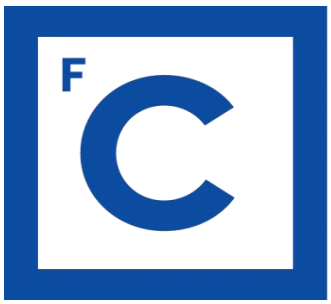

Ciências ULisboa

Faculdade de Ciências da Universidade de Lisboa

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Mestrado em Engenharia Informática

RELATÓRIO

Tolerância a Faltas Distribuída

Projeto: Streamlet consensus algorithm (Phase 1)

Rodrigo Craveiro Rodrigues (Nº64370)

Denis Ungureanu (Nº56307)

Ana Luís (Nº53563)

Professor: **Doutor Alysson Bessani**

1º Semestre Letivo 2024/2025

novembro 2024

Índice

1. Introdução

O projeto foi desenvolvido no âmbito da disciplina de Tolerância a Faltas Distribuída, na qual este relatório descreve a implementação de um sistema distribuído tolerante a faltas utilizando o protocolo de consenso **Streamlet**.

2. Objetivos

O objetivo principal do projeto é implementar o proposto protocolo de consenso Streamlet para alcançar consenso num sistema distribuído, para simular o funcionamento de uma blockchain em um ambiente distribuído tolerante a faltas. 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.

Fase 1 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.

3. Fundamentação Teórica

Nesta secção, serão apresentados os conceitos teóricos fundamentais que sustentam a implementação do projeto, com ênfase nos sistemas distribuídos tolerantes a faltas, protocolos de consenso e o 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 blockchains.

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 “f” falhas, desde que $n \geq 2f + 1$, 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.

4. Organização e Estrutura do Projeto

Para implementar o protocolo Streamlet de forma 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*, que representa um bloco na blockchain. Este módulo é responsável pela criação e manipulação de blocos, incluindo o cálculo de *hashes* e a representação de blocos.
- **transaction.py:** Contém a classe *Transaction*, que representa uma transação entre dois participantes. Este módulo é crucial para simular a atividade de clientes que enviam transações para a rede.
- **message.py:** Inclui a classe *Message* e *MessageType*, que são utilizados para criar e gerir as diferentes mensagens trocadas entre os nós, como propostas, votos e transações.
- **node.py:** Inclui a classe *Message* e *MessageType*, que são utilizados para criar e gerir as diferentes mensagens trocadas entre os nós, como propostas, votos e transações.
- **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.
- **streamletnetwork.py:** Gere a rede de nós, coordenando as épocas, a geração de transações e a comunicação entre os nós. Este módulo é responsável por iniciar a rede, definir o líder de cada época e controlar o fluxo geral do protocolo.
- **main.py:** Ponto de entrada do programa. Configura os parâmetros iniciais, como o número de nós e o valor de Δ, e inicia a execução do protocolo.

4.2. Tecnologias e Ferramentas

1. **Python 3:** 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:** 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.
3. **Subprocessos:** A biblioteca subprocess é 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*.

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_NOTARIZE:** Utilizada para dar *echo* a notarização de um bloco para outros nós. **ECHO_TRANSACTION:** Utilizada para dar *echo* uma transação recebida, garantindo que todos os nós a recebem.
- **START_PROPOSAL:** Mensagem para iniciar uma proposta numa nova época.
- **TRANSACTION:** Contém uma nova transação gerada.
- **DISPLAY_BLOCKCHAIN:** Comando para os nós exibirem o estado atual da sua blockchain.

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.

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*.

5.4. 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.

5.5. 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.

5.6. Módulo streamletnetwork.py

O módulo **streamletnetwork.py** gere a rede de nós, coordenando as épocas, a geração de transações e a comunicação.

5.6.1 Atributos

- **num_nodes:** Número total de nós na rede.
- **delta*:*** Parâmetro de atraso da rede (Δ), utilizado para calcular a duração das épocas.
- **epoch_duration*:*** Duração de cada época (calculada como 2Δ).
- **leader**: Identificador do líder atual.
- **ports*:*** Lista de portas atribuídas a cada nó.
- **transaction_thread*:*** *Thread* responsável pela geração periódica de transações.
- **processes**: Lista de processos que executam os nós.

5.6.2 Métodos

- ***start_network():*** Inicia os nós e aguarda que estejam prontos.
- ***stop_network():*** Encerra os nós e a *thread* de geração transações.
- ***next_leader():*** Determina o próximo líder para a nova época.
- ***start_epoch():*** Inicia uma nova época, solicitando ao líder que proponha um bloco.
- ***run():*** Executa o protocolo pelo número especificado de épocas, controlando o fluxo.
- ***generate_random_transaction():*** Gera uma transação aleatória e envia-a para um nó.
- ***generate_transactions_periodically():*** Gera transações em intervalos regulares.

5.6.3 Decisões de Implementação

O líder é rotacionado de forma *round-robin* para distribuir a responsabilidade de proposta de blocos entre os nós. A *thread* de geração de transações simula a chegada contínua de novas transações à rede, como ocorreria num sistema real. A utilização de subprocess.Popen permite iniciar cada nó em um novo terminal, tornando a simulação mais próxima de um ambiente distribuído real.

5.7. Ficheiro main.py

O ficheiro **main.py** é o ponto de entrada do programa, responsável por configurar e iniciar a rede.

5.7.1 Funcionalidades

- **Processamento de Argumentos:** Permite personalizar a execução através de argumentos de linha de comandos (número de nós, total de épocas, valor de Δ).
- **Inicialização da Rede:** Cria uma instância da classe StreamletNetwork com os

parâmetros fornecidos.

- **Início da Execução:** Chama os métodos para iniciar a rede e executar o protocolo.
- **Encerramento Limpo:** Garante que todos os processos e *threads* são terminados corretamente após a conclusão.

5.7.2 Decisões de Implementação

A configuração via linha de comandos facilita a experimentação com diferentes parâmetros, permitindo estudar o comportamento do protocolo em diferentes cenários. O encerramento adequado da rede evita processos órfãos e libera recursos do sistema.

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 ($n \geq 2f + 1$). 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 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.

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.

Como desafios encontrados 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.

7.2 Trabalhos Futuros

Expandir na segunda fase do projeto para lidar com nós maliciosos, implementando protocolos como o PBFT. Melhorar a eficiência da comunicação e processamento, possivelmente através de mecanismos de compressão de mensagens ou protocolos de comunicação mais eficientes. E por fim, desenvolver uma interface gráfica ou ferramentas de monitorização para visualizar o estado da rede e da blockchain em tempo real, facilitando a análise e depuração.

8. Referências

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

[2] Guerraoui, R., & Wang, L. (2019). Streamlet: Textbook Streamlined Blockchain Consensus Protocol. arXiv preprint arXiv:1906.07374.

[3] Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). Sistemas Distribuídos: Conceitos e Projeto. FCA Editora.