

- Se o user quiser um cenário com perdas e duplicação de mensagens, mas onde as mensagens nem sempre são entregues deverá escolher o Modo 3.

Por exemplo se escolher enviar 3 mensagens irá ver algo deste género:

```
Quantas mensagens pretende enviar?
Numero: 3
Por favor, escolha um modo:
1 - Comunicacao com sucesso (sem perda, nem duplicacao)
2 - Comunicacao com algumas falhas (mensagens perdidas e duplicadas mas todas entregues)
3 - Comunicacao com falhas graves (mensagens nao entregues)
Modo: 3
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
[Sender] A enviar mensagem 1 com bit 0
[Trans] A processar a mensagem 1 com bit 0
[Trans] Mensagem 1 perdida
Nenhuma resposta por 15 segundos. A reenviar mensagem 1
Limite de tentativas de reenvio atingido. Falha no envio. A fechar o sistema.
[success] Total time: 79 s (0:01:19.0), completed 30/05/2025, 16:47:47
```

Fig. 6 PrintScreen do output para 3 mensagens no modo 3