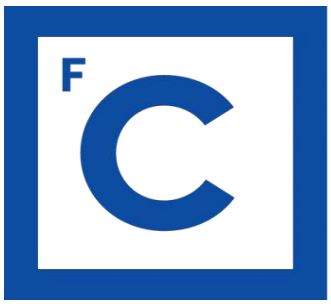

Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Mestrado em Engenharia Informática

RELATÓRIO DE PROJETO

Tolerância a Faltas Distribuída

Streamlet consensus algorithm (Phase 2)

Rodrigo Craveiro Rodrigues (Nº64370)

Denis Ungureanu (Nº56307)

Ana Luís (Nº53563)

Professor: **Doutor Alysson Bessani**

1º Semestre Letivo 2024/2025

dezembro 2024

Índice

1. Introdução

Este relatório descreve a implementação da segunda fase de um sistema distribuído tolerante a faltas, desenvolvido no âmbito da disciplina de **Tolerância a Faltas Distribuída**. O sistema desenvolvido recorre ao protocolo de consenso **Streamlet** para alcançar consenso num ambiente distribuído, simulando o funcionamento de uma blockchain. Na segunda fase do projeto, o foco principal é a implementação de mecanismos que permitam ao sistema lidar com **nós atrasados**, **perda de épocas**, capacidades de ***crash-recovery*** e a **gestão de *forks*** na blockchain, assegurando a convergência para uma única cadeia consistente de blocos.

2. Objetivos

O objetivo desta segunda fase é estender a implementação inicial do protocolo Streamlet para abordar desafios comuns em sistemas distribuídos reais. Pretende-se de forma académica confirmar e demonstrar alguns dos conceitos teóricos abordados em sala de aula. Assim, esta implementação visa proporcionar uma melhor compreensão dos desafios e das soluções associadas à coordenação e consenso dos sistemas distribuídos.

Primeira fase do projeto envolve:

1. **Implementar o protocolo Streamlet:** Desenvolver uma implementação completa do protocolo Streamlet, incluindo a proposta, votação, notarização e finalização de blocos.
2. **Simular um ambiente distribuído:** Criar um ambiente onde múltiplos nós operam de forma concorrente, comunicando-se através de *sockets* de rede, para simular uma rede distribuída real.
3. **Garantir a tolerância a faltas:** Assegurar que o sistema continua a operar corretamente na presença de falhas de paragem (*crash failures*), onde nós podem deixar de funcionar inesperadamente.
4. **Gerar e processar transações:** Implementar a geração de transações aleatórias, simulando a atividade de clientes, e garantir que essas transações são incluídas na blockchain de forma consistente.
5. **Demonstrar o consenso entre os nós:** Verificar que todos os nós alcançam consenso sobre o estado final da blockchain, mesmo na presença de falhas e atrasos na rede.

Segunda fase do projeto envolve:

1. **Lidar com nós atrasados e perda de épocas:** Garantir que o sistema continua a operar corretamente mesmo quando alguns nós perdem épocas devido a atrasos ou falhas temporárias.

2. **Implementar *crash-recovery*:** Permitir que nós que falharam possam reingressar na rede, recuperando o estado da blockchain e participando novamente no processo de consenso sem comprometer a consistência global.
3. **Gerir *forks* na blockchain:** Demonstrar a capacidade do sistema em lidar com *forks*, garantindo que os nós convergem eventualmente para uma única cadeia de blocos.
4. **Assegurar a tolerância a faltas:** Manter a robustez e a consistência do sistema na presença dos desafios acima mencionados.

3. Fundamentação Teórica

Nesta secção, serão apresentados os conceitos teóricos fundamentais que sustentam a implementação da primeira fase e da segunda fase do projeto, com ênfase nos desafios adicionais introduzidos e nas adaptações necessárias ao protocolo Streamlet.

3.1. Protocolos de Consenso em Sistemas Distribuídos

O consenso é um problema fundamental em sistemas distribuídos, essencial para garantir que todos os nós concordem no estado do sistema, mesmo quando alguns componentes podem falhar ou agir de forma maliciosa.

Protocolos de consenso são algoritmos que permitem que os nós em um sistema distribuído cheguem a um acordo sobre um valor ou sequência de valores. Estes protocolos são cruciais para a consistência e confiabilidade de sistemas distribuídos, especialmente em aplicações críticas como bases de dados distribuídas, sistemas de ficheiros distribuídos e blockchain.

3.1.1Propriedades do Consenso

- **Validade:** Se todos os nós corretos propuserem o mesmo valor, então esse valor é decidido.
- **Integridade:** Nenhum nó decide mais do que um valor.
- **Acordo:** Todos os nós corretos decidem o mesmo valor.
- **Terminação:** Todos os nós corretos eventualmente tomam uma decisão.

3.1.2 Desafios e Considerações

- **Assíncronia da Rede:** A imprevisibilidade dos atrasos na comunicação pode dificultar a coordenação.
- **Falhas de Nós:** Os nós podem falhar, comprometendo o processo de consenso.
- **Impossibilidade de FLP:** O teorema de Fischer, Lynch e Paterson (FLP) demonstra que não é possível alcançar consenso determinístico em sistemas completamente assíncronos com possibilidade de falhas de paragem.

3.1.3 Soluções Comuns

- **Protocolos Baseados em Líder (Paxos):** Um nó é designado como líder para coordenar o consenso.
- **Protocolos de Votação:** Os nós votam em propostas, e um quórum é necessário para avançar.
- **PBFT (Practical Byzantine Fault Tolerance):** Protocolo que lida com falhas bizantinas, permitindo que o sistema funcione corretamente mesmo que alguns nós sejam maliciosos.

3.2. Protocolo Streamlet

O Streamlet é um protocolo de consenso conhecido pela sua simplicidade e eficiência, tornando-o adequado para implementações práticas. Este é projetado para blockchains em sistemas tolerantes a faltas de *crash*, na qual opera em épocas incrementais, onde cada época é composta por várias rondas.

3.2.1 Funcionamento do Streamlet

1. **Operação em Épocas:** O tempo é dividido em épocas numeradas sequencialmente. Em cada época, um líder é responsável por propor um bloco.
2. **Rotação de Líderes:** O líder muda a cada época, geralmente seguindo uma ordem predefinida, o que evita a centralização e reduz o impacto de falhas de um único líder.
3. **Proposta e Votação de Blocos:**
    - **Proposta:** O líder da época propõe um novo bloco, que inclui transações pendentes e referencia o *hash* do bloco anterior.
    - **Votação:** Os nós recebem a proposta e decidem se votam no bloco com base em regras predefinidas (se o bloco estende a cadeia mais longa conhecida).
4. **Notarização de Blocos:** Um bloco é considerado notarizado se receber votos de uma maioria dos nós (mais de n/2). A notarização indica que o bloco foi aceito por uma maioria e pode ser considerado seguro para inclusão na blockchain.
5. **Finalização de Blocos:** Um bloco é finalizado quando existem três blocos notarizados consecutivos. A finalização garante que o bloco e todos os blocos anteriores na cadeia não serão revertidos, proporcionando imutabilidade.

3.2.2 Propriedades do Streamlet

1. **Simplicidade:** Fácil de implementar e compreender, o que reduz a possibilidade de erros na implementação.
2. **Eficiência:** Requer apenas duas rondas de comunicação por época (proposta e votação), o que reduz a latência.
3. **Tolerância a Faltas de Paragem:** Funciona corretamente na presença de até “t” falhas,

desde que n ≥ 2t + 1, onde “n” é o número total de nós.

3.2.3 Desafios e Considerações

- **Atrasos na Rede:** Embora o protocolo seja robusto, atrasos significativos na rede podem afetar o desempenho e a rapidez com que o consenso é alcançado.
- **Sincronização de Épocas:** Garantir que todos os nós avançam pelas épocas de forma sincronizada é importante para o funcionamento correto do protocolo.
- **Escalabilidade:** Em redes muito grandes, o número de mensagens trocadas pode aumentar significativamente, o que pode exigir otimizações adicionais.

3.2.4 Adaptação do Protocolo Streamlet

Para lidar com os desafios enfrentados na segunda fase do projeto, o protocolo Streamlet anterior foi adaptado, introduzindo mecanismos adicionais que permitem ao sistema ser mais robusto em cenários reais.

3.2.4.1. Mecanismos de Reingresso e Sincronização

- **Deteção de Reingresso**: Os nós devem ser capazes de identificar se estão a reingressar na rede após uma falha, verificando a existência de um estado local persistido.
- **Solicitação de Blocos em Falta**: Nós que reingressam solicitam aos outros nós os blocos que perderam durante a sua ausência, utilizando mensagens específicas.
- **Atualização do Estado Local**: Os nós integram os blocos recebidos na sua blockchain local (ficheiro JSON), garantindo a consistência com a rede.

3.2.4.2. Gestão de Forks

- **Deteção de *Forks***: Implementação de mecanismos para identificar a existência de múltiplas cadeias válidas.
- **Seleção da Cadeia Mais Longa**: Adotar a cadeia mais longa como critério para resolver *forks*, conforme previsto no protocolo Streamlet.
- **Convergência Eventual**: Assegurar que, mesmo na presença de *forks*, todos os nós convergem para a mesma cadeia.

3.2.4.3. Períodos de Confusão

- **Simulação de Confusão**: Introdução de períodos onde múltiplos líderes podem propor blocos simultaneamente, criando *forks* intencionalmente para testar a robustez do sistema.
- **Lógica de Liderança Adaptada**: Durante estes períodos, a seleção do líder é alterada

para provocar a situação desejada.

3.3. Desafios na Implementação da Segunda Fase

3.3.1 Nós Atrasados e Perda de Época

Em sistemas distribuídos, é comum que alguns nós experimentem atrasos na comunicação ou processamento, levando à perda de épocas. Isto pode ocorrer devido a latências na rede, sobrecarga de recursos ou falhas temporárias. A perda de épocas por parte de nós pode causar inconsistências na blockchain, uma vez que esses nós não participam no consenso durante esses períodos.

3.3.1 *Crash-Recovery*

Os nós podem sofrer falhas de paragem (*crash*) e posteriormente recuperar (*reboot*). É crucial que, ao reingressarem na rede, consigam atualizar o seu estado de forma consistente com o resto do sistema, recuperando os blocos perdidos durante o período de ausência. Sem mecanismos adequados de *crash-recovery*, estes nós permaneceriam desatualizados, comprometendo a integridade e a consistência do sistema.

3.3.1 *Forks* na Blockchain

Os *forks* ocorrem quando há divergência na cadeia de blocos, resultando em múltiplas cadeias válidas. Isto pode acontecer devido a atrasos, perda de mensagens ou proposição simultânea de blocos por diferentes líderes. A gestão de *forks* é essencial para garantir que todos os nós convergem eventualmente para uma única cadeia consistente, mantendo a integridade da blockchain.

4. Organização e Estrutura do Projeto

Para complementar a segunda fase do projeto, a implementação do protocolo Streamlet foi feita de forma cuidadosa, eficiente e modular. O projeto foi organizado em diferentes módulos e scripts separados, cada um responsável por uma componente específica do sistema.

Esta abordagem de modularização facilita a manutenção, testes e futuras extensões do sistema.

4.1. Estrutura de Ficheiros

- **block.py:** Define a classe ***Block***, responsável por representar **blocos na blockchain**. Este módulo contém métodos aprimorados para serialização e desserialização, que permite a

criação e manipulação de blocos, incluindo o cálculo de *hashes* e a representação de blocos.

- **transaction.py:** Contém a classe ***Transaction***, que representa as **transações na rede**. Este módulo contém métodos para conversão entre objetos e dicionários que é crucial para simular a atividade de clientes que enviam transações para a rede.
- **message.py:** Inclui as classes ***Message*** e ***MessageType***, que são utilizados para **criar** e **gerir** as **diferentes mensagens** trocadas entre os nós, como propostas, votos e transações. Este foi ampliado para suportar novos tipos de mensagens necessárias para *crash-recovery* e gestão de *forks* (*QUERY_MISSING_BLOCKS* e *RESPONSE_MISSING_BLOCKS*).
- **node.py:** Define a classe ***Node***, que representa um **nó na rede blockchain.** Este módulo é responsável por gerir a interação entre nós, incluindo a proposta, votação e notarização de blocos. O código foi adaptado para suportar funcionalidades avançadas, incluindo:
    1. **Reingresso e Recuperação:** Implementação de métodos que permitem aos nós reingressarem na rede após falhas ou períodos de desconexão. Isto inclui a deteção de reingresso, o envio de pedidos de blocos em falta (*QUERY_MISSING_BLOCKS*) e a integração desses blocos recebidos através de respostas (*RESPONSE_MISSING_BLOCKS).*
    2. **Gestão de *Forks*:** Lógica para lidar com múltiplos blocos notarizados na mesma época. O nó escolhe a cadeia mais longa (ou a de maior peso, caso aplicável) e atualiza o seu estado local para garantir consistência com a rede.
    3. **Períodos de Confusão:** Seleção de líderes é adaptada para simular períodos de confusão, alternando entre métodos previsíveis (baseados em *seeds*) e aleatórios. Isto aumenta a resiliência e permite testar cenários onde múltiplos líderes podem surgir simultaneamente.
    4. **Persistência:** Mecanismos para guardar e carregar o estado da blockchain, incluindo blocos notarizados, transações pendentes e votos.
    5. **Interação e Mensagens:** Utiliza as classes definidas em message.py para criar, enviar e receber mensagens relacionadas com propostas de blocos, votos, transações e recuperação de blocos em falta.
    6. **Criação e Gestão de Transações:** Gera transações aleatórias em cada época e gere a inclusão de transações em blocos propostos. As transações são verificadas para evitar duplicação, tanto na cadeia local como nas listas de transações pendentes.
    7. **Sincronização de Rede:** Integra funcionalidades de *broadcast* para comunicar com outros nós, permitindo a disseminação eficiente de blocos, transações e votos. Cada nó mantém uma lista de portas de comunicação para interagir com os restantes nós na rede.
    8. **Simulação de Épocas e Liderança:** Garante a execução sequencial de épocas, com intervalos definidos, onde os líderes são responsáveis por propor novos blocos.
- **node_script.py:** *Script* utilizado para **iniciar** e **executar cada nó individualmente**. Permite que cada nó seja executado como um processo separado, simulando um ambiente

distribuído real. Foi adaptado para suportar o reingresso e simulação de atrasos.

- **network_info.json:** **Ficheiro de configuração** que define os **parâmetros da rede**, incluindo portas, tempos de início e períodos de confusão (*confusion_start* e *confusion_duration*).
- **blockchain_[i].json:** **Ficheiros com a blockchain local de cada nó** que é utilizado para guardar a blockchain ao fim de cada época. Os ficheiros são independentes, cada nó utilizando o seu criado. Os ficheiros são utilizados para o *crash recovery* quando for necessário.
- **delete_blockchain_files.py**: Ficheiro responsável por **eliminar todos os antigos ficheiros blockchain_[i].json** presentes no diretório de execução. Este script torna mais simples a limpeza do estado persistido, facilitando a nova execução de testes sem resíduos de execuções anteriores.

4.2. Tecnologias e Ferramentas

1. **Python3:** A Linguagem de programação escolhida para implementar todos os módulos e *scripts* foi o Python, na qual oferece diversas bibliotecas integradas para *threading*, *sockets* e serialização, que são essenciais para a criação deste projeto.
2. **Bibliotecas Padrão:**
   - **threading**: Para criar e gerir *threads*, permitindo a execução concorrente de nós.
   - **socket:** Para comunicação em rede entre os nós através de *sockets* TCP.
   - **Pickle** e **Json:** Para serialização e desserialização de objetos complexos, facilitando o envio de mensagens pela rede.
   - **hashlib:** Para cálculo de *hashes* SHA1, garantindo a integridade dos blocos.
   - **Datatime e time:** Para a gestão de tempos de simulação de atrasos.
3. **Subprocess:** Utilizada para iniciar os nós como processos separados, permitindo uma simulação mais realista de um ambiente distribuído.
4. **Estruturas de Dados e Mecanismos de Sincronização:** Utilização de *locks* (*threading.Lock*) para sincronizar o acesso a recursos partilhados entre *threads*, evitando condições de corrida.

5. Detalhes da Implementação

Nesta secção, serão detalhados os componentes individuais do sistema, explicando as decisões de implementação e como cada módulo contribui para o funcionamento geral do protocolo.

5.1. Módulo block.py

O módulo **block.py** define a classe ***Block***, responsável por representar cada bloco na blockchain.

5.1.1 Atributos

- **epoch*:*** Indica a época em que o bloco foi proposto. Essencial para sincronização e ordenação dos blocos.
- **previous_hash*:*** Contém o *hash* do bloco anterior, estabelecendo a ligação na cadeia e garantindo a integridade sequencial.
- **transactions*:*** Lista de transações incluídas no bloco. Representa as operações que os nós concordaram em adicionar à blockchain.
- **length:** Representa a altura do bloco na blockchain, ou seja, a posição do bloco na cadeia. É utilizado para determinar a cadeia mais longa.
- **hash:** *Hash* único do bloco, calculado a partir dos seus dados. Se não for fornecido, é computado automaticamente.
- **votes:** Número de votos que o bloco recebeu durante o processo de consenso.

5.1.2 Métodos

- **compute_hash():** Calcula o *hash* do bloco considerando todos os atributos relevantes, incluindo as transações e o *timestamp*. Este *hash* é crucial para a identificação única do bloco e para a integridade da cadeia.
- **__repr__():** Fornece uma representação em *string* do bloco, útil para depuração e *logs*.
- **to_dict()** e **from_dict()**: Permitem serializar e desserializar blocos, ou seja, convertem a representação interna do bloco num dicionário ou, inversamente, recriam o bloco a partir de um dicionário recebido. Esta funcionalidade é essencial para facilitar a transmissão de blocos pela rede (via JSON) e o seu armazenamento persistente.

5.1.3 Decisões de Implementação

A inclusão do *length* é fundamental para a implementação do protocolo Streamlet, pois permite que os nós determinem qual a cadeia mais longa e atualizem a sua blockchain local de acordo. O cálculo do *hash* do bloco inclui as transações, o que garante que qualquer alteração nos dados do bloco resulta num *hash* diferente, mantendo a segurança e integridade da blockchain.

5.2. Módulo transaction.py

Define a classe ***Transaction***, que representa uma transação entre um remetente e um destinatário.

5.2.1 Atributos

- **sender:** Identificador do remetente da transação.
- **receiver:** Identificador do destinatário da transação.
- **tx_id:** Identificador único da transação, garantindo a sua unicidade na rede.
- **amount:** Valor da transação.

5.2.2 Métodos

- ***__eq__():*** Define a igualdade entre transações, permitindo que sejam comparadas com base no remetente e no ID da transação.
- ***__hash__():*** Permite que transações sejam utilizadas em conjuntos e dicionários, prevenindo duplicações.
- ***to_dict():*** Converte a transação num dicionário para facilitar a sua serialização e envio.
- ***from_dict():*** Cria uma instância de *Transaction* a partir de um dicionário, permitindo desserializar dados recebidos e recriar a transação.

5.2.3 Decisões de Implementação

A implementação de **'__eq__'** e **'__hash__'** é crucial para evitar a inclusão de transações duplicadas na lista de transações pendentes ou nos blocos. O uso de atributos simples (inteiros) para ***sender*** e ***receiver*** facilita a simulação e geração de transações aleatórias.

5.3. Módulo message.py

Este módulo define a classe ***Message*** e ***MessageType***, que são utilizadas para criar e gerir as diferentes mensagens trocadas entre os nós.

5.3.1 Tipos de Mensagens

- **PROPOSE:** Utilizada pelo líder para propor um novo bloco.
- **VOTE:** Enviada pelos nós para votar num bloco proposto
- **ECHO_TRANSACTION:** Utilizada para dar *echo* uma transação recebida, garantindo que todos os nós a recebem.
- **QUERY_MISSING_BLOCKS:** Mensagem para pedir os blocos com a ultima época antes do crash.
- **RESPONSE_MISSING_BLOCKS:** Mensagem para responder ao **QUERY_MISSING_BLOCKS** com os blocos em falta

5.3.2 Métodos

A utilização de **pickle** para serialização permite enviar objetos complexos (como blocos e transações) de forma simples e eficiente. O prefixo de comprimento nos dados serializados garante que a desserialização das mensagens é feita corretamente, mesmo que os dados sejam recebidos em pacotes separados.

- ***serialize():*** Converte o conteúdo da mensagem (blocos, transações) num formato JSON pronto a enviar pela rede.
- ***deserialize_from_socket()***: Faz a leitura dos dados provenientes do *socket*, reconstrói o objeto *Message* e valida o seu conteúdo antes de o encaminhar para processamento.

5.3.3 Decisões de Implementação

A utilização de pickle para serialização permite transmitir objetos complexos. O prefixo de comprimento nos dados serializados garante a correta leitura dos dados do *socket*.

5.3.4 Adaptações de Implementação Segunda Fase

As funções de **serialização** e **desserialização** foram adaptadas para suportar estes novos tipos, garantindo a correta transmissão e receção dos dados. Novos tipos de mensagens foram adicionados:

- **QUERY_MISSING_BLOCKS**: Mensagem utilizada por nós que reingressam para solicitar blocos em falta, indicando a última época conhecida.
- **RESPONSE_MISSING_BLOCKS**: Mensagem de resposta contendo os blocos solicitados pelo nó que reingressa.

5.4. Módulo node.py

O módulo **node.py** implementa a classe ***Node***, que representa cada nó participante na rede distribuída.

5.4.1 Atributos

- **node_id:** Identificador único do nó.
- **total_nodes:** Número total de nós na rede (para calcular quóruns e maioria).
- **blockchain:** Cadeia local de blocos notarizados, compõem a blockchain local do nó.
- **pending_transactions:** Lista de transações pendentes, que ainda não foram incluídas.
- **vote_count:** Armazena o número de votos que cada bloco recebeu (indicador no bloco quando atinge a maioria de votos para ser notarizado).
- **voted_senders:** Registo de nós que já votaram para cada bloco (evitar duplicados).
- **notarized_blocks:** Dicionário de blocos notarizados.
- **notarized_tx_ids:** Armazena os IDs das transações que já foram notarizadas (evita inclusão de transações duplicada na blockchain).
- **lock:** Lock utilizado para sincronizar o acesso a recursos partilhados entre threads

5.4.2 Métodos

- ***propose_block():*** Propõe um novo bloco com as transações se o nó for o líder da época.
- ***vote_on_block():*** Vota num bloco proposto se estender a cadeia notarizada mais longa conhecida pelo nó.
- ***notarize_block():*** Notariza um bloco quando este recebe votos suficientes, e notifica os outros nós.

- ***finalize_blocks():*** Verifica se existem três blocos notarizados consecutivos para finalizar blocos.
- ***determine_epoch_for_transaction():*** Define o epoch para uma transação.
- ***get_chain_to_block():*** Lista de blocos necessários para formar a cadeia de blocos até um bloco específico, caminhando para trás a partir do bloco dado até encontrar um bloco já notarizado na blockchain.
- ***get_longest_notarized_chain():*** Obtém o último bloco da cadeia notarizada mais longa.
- ***add_transaction():*** Adiciona uma nova transação à lista de transações pendentes.
- ***broadcast_message():*** Transmite uma mensagem para todos os outros nós na rede.
- ***display_blockchain():*** Exibe o estado atual da blockchain do nó.

5.4.3 Decisões de Implementação

A classe **Node** herda de **threading.Thread**, permitindo que cada nó seja executado numa *thread* separada, facilitando a execução concorrente. *Locks* são utilizados para garantir que operações críticas são realizadas de forma segura, evitando condições de corrida. O método *finalize_blocks*() implementa a lógica de finalização de blocos do protocolo Streamlet, garantindo a imutabilidade dos blocos finalizados.

5.4.4 Adaptações de Implementação Segunda Fase

5.4.4.1. Reingresso e Recuperação

- **Deteção de Reingresso**: Ao iniciar, o nó verifica se possui uma blockchain guardada em ficheiro. Se existir, interpreta-se que o nó está a reingressar após uma falha.
- **Solicitação de Blocos em Falta**: O nó envia uma mensagem *QUERY_MISSING_BLOCKS* aos outros nós, indicando a última época que possui obtido pelo ficheiro **blockchain_[i].json** que guarda a blockchain ao fim de cada época.
- **Resposta a Pedidos**: Os nós ativos respondem com *RESPONSE_MISSING_BLOCKS*, enviando os blocos das épocas em falta.
- **Atualização do Estado Local**: O nó reingressado integra os blocos recebidos na sua blockchain e ativa a flag **recovery_completed** para não utilizar as mensagens dos outros nós, atualizando o estado e participando novamente no consenso entrando no ciclo for das épocas com a ultima época dada na blockchain.

5.4.4.2. Gestão de Forks

- **Construção da Cadeia Mais Longa**: Métodos para percorrer os blocos notarizados e construir a cadeia mais longa possível, seguindo os *hashes* dos blocos anteriores.
- **Deteção de *Forks***: Identificação de múltiplos blocos notarizados para a mesma época, indicando a presença de *forks*.
- **Resolução de *Forks***: Adoção da cadeia mais longa como a blockchain válida,

assegurando a convergência eventual dos nós.

5.4.4.3. Períodos de Confusão

- **Simulação de Confusão**: Implementação de períodos específicos onde múltiplos líderes podem propor blocos simultaneamente, criando *forks* intencionalmente.
- **Lógica de Liderança Adaptada**: Durante estes períodos, a seleção do líder é alterada para provocar a situação desejada, permitindo testar a robustez do sistema.

Durante estes períodos de confusão, para além da alteração da lógica de seleção do líder, foram também introduzidos **atrasos**, **omissões** e **reordenações** aleatórias de mensagens. Através das funções ***broadcast_message()*** e ***process_message_queue()***, o sistema pode simular perdas de pacotes (*drop*), atrasos artificiais (*delays*) e *reorderings* de mensagens. Este comportamento reforça a resiliência do programa ao testar a capacidade de convergir para um estado consistente mesmo em cenários realistas de falhas e condições adversas.

5.4.4.4. Persistência e Carregamento da Blockchain

- **Guardar o Estado**: Após cada época, a blockchain é guardada num ficheiro (*blockchain_[node_id].json*), permitindo a recuperação em caso de falha.
- **Carregar o Estado**: Ao iniciar, o nó tenta carregar a blockchain guardada. Se existir, continua a partir daí; caso contrário, inicia com o bloco génese.

5.5. Script node_script.py

O script **node_script.py** é utilizado para iniciar e executar cada nó individualmente.

5.5.1 Funcionalidades

- **Processamento de Argumentos:** Lê os argumentos de linha de comandos para configurar o nó (ID, número total de nós, porta, etc.).
- **Inicialização do Nó:** Cria uma instância da classe Node com os parâmetros fornecidos.
- **Sinalização de Prontidão:** Envia um sinal para o gestor de rede indicando que o nó está pronto para iniciar.
- **Escuta de Mensagens:** Inicia um loop que escuta por conexões na porta designada e processa as mensagens recebidas.
- **Processamento de Mensagens:** Executa ações específicas com base no tipo de mensagem recebida (propor blocos, votar, processar transações).

5.5.2 Decisões de Implementação

Cada nó é executado como um processo separado, o que simula um ambiente distribuído real onde os nós operam independentemente. A comunicação via *sockets* TCP permite que os nós se comuniquem de forma fiável e sem necessidade de bibliotecas adicionais.

5.5.3 Adaptações de Implementação Segunda Fase

Foram implementadas novas adaptações neste método de forma a ser possível a **inicialização com reingresso**, na qual suporta a opção de reingresso, permitindo simular nós que falharam e estão a reingressar na rede. E também a **simulação de atrasos e confusão**, onde se adicionou parâmetros para definir períodos de confusão e simular atrasos nos nós.

6. Testes e Validação

Para assegurar que a implementação funciona corretamente e atende aos objetivos definidos, foram realizados diversos testes abrangentes e detalhados.

6.1 Execução com Diferentes Números de Nós

Objetivo: Verificar o comportamento do sistema em redes de diferentes dimensões.

Procedimento: O sistema foi executado com 3, 5 e 7 nós, ajustando os parâmetros de linha de comandos no main.py.

Resultados: O protocolo funcionou corretamente em todos os casos, com os nós alcançando consenso sobre a blockchain final. Observou-se um aumento no tempo de consenso e no número de mensagens trocadas com o aumento do número de nós, conforme esperado.

6.2 Simulação de Falhas de Nós

Objetivo: Testar a tolerância a faltas de paragem, garantindo que o sistema continua a funcionar corretamente mesmo quando alguns nós falham.

Procedimento: Durante a execução, alguns processos de nós foram terminados manualmente utilizando comandos do sistema operativo.

Resultados: O sistema continuou a operar corretamente, desde que o número de nós ativos fosse suficiente para formar uma maioria ($n \geq 2t + 1$). Os nós restantes alcançaram consenso e finalizaram blocos conforme esperado. As falhas de nós causaram atrasos na propagação de mensagens, mas não comprometeram a integridade do sistema.

6.3 Verificação da Finalização de Blocos

Objetivo: Confirmar que os blocos são finalizados apenas quando as condições do protocolo são satisfeitas.

Procedimento: Analisaram-se os *logs* dos nós para verificar quando os blocos eram finalizados, assegurando que apenas ocorria após três blocos notarizados consecutivos.

Resultados: Os blocos foram finalizados corretamente seguindo as regras do protocolo

Streamlet. Todos os nós concordaram na sequência de blocos finalizados, demonstrando consistência.

6.4 Consistência da Blockchain Entre Nós

Objetivo: Garantir que todos os nós possuem a mesma versão da blockchain após a execução.

Procedimento: Utilizou-se o comando DISPLAY_BLOCKCHAIN para cada nó após a conclusão das épocas e compararam-se as blockchains.

Resultados: As blockchains de todos os nós eram idênticas, indicando que o consenso foi alcançado com sucesso. As transações incluídas nos blocos finalizados eram consistentes entre os nós.

6.5 Geração e Propagação de Transações

Objetivo: Testar a geração contínua de transações e a sua inclusão nos blocos.

Procedimento: Observou-se a geração de transações pela *thread* dedicada e a sua propagação através das mensagens *TRANSACTION* e *ECHO_TRANSACTION.*

Resultados: As transações foram geradas e propagadas corretamente entre os nós. Não ocorreram duplicações de transações nos blocos, graças à implementação adequada dos métodos '*__eq__()*' e '*__hash__()*' na classe *Transaction*.

6.6 Simulação de *Crash-Recovery*

Objetivo: Verificar se os nós que falham podem reingressar na rede, recuperando o estado da blockchain e participando novamente no consenso.

Procedimento: Durante a execução, alguns nós foram terminados manualmente. Após um período, os nós foram reiniciados com a opção de reingresso. Observou-se o processo de recuperação e sincronização com a rede.

Resultados: Os nós reingressados enviaram mensagens *QUERY_MISSING_BLOCKS* corretamente. Os nós ativos responderam com *RESPONSE_MISSING_BLOCKS*, fornecendo os blocos em falta. Os nós reingressados integraram os blocos recebidos e retomaram a participação no consenso sem problemas.

6.7 Períodos de Confusão e *Forks*

Objetivo: Testar a capacidade do sistema em lidar com *forks* e convergir para uma única cadeia.

Procedimento: Configuraram-se períodos de confusão no ficheiro de configuração, onde múltiplos líderes estão ativos simultaneamente. Observou-se a criação de *forks* na blockchain durante esses períodos.

Resultados: *Forks* foram criados intencionalmente, com diferentes nós possuindo cadeias

distintas. Após o período de confusão, os nós utilizaram os mecanismos de seleção da cadeia mais longa para convergir para uma única cadeia. A consistência eventual foi alcançada, com todos os nós sincronizados.

6.8 Nós Atrasados e Perda de Épocas

Objetivo: Verificar o comportamento do sistema quando alguns nós sofrem atrasos e perdem épocas.

Procedimento: Simularam-se atrasos na comunicação e processamento para alguns nós, fazendo com que perdessem épocas. Observou-se como os nós atrasados recuperaram e atualizaram o seu estado.

Resultados: Os nós atrasados conseguiram sincronizar o seu estado com a rede, solicitando blocos em falta quando necessário. A participação no consenso foi retomada sem comprometer a consistência do sistema.

6.9 Consistência Global e Resolução de *Forks*

Objetivo: Garantir que todos os nós possuem a mesma versão da blockchain após a execução.

Procedimento: Após a conclusão das épocas, utilizou-se o comando para exibir a blockchain em cada nó. Compararam-se as blockchains de todos os nós.

Resultados: As blockchains de todos os nós eram idênticas, indicando que o consenso foi alcançado com sucesso. Os *forks* foram resolvidos corretamente, demonstrando a eficácia dos mecanismos implementados.

6.10 Análise de Desempenho

Objetivo: Avaliar o desempenho do sistema em termos de latência e utilização de recursos.

Procedimento: Mediram-se os tempos de execução das épocas e observaram-se os recursos do sistema (CPU, memória).

Resultados: O sistema apresentou um desempenho adequado para os objetivos do projeto. A latência aumentou com o número de nós, mas manteve-se dentro de limites aceitáveis. A utilização de recursos manteve-se eficiente, mesmo com o aumento da complexidade devido às novas funcionalidades.

7. Conclusão

A implementação do protocolo Streamlet neste projeto permitiu atingir com sucesso os objetivos propostos, proporcionando uma compreensão aprofundada dos desafios e soluções associados

ao consenso em sistemas distribuídos tolerantes a faltas.

7.1 Aprendizagens e Desafios

Com este projeto retiramos como aprendizagem a implementação do protocolo desde a base permitiu uma compreensão detalhada dos seus mecanismos, vantagens e limitações. O projeto reforçou competências na utilização de *threads*, processos e *sockets* em Python, essenciais para o desenvolvimento de sistemas distribuídos. A simulação de falhas de nós e a observação do comportamento do sistema nestas condições evidenciaram a importância de protocolos robustos e da implementação cuidadosa de mecanismos de redundância e consenso.

Ao longo do desenvolvimento deste projeto a garantia que as mensagens eram recebidas e processadas na ordem correta foi um desafio, exigindo cuidados na implementação dos métodos de comunicação. A necessidade de sincronização entre *threads* e processos levou à implementação de *locks* e à atenção especial para evitar condições de corrida. E por fim, a dificuldade de implementar mecanismos para lidar com falhas de comunicação e erros inesperados foi crucial para a robustez do sistema.

A implementação da segunda fase do projeto permitiu atingir com sucesso os objetivos propostos, demonstrando a capacidade do sistema em lidar com desafios comuns em sistemas distribuídos reais. Através das adaptações ao protocolo Streamlet, o sistema mostrou-se robusto na presença de nós atrasados, perda de épocas, *crash-recovery* e *forks*, assegurando a consistência e a convergência para uma única cadeia de blocos. Os desafios enfrentados reforçaram a compreensão dos mecanismos necessários para manter a integridade e a fiabilidade em sistemas distribuídos tolerantes a faltas. A implementação prática destes conceitos permitiu consolidar conhecimentos teóricos e desenvolver competências técnicas relevantes na área.

8. Referências

[1] Alysson Bessani (2024). Documentação sobre Tolerância a Faltas Distribuída no Moodle.

[2] Benjamin Y. Chan and Elaine Shi. Diego. Streamlet: Textbook Streamlined Blockchains. Proc. of the 2nd ACM Conference on Advances in Financial Technologies (AFT ’20). 2020.