universidade de aveiro

deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

Mecanismos de Tolerância a Ataques

tomas.matos@ua.pt, 108624

5 de janeiro de 2025

1 Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular Segurança em Sistemas de Larga Escala, pertencente ao Mestrado em Cibersegurança da Universidade de Aveiro. O objetivo principal consiste em criar mecanismos robustos de tolerância a ataques, projetados para serem compatíveis com arquiteturas baseadas em serviços e capazes de escalar para sistemas de grande dimensão.

Adicionalmente, o projeto investiga o uso de técnicas e ferramentas avançadas para abordar as complexidades inerentes à partilha de mensagens em ambientes distribuídos, com ênfase no impacto do sistema implementado. Estes ambientes exigem soluções eficazes para assegurar o consenso entre os participantes, um aspeto fundamental para garantir a fiabilidade e a segurança das operações.

Uma componente central do projeto é a aplicação de estratégias de Moving Target Defense (MTD), visando a redução da superfície de ataque (Attack Surface) em diferentes camadas do sistema.

Por fim, o trabalho enfrenta desafios críticos, incluindo a implementação de algoritmos de consenso distribuído que considerem cenários de falhas bizantinas, reforçando a resiliência e a integridade das operações em ambientes adversos.

2 Estado da Arte

Com o crescimento exponencial do número de dispositivos conectados e o aumento da complexidade das infraestruturas tecnológicas, a cibersegurança moderna enfrenta desafios cada vez mais sofisticados, especialmente no que diz respeito ao consenso e à tolerância a falhas. Diversos algoritmos e técnicas têm sido desenvolvidos para garantir que os sistemas mantenham a sua integridade e segurança, mesmo diante de ataques ou falhas.

As arquiteturas baseadas em serviços desempenham um papel central neste panorama, caracterizando-se pela sua modularidade, escalabilidade e capacidade de interoperabilidade. Estas características permitem a integração de soluções de cibersegurança de forma eficiente e adaptativa. Ferramentas e frameworks que operam segundo princípios de orquestração de serviços e computação distribuída têm demonstrado elevada eficácia na proteção de sistemas contra ataques direcionados e falhas sistémicas.

Uma das abordagens mais promissoras neste contexto é a combinação de técnicas de consenso e defesa dinâmica com arquiteturas de serviços. Protocolos de consenso, como o Paxos e o PBFT, asseguram que os sistemas distribuídos cheguem a acordos confiáveis mesmo em cenários adversos. O Paxos, amplamente utilizado em sistemas distribuídos, foca-se na consistência e na confiabilidade, enquanto o PBFT expande essas garantias ao lidar com falhas bizantinas, onde os nós podem apresentar comportamentos maliciosos.

Por outro lado, as estratégias de Moving Target Defense introduzem uma camada adicional de proteção ao alterar continuamente a superfície de ataque do sistema. Estas estratégias, como a mudança dinâmica de endereços de rede, a reconfiguração de recursos em tempo de execução e a redistribuição de dados sensíveis, tornam os sistemas imprevisíveis e mais difíceis de serem explorados por atacantes. Quando integradas em arquiteturas baseadas em serviços, essas técnicas reforçam ainda

mais a resiliência dos sistemas, garantindo uma defesa proativa e adaptativa contra ameaças emergentes.

A seguir, serão discutidos os protocolos de consenso mais relevantes, tal como estratégias de Moving Target Defense (MTD) que se aplicam a ambientes distribuídos.

2.1 Protocolos de Consenso

Os protocolos de consenso são fundamentais para garantir que os participantes de um sistema distribuído concordem sobre um único valor, mesmo na presença de falhas ou ataques. Dois dos protocolos mais conhecidos são o Paxos e o PBFT.

2.1.1 Paxos

O algoritmo de consenso Paxos foi desenvolvido por Leslie Lamport em 1989 e é amplamente utilizado em sistemas distribuídos, para garantir a consistência dos dados em presença de falhas. O Paxos é baseado no conceito de proposições, onde um conjunto de réplicas de um sistema distribui mensagens para votar sobre qual valor deve ser escolhido como valor consistente para o sistema. Paxos é um algoritmo eficiente quando se lida com um número pequeno de falhas, mas pode ser difícil de entender e implementar em larga escala devido à complexidade e ao número de mensagens necessárias para a comunicação entre os nós. Além disso, o Paxos não lida diretamente com falhas bizantinas, o que limita a sua utilização em cenários adversos.

2.1.2 PBFT (Practical Byzantine Fault Tolerance)

O PBFT é um protocolo de consenso desenvolvido para lidar com falhas bizantinas, onde os participantes podem agir de maneira arbitrária ou maliciosa. Ao contrário do Paxos, o PBFT permite que o sistema continue a operar corretamente mesmo que até um terço dos nós se comportem de forma maliciosa ou defeituosa. O protocolo PBFT requer que cada operação seja executada por meio de um processo de votação entre os nós, garantindo que o valor acordado seja consistente, mesmo quando alguns nós tentam comprometer o consenso. Este protocolo é particularmente útil em sistemas onde a confiança nos participantes é baixa e falhas bizantinas são uma preocupação. No entanto, o PBFT apresenta limitações em termos de escalabilidade, especialmente em sistemas com um grande número de nós, devido ao elevado número de mensagens necessárias para garantir o consenso.

2.1.3 Outros Protocolos Relevantes

Além do Paxos e PBFT, existem outros protocolos de consenso que são relevantes em cenários de sistemas distribuídos, como o Raft e o Zyzzyva.

O Raft foi desenvolvido como uma alternativa mais simples ao Paxos, oferecendo uma implementação mais compreensível sem comprometer a segurança. Este utiliza um modelo de líder para coordenar as operações do sistema, onde um líder é eleito entre os nós do sistema para gerenciar a replicação e assegurar que todos os nós concordem sobre o mesmo estado. Quando o líder falha, um novo líder é escolhido, permitindo que o sistema continue a funcionar.

Já o Zyzzyva é um protocolo de consenso tolerante a falhas bizantinas que, assim como o PBFT, busca assegurar a operação correta do sistema, mesmo em ambientes adversos. Este protocolo utiliza um modelo de duas fases: uma fase inicial onde as propostas são enviadas e uma fase final onde a decisão é tomada. Embora o Zyzzyva seja mais eficiente que o PBFT em termos de comunicação, ele assume que a maioria dos nós são confiáveis e que apenas um número limitado de falhas bizantinas pode ocorrer, o que pode ser uma limitação em cenários extremamente adversos.

2.1.4 Tabela Comparativa

Protocolo	Tipo de Falha Tolerada	Escalabilidade	Desempenho
Paxos	Falhas de nós (não bizantinas)	Moderada	Alto em cenários pequenos
PBFT	Falhas bizantinas	Baixa a moderada	Baixo devido à sobrecarga
Raft	Falhas de nós (não bizantinas)	Alta	Bom em pequenos clusters
Zyzzyva	Falhas bizantinas	Moderada	Bom em cenários otimizados

Tabela 1: Características gerais dos protocolos de consenso

Protocolo	Aplicações Comuns	Principais Limitações
Paxos	Sistemas distribuídos, bancos de dados	Complexidade na implementação; elevado número de mensagens.
PBFT	Blockchain, sistemas financeiros, sistemas críticos	Elevado número de mensagens; escalabilidade limitada.
Raft	Bancos de dados distribuídos, sistemas de logs	Menos eficiente em cenários de escrita intensiva ou alta latência.
Zyzzyva	Aplicações críticas com alta tolerância a falhas	Depende da suposição de poucas falhas simultâneas.

Tabela 2: Aplicações e limitações dos protocolos de consenso

2.2 Estratégias de Moving Target Defense (MTD)

As estratégias de Moving Target Defense (MTD) têm ganhado destaque como uma técnica eficaz para aumentar a segurança em sistemas distribuídos, tornando os alvos dos ataques mais dinâmicos e difíceis de prever ou explorar. As técnicas MTD visam alterar continuamente a superfície de ataque do sistema, dificultando que os atacantes localizem ou comprometam componentes críticos. Algumas das principais estratégias de MTD incluem:

2.2.1 Mudança de Endereços e Portas

Uma das abordagens mais comuns de MTD envolve a mudança dinâmica de endereços IP e portas, de forma que os atacantes não consigam localizar os alvos da rede facilmente. Esse tipo de defesa é útil em ambientes de redes distribuídas, onde a localização dos nós pode ser alterada periodicamente, dificultando a execução de ataques direcionados, como ataques DDoS (Distributed Denial of Service).

2.2.2 Reconfiguração de Recursos e Processos

A reconfiguração dinâmica de recursos e processos no nível da plataforma e da aplicação pode aumentar significativamente a resiliência do sistema a ataques. Isso inclui a alteração da configuração do sistema operativo ou a execução de tarefas em diferentes instâncias de máquinas virtuais. Esse tipo de técnica é eficaz para dificultar a exploração de vulnerabilidades, uma vez que a estrutura do sistema é alterada periodicamente, tornando os ataques mais difíceis de executar.

2.2.3 Encriptação e Redistribuição de Dados

A encriptação dinâmica e a redistribuição de dados são estratégias MTD relevantes para proteger os dados sensíveis. Ao alterar a forma como os dados são armazenados ou distribuídos dentro de um sistema, é possível tornar os dados inacessíveis para atacantes, mesmo que eles consigam penetrar em partes do sistema. A encriptação é uma das formas mais eficazes de proteger informações em ambientes distribuídos, e a redistribuição de dados assegura que os dados críticos não fiquem concentrados em um único ponto vulnerável.

2.2.4 Mudança de Software e Aplicações

Outra técnica de MTD é a alteração dinâmica do software e das aplicações, fazendo com que o código seja reconfigurado de forma imprevisível durante a execução. Isso pode envolver a modificação de funções ou a implementação de diferentes versões de uma aplicação em execução. Essa abordagem pode dificultar a realização de ataques como a injeção de código ou ataques de exploração de falhas, pois o comportamento do sistema está em constante mudança.

2.3 Arquiteturas Baseadas em Serviços

A arquitetura baseada em serviços é uma abordagem eficaz para a construção de sistemas escaláveis, flexíveis e resilientes. No contexto da cibersegurança, ela permite a modularização das funcionalidades de segurança e facilita a integração de diferentes componentes que trabalham de forma independente, mas coordenada. Abaixo estão algumas das abordagens e tecnologias utilizadas na implementação de arquiteturas baseadas em serviços.

- **Microserviços na Segurança**: são uma abordagem arquitetônica que se baseia na divisão de um sistema em componentes pequenos e independentes, cada um com uma responsabilidade específica. Essa abordagem oferece diversos benefícios para a segurança e escalabilidade dos sistemas. No âmbito do projeto, as comunicações entre os diferentes serviços foram feitas de forma eficiente utilizando JWT (JSON Web Tokens), que garantem a autenticação e a autorização seguras em um ambiente distribuído. Os JWTs permitem que o sistema autentique utilizadores e serviços sem a necessidade de manter um estado de sessão, o que é particularmente útil em arquiteturas de microserviços.
- **Arquitetura Serverless**: permite a execução de funções em resposta a eventos, sem a necessidade de gerenciar servidores ou infraestrutura. Serviços como AWS Lambda e Azure Functions oferecem escalabilidade automática e pagam apenas pelo uso real, otimizando os custos.
- **Modelos de Comunicação**: escolha de um modelo de comunicação eficiente e seguro entre os microserviços e outras partes do sistema é essencial para garantir a integridade e a confidencialidade dos dados em trânsito.

2.4 Tecnologias

Ferramenta	Descrição
Node.js	Plataforma de execução JavaScript no lado do servidor, que permite a criação de aplicações rápidas e escaláveis. É útil para a construção de microserviços, garantindo a comunicação segura e facilitando a implementação de funcionalidades em larga escala.
Flask	Foi utilizado como framework web para construir APIs RESTful, facilitando a criação de endpoints seguros e escaláveis para comunicação entre os microserviços.
Incus	Uma tecnologia de containerização que ajudou a gerenciar e isolar os microserviços, permitindo sua execução de forma eficiente e escalável.

Tabela 3: Tecnologias

3 Arquitetura de Serviços Utilizada

Exemplificando: Vamos imaginar que fazemos parte de uma equipa de Cibersegurança contratada para implementar técnicas de segurança num sistema bancário. Nesse sistema, existem dois tipos principais de serviços: o denominado "bank node", que armazena os registos das contas bancárias e permite que os utilizadores criem contas, depositem ou retirem dinheiro, e o "registry", que contém informações sobre os diversos nós existentes no sistema.

Como o sistema é utilizado por um grande número de utilizadores, surgiu a necessidade de distribuir a carga por vários nós. Neste caso, a empresa bancária é portuguesa e, atualmente, existem 4 nós localizados nas cidades de Covilhã, Porto, Lisboa e Coimbra. Esta distribuição visa aumentar a escalabilidade do sistema, permitindo-lhe suportar o volume de transações e pedidos esperados.

No entanto, os utilizadores do sistema podem deslocam-se pelo país e precisar de realizar transações em qualquer região. Por exemplo, um utilizador que tenha uma conta criada na Guarda, mas estude em Aveiro e, no verão, viaje para a Figueira da Foz e Évora irá aceder a nós diferentes em cada localidade referida, ou seja, para garantir que o sistema funcione corretamente, os nós responsáveis por cada localidade precisam de estar sempre em consenso e ser capazes de tolerar as possíveis falhas .

Além disso, para facilitar a verificação e o monitoramento do sistema, existe uma aplicação chamada "check nodes", que permite comparar os saldos das contas em diferentes nós, garantindo que as informações estejam consistentes e atualizadas em toda a rede.

Este cenário exige uma implementação robusta de segurança, além de técnicas avançadas de sincronização e alta disponibilidade, para garantir que as transações sejam realizadas com integridade, mesmo em face de falhas ou mudanças de localidade.

A arquitetura utilizada é apresentada como a Figura 1 descreve.

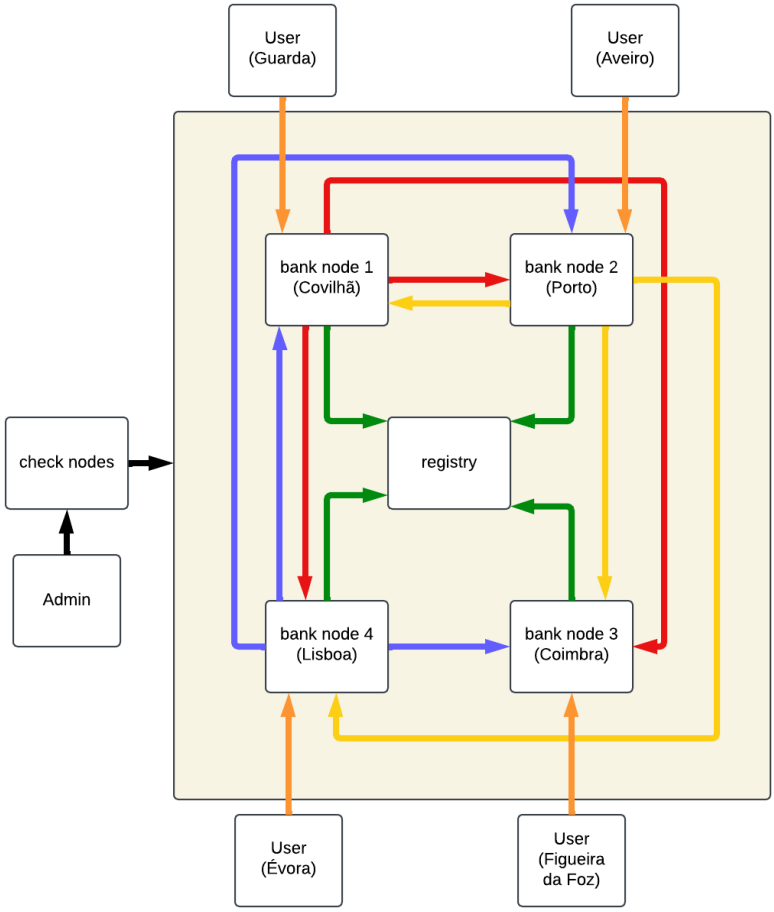

Figura 1: Arquitetura

3.1 registry

Este serviço contém informações sobre os nós disponíveis no sistema, de forma que cada nó tenha conhecimento dos outros nós em operação e possa comunicar com eles para estabelecer consenso nas transações. A primeira ação que um "bank node"realiza ao ser inicializado é notificar o registry sobre a sua existência e, em seguida, obter informações sobre os outros nós disponíveis na rede.

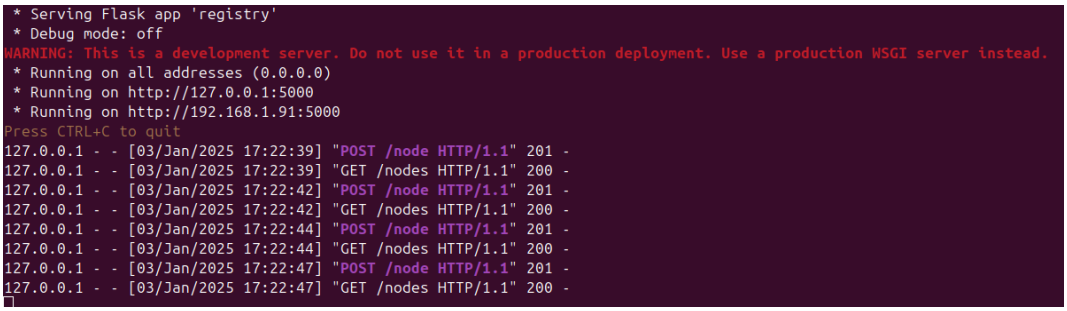

Figura 2: Registry com novos nós

Endpoints:

- **/nodes GET**: Retorna a lista de todos os nós atualmente disponíveis no sistema.
- **/node POST**: Adiciona um novo nó à lista de nós disponíveis.
- **/rm_node POST**: Remove um nó da lista de nós disponíveis.

3.2 bank node

Conforme mencionado anteriormente e ilustrado na Figura 1, após a inicialização, o bank node comunica-se com o registry, notificando-o sobre a sua existência. Em seguida, solicita a lista atualizada de nós e utiliza essa lista para notificar os outros nós sobre a sua disponibilidade através do endpoint **/newNode**. Os nós que recebem esta notificação adicionam o novo nó à sua lista interna. Após este processo de sincronização inicial, os utilizadores podem começar a realizar as várias transações disponíveis por meio dos seguintes endpoints:

- **/create_account POST**: Criar uma conta nova.
- **/deposit POST**: Depositar numa conta já existente
- **/withdraw POST**: Levantar dinheiro de uma conta já exitente

Para garantir o consenso entre os diversos nós, foi implementada uma comunicação inspirada no protocolo PBFT, como apresentado na Figura 3. No sistema implementado, o Leader é escolhido com base na localização do cliente, permitindo uma operação adaptada à proximidade do utilizador.

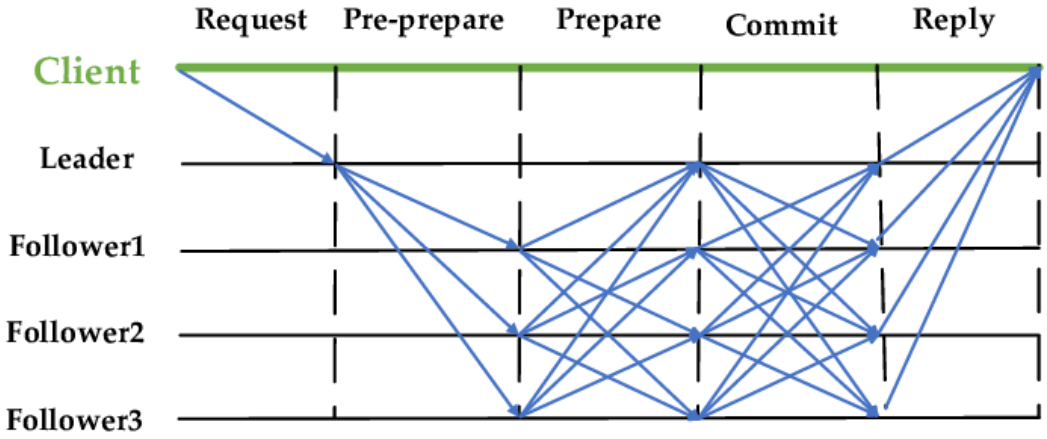

Figura 3: comunicação com PBFT

Para que a troca de mensagens necessária ao protocolo implementado seja possível, foram definidos os seguintes endpoints:

- **/preprepare POST**: Utilizado pelo Leader para enviar mensagens iniciais aos outros nós.
- **/prepare POST**: Todos os nós, exceto o Leader, fazem broadcast desta mensagem para confirmar a transação proposta.
- **/commit POST**: Todos os nós fazem broadcast desta mensagem para confirmar a execução da transação.

Além destes endpoints, é possível consultar as reputações de cada nó e obter informações sobre todas as contas. Estas operações são destinadas ao administrador através da aplicação "check nodes".

- **/accounts GET**: Retorna a lista completa de todas as contas.
- **/reputations GET**: Retorna as reputações atribuídas aos outros nós.

Exemplo de uma transação realizada no **bank node 1** do tipo: **create_account**, com o dono: **a** e um valor total de **1** euro sem nunhuma falha associada:

Neste cenário, o líder será o **bank node 1**, que, ao receber o pedido, irá criar um identificador único para a transação e gerar um hash correspondente. Este hash será includido na mensagem a enviar em broadcast para os outros nós através do endpoint **/preprepare**.

Ao receber as mensagens do tipo "preprepare", os nós calculam o hash da mensagem e comparam-no com o hash enviado pelo líder para garantir a integridade dos dados. Em seguida, a mensagem recebida é armazenada num dicionário associado ao identificador da transação, e o nó envia uma mensagem do tipo "prepare"para todos os outros nós, utilizando o endpoint **/prepare**.

À medida que os nós recebem as mensagens "prepare", o hash de cada uma é verificado, e as mensagens são adicionadas ao dicionário associado ao identificador da transação. O número de mensagens necessárias para avançar na próxima etapa é calculado através da fórmula: **n - 1 - f**, onde **n** representa o total de nós na rede e **f** o número de falhas toleradas, que é definido como **(n - 1) // 3**. Quando um nó "follower"recebe o número necessário de mensagens **(n - 1 - f)**, ele está apto a enviar o seu commit para todos os nós. Já o líder, por sua vez, necessita de receber **(n - f)** mensagens antes de enviar o seu commit, conforme representado na Figura 3. Antes do envio do commit, os dicionários associados às fases "preprepare"e "prepare"são consultados, e a mensagem cujo hash ocorre mais vezes será escolhida para envio, **/commit**.

Ao receber os commits, assim como nas etapas anteriores, os hashes das mensagens recebidas são verificados, e as mensagens são armazenadas num dicionário associado ao identificador da transação.

Quando um nó acumula um total de **(n - 1 - f)** mensagens de commit, os dicionários de "preprepare", "prepare"e "commit"são consultados. A mensagem com o hash mais repetido é considerada como a que contém a informação correta e será executada.

3.3 check nodes

Esta aplicação foi desenvolvida para efeitos de monitorização e teste do sistema e deve ser utilizada exclusivamente por um administrador ou desenvolvedor. A sua principal função é verificar o comportamento do sistema, incluindo o estado dos nós, a consistência dos dados e a execução das transações.

A interface da aplicação é apresentada na **Figura 1**:

```
==== Nodes Application ====
1. Check nodes data
2. Add account
3. Withdraw from account
4. Deposit in account
5. Check reputations
6. Exit
Choose an option (1-5):
```

Figura 4: Interface check nodes

Com esta aplicação, é possível realizar todas as transações disponíveis no sistema, bem como verificar o consenso entre os nós. Abaixo, estão detalhadas as funcionalidades através de exemplos práticos:

A funcionalidade de criação de conta está disponível para simular a abertura de novas contas no sistema.

```
Choose an option (1-5): 2
Insert in node: http://0.0.0.0:5001 ? (y/N)
Insert in node: http://0.0.0.0:5002 ? (y/N)
Insert in node: http://0.0.0.0:5003 ? (y/N)
Insert in node: http://0.0.0.0:5004 ? (y/N)y
     Owner:b
     Amount:1
     Result:  201
```

Figura 5: Adicionar um conta

É possível verificar se os dados foram sincronizados corretamente entre os nós.

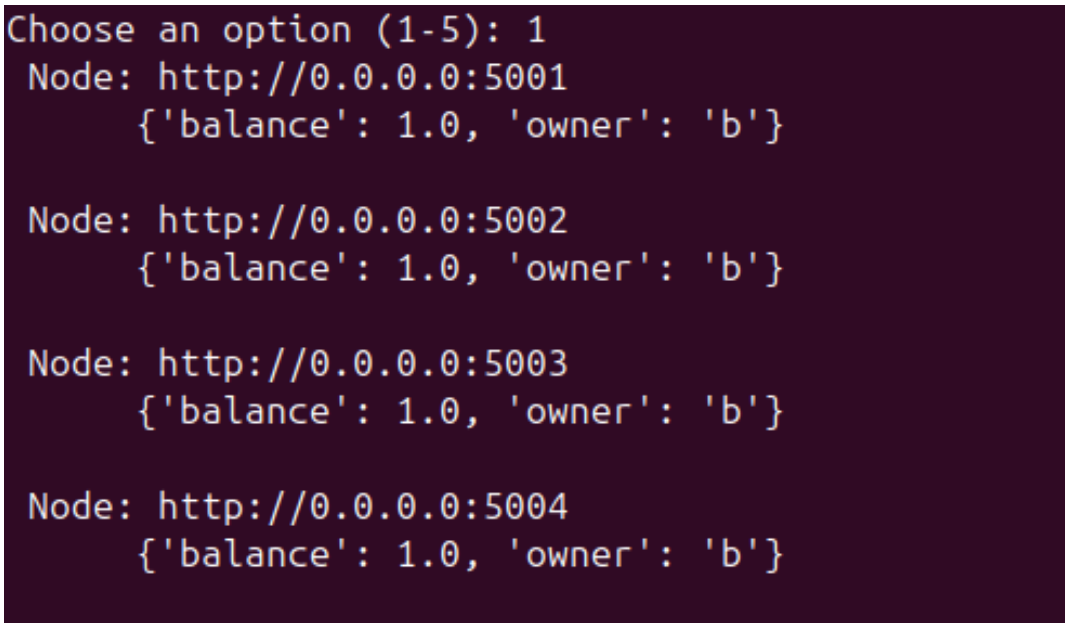

Figura 6: Verificar alteração

Permite simular o depósito de dinheiro numa conta existente.

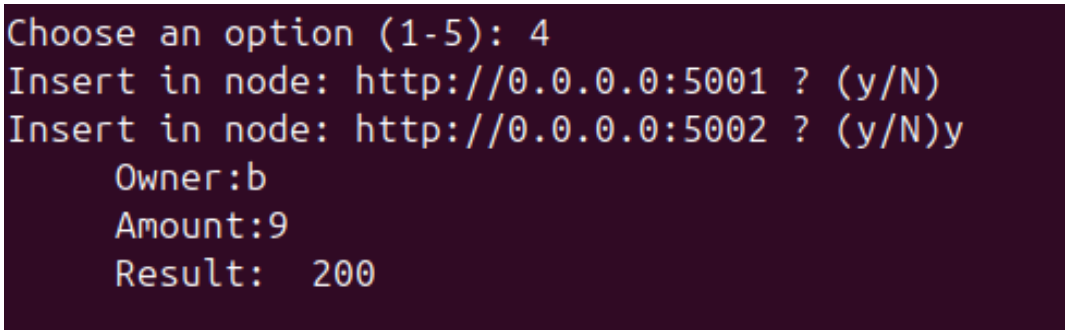

Figura 7: Depositar num conta

A aplicação verifica e confirma a alteração após o depósito.

```
Choose an option (1-5): 1
 Node: http://0.0.0.0:5001
      {'balance': 10.0, 'owner': 'b'}

 Node: http://0.0.0.0:5002
      {'balance': 10.0, 'owner': 'b'}

 Node: http://0.0.0.0:5003
      {'balance': 10.0, 'owner': 'b'}

 Node: http://0.0.0.0:5004
      {'balance': 10.0, 'owner': 'b'}
```

Figura 8: Verificar alteração

Simula o levantamento de dinheiro de uma conta já existente.

```
Choose an option (1-5): 3
Insert in node: http://0.0.0.0:5001 ? (y/N)
Insert in node: http://0.0.0.0:5002 ? (y/N)
Insert in node: http://0.0.0.0:5003 ? (y/N)y
     Owner:b
     Amount:5
     Result:  200
```

Figura 9: Levantar dinheiro de uma conta

Após o levantamento, a aplicação valida a operação e confirma a atualização dos dados.

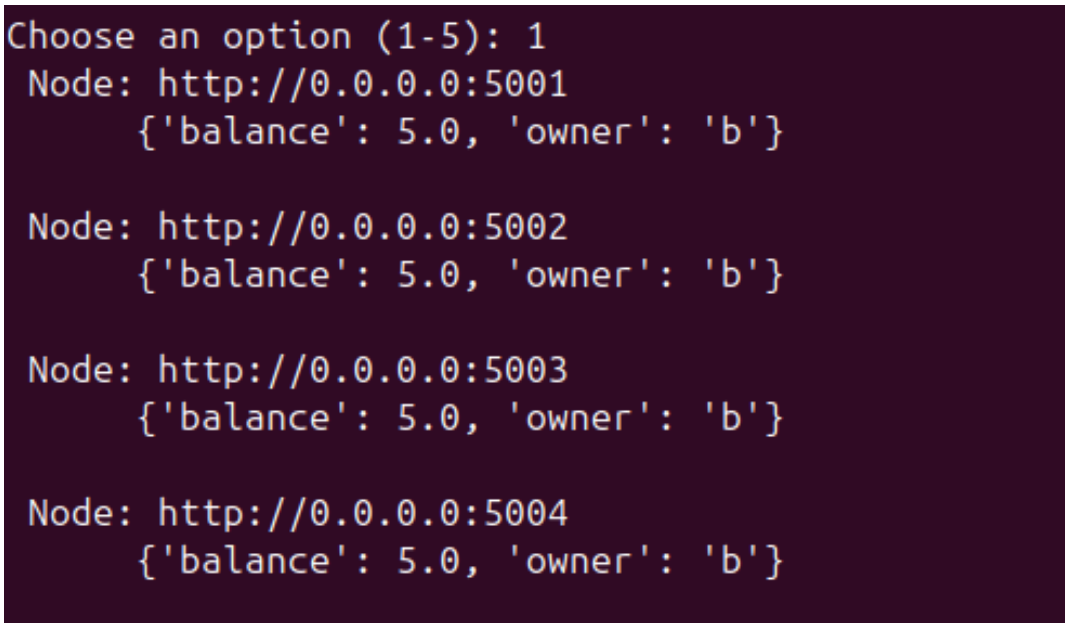

Figura 10: Verificar alteração

A aplicação também permite ao administrador consultar as reputações dos nós, auxiliando na análise do desempenho e confiabilidade do sistema. Uma vez que não ocorreu nenhuma falha as reputações mativeram se intactas.

Figura 11: Verificar reputações

NOTA: Para simplificar a implementação os nós foram inicializados localmente em portas distintas. No entanto, o sistema poderia ser facilmente containerizado, usando Incus.

4 Gestão de Risco e Mitigação

4.1 Árvores de ataque e defesa

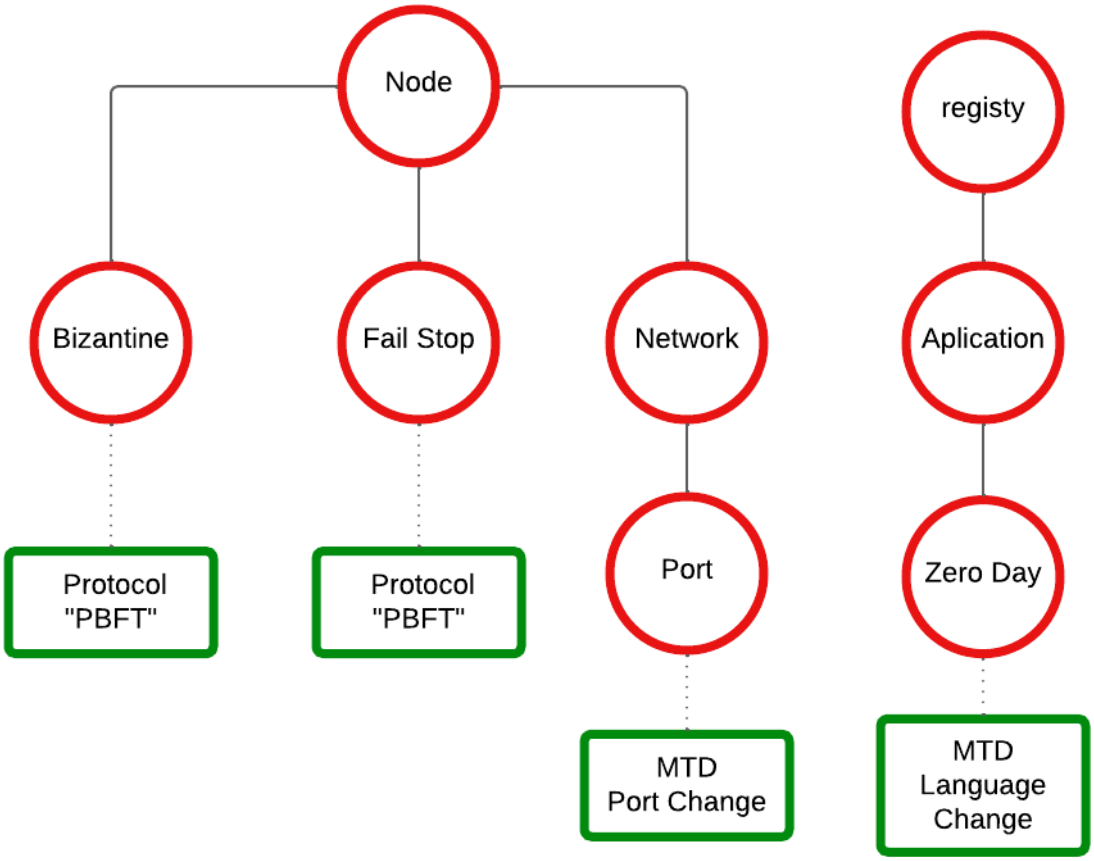

Figura 12: ADT do sistema

4.2 Superfícies de ataque

A superfície de ataque refere-se a todas as possíveis entradas e pontos de interação que podem ser exploradas por agentes maliciosos para comprometer um sistema. A identificação e redução da superfície de ataque é essencial para mitigar riscos e fortalecer a segurança. No sistema realizado, as principais superfícies de ataque são os serviços:

- **"bank node"**

  Os caminhos de ataque para o serviço "bank node"incluem falhas bizantinas e falhas completas (fail stop). No caso das falhas bizantinas, o ataque pode ocorrer quando os nós da rede se comportam de forma maliciosa ou inconsistente, comprometendo a integridade do sistema e o consenso da rede. Para mitigar este tipo de ataque, utiliza-se o protocolo baseando nas ideologias do PBFT (Practical Byzantine Fault Tolerance), que assegura a tolerância a falhas bizantinas ao alcançar consenso mesmo na presença de nós maliciosos. Além disso, as falhas completas (fail stop) ocorrem quando os nós deixam de operar de forma abrupta, prejudicando o funcionamento do sistema. Neste cenário, o protocolo implementado também é eficaz, garantindo a continuidade do sistema mesmo na eventualidade de falhas em alguns nós.

Outra superfície de ataque no "bank node"encontra-se ao nível da rede, particularmente na exploração de portas abertas. Os atacantes podem tentar comprometer a comunicação entre os nós explorando essas vulnerabilidades. Para mitigar este risco, utiliza-se uma estratégia de MTD (Moving Target Defense), através de mudanças dinâmicas de portas. Esta abordagem consiste em alterar regularmente as portas de comunicação, reduzindo a previsibilidade e tornando mais difícil para os atacantes identificarem e explorarem pontos de entrada na rede.

- **"registry"**

  Os principais caminhos de ataque para o serviço "registry"estão relacionados a vulnerabilidades em aplicações, incluindo ataques de tipo "Zero Day". Estas vulnerabilidades exploram falhas desconhecidas no software, permitindo que os atacantes comprometam o sistema antes que sejam implementadas medidas de correção. Para reduzir a superfície de ataque, recorre-se à defesa baseada em MTD (Moving Target Defense) com mudanças dinâmicas de linguagem. Esta estratégia altera continuamente o ambiente de execução, tornando mais difícil para os atacantes explorarem vulnerabilidades, uma vez que o sistema se mantém imprevisível e dinâmico.

4.3 Monitorização de sistemas em larga escala

Para monitorizar o sistema em larga escala, apenas foi desenvolvida a aplicação anteriormente referida, designada como "check nodes". Esta ferramenta permite que os administradores acedam e verifiquem as informações armazenadas em cada nó, incluindo dados relevantes como a reputação de cada um. Esta funcionalidade assegura uma supervisão contínua e eficaz, garantindo maior controlo e transparência no funcionamento do sistema.

5 Abordagem a ameaças

5.1 Falhas Fail Stop

- Explicação

  Estas falhas ocorrem quando um componente ou nó do sistema deixa de operar de forma súbita, interrompendo completamente a sua funcionalidade. Este tipo de falha pode resultar de diversos fatores, como erros de hardware, interrupções de energia, falhas de software ou outros eventos inesperados que impossibilitem o nó de continuar a desempenhar o seu papel na rede. Embora não envolvam comportamentos maliciosos, estas falhas podem comprometer a estabilidade e eficiência do sistema caso não sejam geridas de forma eficaz.

- Mitigação

  Para mitigar estas falhas, o protocolo foi concebido para suportar a perda de nós, assegurando que o funcionamento da rede não seja interrompido. O exemplo abaixo ilustra uma função de broadcast que adapta dinamicamente a lista de nós ativos, permitindo ao sistema contornar falhas de conexão:

Listing 1: Função de broadcast

```
        def send_request(node):
            delay = random.uniform(0, 1)
            try:
                time.sleep(delay)
                requests.post(node + endpoint, data=data)
            except Exception as e:
                print(f"Failed␣to␣contact␣node␣{node}:␣{e}")
                nodes.remove(node)
```

Durante o processo de broadcast, esta função é invocada. Caso ocorra algum problema de conexão com um nó, este será automaticamente removido da lista de nós ativos. Esta abordagem permite que o protocolo continue a sua execução de forma estável e sem interrupções, mesmo na presença de falhas em alguns nós da rede.

5.2 Falhas Bizantinas

- Explicação

  Estas ocorrem quando um componente ou nó de um sistema distribuído se comporta de forma arbitrária ou maliciosa, enviando informações incorretas, inconsistentes ou deliberadamente enganosas para outros nós da rede. Este tipo de falha é particularmente desafiador porque não se limita a falhas simples (como paragens abruptas, típicas das fail stop), mas envolve comportamentos que podem minar a confiança e o consenso dentro do sistema.

  Estas falhas podem ser causadas por ataques intencionais, como hackers a explorar vulnerabilidades do sistema, ou por erros internos, como corrupção de memória ou bugs críticos. Num cenário bizantino, o nó comprometido pode apresentar ações imprevisíveis, como reportar estados falsos ou tentar sabotar decisões coletivas.

- Mitigação

  Para mitigar falhas bizantinas, o protocolo de consenso tolerantes a falhas bizantinas, semelhante como o PBFT (Practical Byzantine Fault Tolerance) foi implementado. Estes protocolos asseguram que o sistema alcance consenso mesmo na presença de nós maliciosos, desde que o número de falhas não ultrapasse um limite específico, neste caso **f = (n - 1) // 3**, onde **n** é o número total de nós.

  No caso de existirem 7 nós onde 2 são bizantinos o protocolo, vai garantir que os todos nós mantenham a mesma informação.

  Além disto, um sistema de reputações também foi implementado, através da comparação dos hashes das mensagens recebidas, sempre que for detetada alguma alteração a reputação do nó que enviou a mensagens fraudulenta vai diminuir. Ao inicializar um nó, quando este recebe informação da existencia de outros nós partem todos com 100 pontos de reputação. Tal como o código em baixo demonstra, se alguma incoerencia for encontrada a reputação do nó que enviou diminui 25 pontos.

Listing 2: Redução de pontos

```
request_digest = data.pop("digest", None)
recv_node = data.pop("node", None)

serialized_data = json.dumps(data, sort_keys=True)
digest = hashlib.sha256(serialized_data.encode()).hexdigest()
data["digest"] = digest

if request_digest != digest:
    reputation[recv_node] -= 25 if reputation[recv_node] >= 25 else 0
```

Esta reputação é fundamental para a função de broadcast, onde as mensagens são enviadas apenas para os nós cuja reputação seja igual ou superior a 20. A lógica de filtragem e envio é implementada como segue:

Listing 3: Redução de pontos

```
nodes = [node for node in nodes if reputation[node] >= 20]
for node in nodes:
    thread = threading.Thread(target=send_request, args=(node,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

Porém, surge o problema de um líder bizantino. Se o líder for malicioso, irá enviar mensagens inconsistentes durante as fases de **preprepare** e **commit**. Os outros nós, ao detectarem essas

inconsistências, irão reduzir a reputação do líder. No entanto, essa redução de reputação não é suficiente para evitar a execução de uma transação fraudulenta.

Como contramedida, no final de cada commit, os nós executam uma função de verificação adicional. Esta função avalia as mensagens de preprepare armazenadas de transações anteriores. Se for detectado que o líder de uma transação possui uma reputação abaixo do limite permitido (inferior ou igual a 20), as mensagens relacionadas são revertidas. Esta medida é extrema, pois pode penalizar um nó que tenha se tornado bizantino apenas após realizar várias transações válidas. No entanto, é essencial para garantir a integridade do consenso, especialmente num contexto tão crítico como um sistema bancário.

Listing 4: Redução de pontos

```
        def fix_bizantine_changes(pre_messages):
            global preprepared_messages

            to_remove = []

            for id, messages in pre_messages.items():
                preprepare_msg = messages[0]
                leader_node = preprepare_msg["node"]

                if reputation[leader_node] <= 20:

                    reverse_execution(preprepare_msg)
                    to_remove.append(id)

            for id in to_remove:
                del preprepared_messages[id]
```

Esta abordagem permite que o sistema reverta automaticamente alterações indesejadas provocadas por líderes maliciosos. Contudo, a medida requer uma análise cuidadosa para evitar reverter transações legítimas desnecessariamente, preservando a confiança e a estabilidade do sistema.

5.3 MTD (Moving Target Defense)

O uso de técnicas de Moving Target Defense (MTD) é uma abordagem eficaz para lidar com ataques relacionados à previsibilidade e exploração estática de vulnerabilidades. Conforme ilustrado no ataque e defesa mostrado na imagem fornecida:

- Mudança Dinâmica de Portas (MTD Port Change) Esta técnica é utilizada para mitigar ataques que exploram vulnerabilidades de rede, como ataques direcionados a portas de serviços específicas. Alterando dinamicamente as portas utilizadas pelos nós, os atacantes enfrentam maior dificuldade em identificar e explorar serviços disponíveis, aumentando significativamente a segurança do sistema.

  No sistema implementado, após um nó ser inicializado é executada uma thread secundária responsável por monitorar e realizar a mudança de porta. Após 30 segundos, esta thread encerra o serviço ativo e inicializa um novo processo em uma porta aleatória, escolhida dentro do intervalo permitido (1024–65535).

  No entanto, como o sistema atual não está integrado com uma base de dados persistente, todos os dados armazenados no nó são perdidos no momento em que o serviço é encerrado e reiniciado. Isso representa uma limitação significativa, pois informações críticas ou estados do sistema não são preservados. Para resolver essa questão, seria ideal integrar uma base de dados persistente que permita o armazenamento e a recuperação de dados mesmo após o reinício do serviço.

Listing 5: Redução de pontos

```
        def change_port():
            global nodes

            time.sleep(30)

            requests.post(registry + "/rm_node", data={"url": f"http://{incus_ip}:{
                port}"})

            for node in nodes:
                requests.post(node + "/rmNode", data={"url": f"http://{incus_ip}:{port
                    }"})

            random_port = random.randint(1024, 65535)

            print(f"Changing␣port␣to␣{random_port}...")

            subprocess.Popen([sys.executable, 'good_pbft/bank_node.py', str(
                random_port)])

            sys.exit()

        change_port_thread = threading.Thread(target=change_port, daemon=True)
        change_port_thread.start()
```

- Mudança Dinâmica de Linguagem (MTD Language Change)

  No contexto de vulnerabilidades de aplicações, especialmente ataques Zero Day, a mudança dinâmica de linguagem ou do ambiente de execução impede que os atacantes explorem falhas específicas. Por exemplo, se uma aplicação é executada em Python, ela pode ser transferida para um ambiente de Node.js, tornando inválidas quaisquer explorações previamente preparadas.

  Esta abordagem cria um sistema mais dinâmico e resistente, dificultando a preparação de ataques direcionados. No caso do sistema implementado, o serviço mencionado anteriormente, Registry, foi desenvolvido tanto em Python, utilizando o Flask, quanto em JavaScript, utilizando o Node.js. Essa abordagem garante flexibilidade no desenvolvimento, no entanto, surge a necessidade de um mecanismo para que um administrador possa alternar entre os dois serviços de forma eficiente, sem interrupções significativas no funcionamento do sistema.

A estratégia MTD, como sugerida na árvore de defesa, oferece um nível adicional de proteção em cenários onde soluções tradicionais são insuficientes.

6 Avaliação das abordagens e escalabilidade da solução

Avaliação das Abordagens:

- **Solução e escalabilidade:**
  - Robustez e Consistência: A solução proposta utiliza um protocolo inspirado no PBFT para garantir a integridade dos dados e a consistência em um sistema distribuído. Essa abordagem assegura que, mesmo em cenários adversos com falhas bizantinas, o sistema seja capaz de operar sem comprometer a confiança nos resultados.
  - Desafios de Escalabilidade: Embora eficaz, o protocolo enfrenta limitações em termos de escalabilidade devido ao aumento do tráfego de mensagens em redes maiores. Estratégias como agrupamento de nós (clustering) e variações otimizadas do protocolo podem ser implementadas para reduzir a sobrecarga em redes amplas, permitindo que o sistema cresça sem perda significativa de desempenho.
- **Protocolo de comunicação:**
  - Estrutura Baseada em Consenso: O protocolo é estruturado em fases bem definidas (pre-prepare, prepare e commit), onde os nós participantes trocam mensagens para validar e alcançar consenso sobre cada operação. Essa abordagem minimiza a chance de inconsistências no sistema, garantindo que todos os nós compartilhem o mesmo estado.

  - Tolerância a Falhas Bizantinas: O protocolo garante que o sistema funcione de forma confiável mesmo quando até f nós apresentarem comportamento malicioso ou falhas arbitrárias. Isso é obtido com a condição de que o número total de nós seja no mínimo $3f + 1$, promovendo segurança e resiliência em redes distribuídas, também o sistema de reputação é bastante relevante para a tolerância a estas falhas.

- **Moving Target Defense:**

  - Rotação de Portas de Comunicação: O sistema adota a técnica de alterar dinamicamente as portas de comunicação utilizadas pelos serviços. Essa estratégia dificulta a exploração por parte de atacantes que dependem de informações estáticas sobre a infraestrutura do sistema, reduzindo a probabilidade de ataques bem-sucedidos, como varreduras de portas ou tentativas de intrusão direcionadas.
  - Modificação do Código do Serviço: Outra camada de defesa é implementada através de alterações periódicas no código do serviço, como mudanças nos padrões de resposta ou lógica interna. Essa abordagem adiciona uma imprevisibilidade adicional, desafiando adversários que tentem explorar vulnerabilidades conhecidas ou padrões previsíveis no comportamento do sistema. Essa técnica também reforça a resiliência do sistema em cenários onde um ataque prolongado poderia comprometer o serviço.

Quanto à escalabilidade, a solução tem potencial para ser escalável, uma vez que é uma arquitetura baseada em serviços que permitem integrar facilmente novos módulos. O uso de tecnologias como Flask e de protocolos de comunicação distribuido garante ao sistema a capacidade de se saber comportar com um aumento do número de serviços ou utilizadores.

7 Conclusão

Este projeto abordou os principais desafios associados à comunicação segura num sistema distribuido e prevenção de ameaças em sistemas de larga escala.

- **Limitações da Solução:**

  Uma das principais limitações da solução é a ausência de uma base de dados persistente em cada nó. Isso significa que, sempre que a porta do serviço é alterada ou o servidor é reiniciado, o nó perde toda a informação armazenada. Essa falta de persistência compromete a continuidade dos dados e pode levar a inconsistências ou necessidade de sincronização adicional com outros nós na rede. Para mitigar esse problema, seria essencial implementar um sistema de armazenamento persistente local, que permita que os dados sejam mantidos mesmo após reinicializações ou mudanças nas configurações.

  Outro ponto a ser considerado é o sistema de reputação, que, na solução atual, ainda não se encontra em um estado suficientemente robusto. Ele poderia ser aprimorado por meio de um estudo mais aprofundado sobre algoritmos de reputação que levem em conta fatores como confiabilidade histórica, frequência de falhas e comportamento malicioso detectado. O uso de abordagens baseadas em machine learning ou modelos estatísticos poderia ajudar a atribuir reputações mais precisas aos nós, permitindo decisões mais informadas durante o processo de consenso.

  Além disso, o protocolo de comunicação utilizado poderia ser refinado com melhorias específicas e a realização de testes mais abrangentes. Alterações no protocolo, como otimizações no número de mensagens trocadas ou maior eficiência no processo de validação, poderiam reduzir a sobrecarga de comunicação. Testes rigorosos, incluindo cenários adversos e simulações de redes maiores, também são cruciais para validar a robustez do sistema e identificar vulnerabilidades potenciais antes de seu uso em produção. Essas melhorias não apenas resolveriam as limitações atuais, mas também fortaleceriam a solução para escalabilidade e resiliência a longo prazo.

- **Trabalho Futuro:**

  - **Persistência de Dados:** Desenvolver uma solução para garantir a persistência dos dados em cada nó do sistema, mesmo durante alterações de porta ou reinicializações. A implementação de bases de dados locais leves, como SQLite ou LevelDB, pode ser uma solução prática para assegurar a continuidade e consistência dos dados.
  - **Otimização de Sistema Operacional:** Investigar a personalização do sistema operacional ou configurações específicas para melhorar o desempenho do protocolo. Por exemplo, ajustar parâmetros de rede e priorizar conexões de baixa latência pode ajudar a otimizar a troca de mensagens em redes distribuídas.
  - **Novas Técnicas de MTD:** Ampliar a aplicação de Moving Target Defense além da rotação de portas e mudanças no código, como a utilização de balanceamento dinâmico de carga, modificação periódica da topologia da rede, e randomização de atributos de comunicação para dificultar ainda mais ataques direcionados.
  - **Aprimoramento do Sistema de Reputação:** Integrar técnicas avançadas, como modelos probabilísticos ou machine learning, para criar um sistema de reputação mais confiável e adaptativo. Esse aprimoramento permitirá identificar nós maliciosos com maior precisão e fortelacer a resiliência do protocolo.
  - **Testes Extensivos no Protocolo:** Realizar testes mais abrangentes em ambientes simulados para avaliar a resiliência e escalabilidade do protocolo. Simulações de falhas bizantinas, aumento do número de nós e cenários adversos ajudarão a identificar melhorias e validar o sistema para ambientes reais.
  - **Exploração de Arquiteturas Distribuídas:** Avaliar o uso de plataformas modernas, como Kubernetes, para gerenciar e orquestrar os nós da rede. Isso pode trazer benefícios como maior escalabilidade, balanceamento de carga automático e recuperação de falhas, além de simplificar a administração do sistema.

8 Recursos

Link do Onedrive com os videos
    Repositorio com os serviços