Consenso Bizantino Baseado em uma Camada de Rede com Ordenação de Mensagens Tolerante a Intrusões

Gabriel Faustino Lima da Rocha[1], **Eduardo A. P. Alchieri**[1], **Giovanni Venâncio**[2], **Vinicius Fulber-Garcia**[2], **Elias P. Duarte Jr.**[2]

[1]Universidade de Brasília (UnB)
Brasília - Distrito Federal - Brasil

[2]Universidade Federal do Paraná (UFPR)
Curitiba - Paraná - Brasil

Abstract. *Recent work proposes consensus protocols that divide the tasks of ensuring agreement and termination properties between the network and application layers, respectively. The network layer must deliver ordered messages to ensure agreement. To achieve this, these solutions use a sequencer. However, tolerating a malicious sequencer brings new challenges, since it can assign the same sequence number to different messages and send them to different replicas. To circumvent this problem, NeoBFT employs an additional communication step between replicas. Although this approach mitigates the problem, it affects system performance by requiring additional synchronization between replicas before requests are executed. This work proposes NsoBFT (Network Secure Ordered BFT), a consensus protocol that uses a network layer with a secure message ordering service implemented by the secure component USIG (Unique Sequential Identifier Generator). Thus, it is not necessary to perform additional synchronizations among the replicas before executing requests. Experimental results compare NsoBFT with related work, and show that this feature allows NsoBFT to surpass the performance of NeoBFT.*

Resumo. *Trabalhos recentes propuseram protocolos de consenso que dividem as tarefas de garantir as propriedades de acordo e de terminação entre as camadas de rede e de aplicação, respectivamente. A camada de rede precisa entregar as mensagens já ordenadas para garantir acordo. Para tanto, essas soluções normalmente utilizam um sequenciador. No entanto, tolerar um sequenciador malicioso traz novos desafios, uma vez que um mesmo número de sequência pode ser atribuído para diferentes mensagens e enviadas para diferentes réplicas. Para contornar tais problemas, o NeoBFT emprega um passo adicional de comunicação entre as réplicas. Apesar de lidar com o problema, essa abordagem impacta o desempenho do sistema, uma vez que sincronizações adicionais são necessárias nas réplicas antes da execução das requisições. Este trabalho propõe o NsoBFT (Network secure ordered BFT), um protocolo de consenso que utiliza uma camada de rede com um serviço seguro de ordenação de mensagens implementado através do componente seguro USIG (Unique Sequential Identifier Generator). Desta forma, não é necessário executar sincronizações adicionais nas réplicas antes de executar as requisições. Resultados experimentais comparam as propostas similares encontradas na literatura e mostram que esta característica faz com que o NsoBFT tenha um desempenho superior ao NeoBFT.*

1. Introdução

O consenso [Lamport et al. 1982] e, de modo geral, os algoritmos de acordo formam a base para a solução de muitos problemas relacionados ao desenvolvimento de sistemas distribuídos confiáveis. Por exemplo, resolver o consenso é fundamental para a implementação da replicação de máquina de estados [Schneider 1990], uma abordagem abrangente e amplamente utilizada na construção de sistemas tolerantes a falhas. Além disso, recentemente, o problema do consenso ganhou grande atenção com o surgimento e expansão das redes *blockchain* [Vukolić 2015, Liu and Yu 2024].

Um protocolo de consenso clássico é o Paxos [Lamport 1998], que, do ponto de vista de tolerância a falhas, lida apenas com falhas por parada (*crash*). Além do Paxos, o PBFT (*Practical Byzantine Fault Tolerance*) [Castro and Liskov 1999, Castro and Liskov 2002] foi a primeira solução a considerar aspectos práticos para a implementação do consenso tolerante a falhas bizantinas. Devido a sua importância, diversos trabalhos propuseram variações aos algoritmos do Paxos e do PBFT [Singh et al. 2022], assim como foram utilizados em diferentes cenários e aplicações [Zou et al. 2024, Li et al. 2024, Venâncio et al. 2024].

A complexidade das soluções de consenso está diretamente relacionada a aspectos fundamentais do modelo de sistema, como o tipo de canais de comunicação e o modelo de falhas suportado. As propriedades do consenso podem ser classificadas em duas categorias: as que visam garantir o acordo (*safety*) e as que asseguram a terminação (*liveness*). Recentemente, alguns estudos demonstraram que dividir essas responsabilidades entre as camadas de rede e de aplicação aumenta significativamente o desempenho do sistema. Nesse contexto, o NOPaxos [Li et al. 2016] emprega uma camada de rede que ordena as mensagens por meio de um sequenciador, responsável por atribuir números de sequência às requisições (ou mensagens) dos clientes. Já na camada de aplicação, apenas mensagens perdidas são recuperadas. Esse protocolo tolera apenas falhas *crash*, mas apresenta uma latência próxima à latência da rede, introduzindo uma sobrecarga mínima no sistema.

Outro protocolo que adota essa divisão de tarefas é o NeoBFT [Sun et al. 2023], tolerando falhas bizantinas por meio do uso de assinaturas digitais nos números de sequência atribuídos pelo sequenciador. Assim, com uma camada de rede que fornece um serviço de ordenação autenticado, as réplicas podem recuperar mensagens perdidas. No entanto, esse protocolo requer um passo adicional de comunicação entre as réplicas antes da execução de uma requisição, uma vez que o sequenciador pode atribuir números de sequência diferentes para a mesma mensagem. Nesse caso, a sobrecarga no sistema aumenta tanto pelo uso de assinaturas digitais quanto pela necessidade de coordenação adicional entre as réplicas na camada de aplicação.

Visando contornar os problemas relacionados com o NeoBFT, além de melhorar o desempenho deste protocolo, este trabalho propõe uma solução que também tolera falhas bizantinas, mas impede que um sequenciador malicioso atribua números de sequência diferentes para uma mesma mensagem. Para isso, a solução proposta emprega o componente seguro USIG (Unique Sequential Identifier Generator) [Veronese et al. 2013] no sequenciador – um gerador de identificadores sequenciais únicos. A solução proposta é chamada de NsoBFT (*Network secure ordered BFT*) e não necessita de coordenação adicional nas réplicas, *i.e.*, demanda um passo a menos de comunicação quando comparado com o NeoBFT. A implementação do USIG na camada de rede pode seguir

Característica	Paxos	PBFT	NOPaxos	NeoBFT	**NsoBFT**	**MinNsoBFT**	**MinZyzNsoBFT**
Modelo de falhas	*crash*	bizantina	*crash*	bizantina	**bizantina**	**bizantina**	**bizantina**
Número de réplicas[1]	*2f+1*	*3f+1*	*2f+1*	*3f+1*	***3f+1***	***2f+1***	***2f+1***
Passos de comunicação	4	5	2	3	**2**	**3**	**2**
Complex. de comunicação[2]	O(n²)	O(n²)	O(n)	O(n²)	**O(n)**	**O(n²)**	**O(n)**
Componente seguro	-	-	-	-	**USIG**	**USIG**	**USIG**

Tabela 1. Comparação dos protocolos de consenso. Nos protocolos que ordenam as requisições na camada de rede (NoPaxos, NeoBFT, NsoBFT e suas variantes), o sequenciador "intercepta" a requisição nesta camada para adicionar a ordem, e com isso economizam um passo de comunicação.
Notas:
[1] *f* representa o número de falhas toleradas.
[2] *n* representa a quantidade de réplicas no sistema.

implementações já propostas, como a do próprio NeoBFT [Sun et al. 2023], que utilizam um co-processador acoplado ao *switch* para executar operações mais complexas. Neste caso, o co-processador precisaria ser seguro, porém, ainda existem outras técnicas para a implementação deste componente, como o emprego de virtualização [Veronese et al. 2013]. Ressalta-se ainda que, assim como o NeoBFT e os protocolos clássicos de consenso bizantino como o PBFT, o NsoBFT necessita de $3f + 1$ réplicas para tolerar até f réplicas maliciosas, além do sequenciador.

Este trabalho também apresenta e discute duas variantes do NsoBFT que demandam apenas $2f + 1$ réplicas. A primeira variante é chamada de MinNsoBFT e emprega um passo adicional de comunicação, enquanto a segunda, chamada de MinZyzNsoBFT, não necessita de passos adicionais. No último caso, no entanto, o cliente finaliza uma operação apenas quando recebe a mesma resposta de todas as réplicas, executando ações adicionais para sincronizá-las caso isso não ocorra em um período pré-determinado. A Tabela 1 compara as soluções discutidas para o problema do consenso em termos de falhas suportadas, número de réplicas, passos de comunicação e complexidade de comunicação demandados no sistema, além da utilização de componentes seguros.

Além de apresentar e discutir esses protocolos, este trabalho realiza uma avaliação experimental dos protocolos que dividem as responsabilidades de ordenação e terminação entre as camadas de rede e aplicação (NOPaxos, NeoBFT e NsoBFT). De modo geral, o NOPaxos apresenta um desempenho superior, mas tolera apenas falhas *crash*, enquanto o NsoBFT supera o desempenho do NeoBFT por demandar um passo a menos de comunicação.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica e se aprofunda nos trabalhos relacionados. A Seção 3 detalha o protocolo NsoBFT e suas variantes. Uma avaliação experimental é apresentada na Seção 4. Finalmente, a Seção 5 conclui o trabalho.

2. Fundamentação Teórica e Trabalhos Relacionados

Esta seção inicia com a discussão das propriedades que um protocolo de consenso deve garantir, para então apresentar alguns protocolos de consenso, enfatizando as soluções que distribuem as responsabilidades, atendendo a algumas dessas propriedades na camada de rede e outras na camada de aplicação.

2.1. O Problema do Consenso

Em um sistema distribuído composto por vários processos independentes, o *problema do consenso* consiste em garantir que todos os processos corretos decidam sobre um mesmo valor, previamente proposto por algum dos processos do sistema. Formalmente, esse problema é definido em termos de duas primitivas [Hadzilacos and Toueg 1994]:

- `propose(G,v)`: executada para propor o valor v ao conjunto de processos G.
- `decide(v)`: executada pelo protocolo de consenso para notificar ao(s) interessado(s) (geralmente alguma aplicação) que v é o valor decidido.

Para que essas primitivas satisfaçam as propriedades de correção (*safety*) e vivacidade (*liveness*), elas devem satisfazer as seguintes propriedades:

- **Acordo**: Se um processo correto decide v, então todos os processos corretos terminam por decidir v.
- **Validade**: Um processo correto decide v somente se v foi previamente proposto por algum processo.
- **Terminação**: Todos os processos corretos terminam por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor. A validade relaciona o valor decidido com os valores propostos, e sua alteração dá origem a outros tipos de consensos. As propriedades de acordo e validade definem os requisitos de correção (*safety*) do consenso, enquanto a propriedade de terminação define o requisito de vivacidade (*liveness*) deste problema.

Assim, o problema do consenso é equivalente ao problema de difusão atômica. Em um trabalho relacionado [Venâncio et al. 2022] é apresentada uma solução para que a própria rede ofereça um serviço de difusão atômica baseado em Virtualização de Funções de Rede [Venâncio et al. 2022]. A difusão atômica, por sua vez, é a base para a implementação de replicação de máquina de estados [Schneider 1990]. De forma resumida, uma sequência de instâncias de consenso é executada, e o valor decidido na instância i representa a i-ésima requisição a ser executada pelo sistema replicado.

Um dos trabalhos mais importantes envolvendo o problema do consenso é o de [Fischer et al. 1985], que prova a impossibilidade de resolver este problema de forma determinística em um sistema distribuído completamente assíncrono onde pelo menos um processo pode falhar. Em vista disso, a maioria das propostas na literatura considera algum comportamento temporal do sistema.

2.2. Paxos e Variantes

Muitos sistemas replicados com guarantias de consistência utilizam o protocolo de consenso Paxos [Lamport 1998] para ordenar requisições. Esse protocolo tolera apenas falhas *crash* e é executado em duas fases: uma fase de preparação e outra que efetivamente leva a uma decisão, demandando dois passos de comunicação em cada fase. O PBFT [Castro and Liskov 1999, Castro and Liskov 2002] foi o primeiro protocolo tolerante a falhas bizantinas, também dito tolerante a intrusões, sendo executado em três fases. Similar ao Paxos, o PBFT utiliza uma combinação de réplicas primárias e de *backup* para ordenar requisições: o primário (ou líder) atribui números de sequência às requisições (faz uma proposta), enquanto os *backups* verificam esses números para garantir consistência,

além de monitorar o primário para detectar quando ele sofre uma parada ou se comporta de forma maliciosa.

Vários protocolos aprimoram aspectos do Paxos/PBFT. Por exemplo, o MinBFT [Veronese et al. 2013] utiliza componentes seguros para reduzir tanto a quantidade de réplicas quanto os passos de comunicação. Outros protocolos foram propostos para suportar a coexistência de múltiplas réplicas [Saramago et al. 2018] ou considerar sistemas nos quais as réplicas inicialmente não se conhecem [Alchieri et al. 2018]. Todos esses protocolos ordenam as requisições por meio da troca de mensagens na camada de aplicação. A seguir, são apresentadas duas soluções que se assemelham à nossa proposta, pois ordenam as requisições na camada de rede.

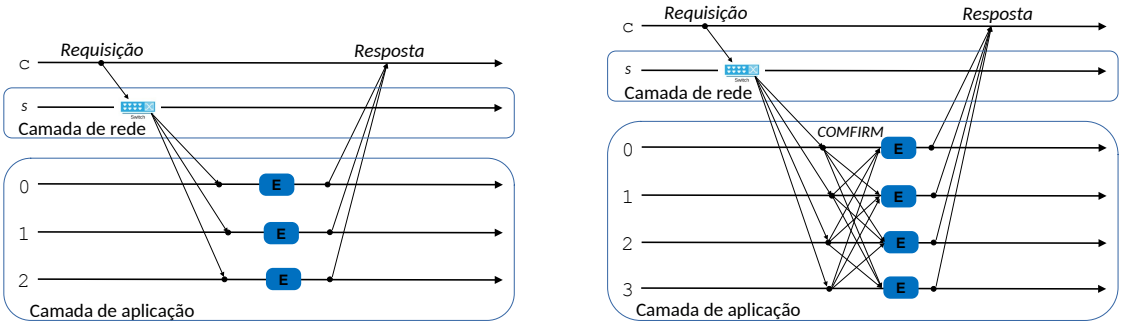
2.2.1. NOPaxos e NeoBFT

O NOPaxos (Network-Ordered Paxos) [Li et al. 2016] foi o primeiro protocolo a demonstrar que uma nova divisão de responsabilidades entre as camadas de rede e aplicação pode aumentar o desempenho dos sistemas replicados. Nessa abordagem, a camada de rede fornece um serviço de *multicast* não confiável ordenado, *i.e.*, todas as réplicas recebem e processam as requisições na mesma ordem, mas requisições podem ser perdidas. Nesse caso, requisições perdidas são recuperadas na camada de aplicação. Esse modelo é suficientemente flexível e robusto para ser implementado de forma eficiente e reduzir significativamente os custos de um protocolo de replicação.

De forma resumida, o NOPaxos utiliza um sequenciador na camada de rede (*e.g.,* implementado em um *switch*) que atribui números de sequência às requisições, que então são encaminhadas para as réplicas. As réplicas entregam e executam as requisições seguindo esses números (ordem definida na camada de rede) e tentam recuperar das outras réplicas quaisquer requisições com um número de ordem perdido. Em uma execução normal, as requisições são executadas em apenas um *round-trip* (Figura 1(a)): um cliente difunde uma requisição r para as réplicas através do sequenciador; as réplicas adicionam r em um *log*, e uma réplica líder também executa a operação; todas as réplicas respondem ao cliente com a posição p em que r foi adicionada no *log*, enquanto que o líder também envia a resposta; o cliente aguarda por $f + 1$ respostas com o mesmo p, incluindo a resposta do líder, para então obter a resposta final. As réplicas também monitoram e realizam a troca do sequenciador e/ou da réplica líder caso ocorram falhas. Como em outros protocolos que toleram apenas *crashes*, são necessárias pelo menos $2f + 1$ réplicas para tolerar até f réplicas faltosas. Finalmente, como apenas a réplica líder executa as requisições, não é necessário realizar *rollback* do estado das réplicas. Por outro lado, caso uma réplica líder não faltosa seja trocada devido a falsas suspeitas, as requisições por ela executadas podem não refletir no estado da nova réplica líder. Nesse caso, a antiga réplica líder atualiza seu estado a partir da nova líder, ao invés de realizar *rollback*.

Buscando tolerar falhas maliciosas, o NeoBFT [Sun et al. 2023] segue uma abordagem semelhante ao NOPaxos. No entanto, o sequenciador assina a requisição com o número de sequência atribuído, resultando em um serviço autenticado de ordenação na camada de rede. Essa abordagem permite que uma réplica recupere, a partir das outras réplicas, uma requisição perdida, bastando verificar se a assinatura correspondente é válida. Como um sequenciador malicioso pode atribuir o mesmo número de sequência a

diferentes requisições, é necessário um passo de comunicação adicional neste protocolo (Figura 1(b)) para verificar se pelo menos $2f+1$ réplicas receberam a mesma informação. Quando isso ocorre, a requisição é executada pelas réplicas, e a resposta é enviada ao cliente, que aguarda por $2f + 1$ respostas idênticas para obter a resposta final.



(a) NOPaxos. (b) NeoBFT.

Figura 1. Execução normal dos protocolos NOPaxos e NeoBFT.

Com isso, o NeoBFT requer $3f + 1$ réplicas para tolerar até f falhas maliciosas. Além disso, em alguns casos, é necessário ordenar uma "requisição" de *gap* para garantir o progresso do sistema. Quando isso ocorre, ou devido à troca de um sequenciador, pode ser necessário que alguma réplica realize *rollback* de seu estado. Finalmente, foram propostas algumas alternativas para a autenticação na camada de rede [Sun et al. 2023], como acoplar ao *switch* um co-processador capaz de gerar assinaturas ou utilizar um vetor de HMAC implementado diretamente no *switch*.

3. O Protocolo de Consenso NsoBFT

Neste trabalho, é proposto o NsoBFT (*Network secure ordered* BFT), um protocolo de consenso que, assim como o NOPaxos e o NeoBFT, divide as responsabilidades entre as camadas de rede e aplicação. Contudo, o NsoBFT utiliza o componente seguro USIG [Veronese et al. 2013] para gerar números de sequência de forma segura, *i.e.*, um sequenciador malicioso não é capaz de atribuir o mesmo número de sequência a diferentes mensagens. Como as requisições são ordenadas de forma segura, o NsoBFT elimina a necessidade do passo adicional de comunicação presente no NeoBFT e apresenta uma complexidade de troca de mensagens equivalente à do NOPaxos, mas com suporte a um modelo de falhas mais abrangente.

3.1. Modelo de Sistema

Assume-se um sistema distribuído composto por processos interconectados. Há um conjunto ilimitado de processos cliente e um conjunto de n processos servidor (réplicas). Considera-se que existe pelo menos um sequenciador correto instalado na camada de rede (*e.g.*, em um *switch*) que utiliza o USIG para atribuir números de sequência às requisições dos clientes. Assume-se o modelo de falhas bizantinas, *i.e.*, processos podem se comportar de forma maliciosa e não seguir as especificações dos protocolos. Um processo é *correto* se não falha, ou *faltoso* caso contrário. Existem, no máximo, f réplicas faltosas, de $n = 3f + 1$ réplicas. Os processos possuem identificadores únicos, sendo impossível obter identificadores adicionais para lançar um ataque Sybil [Douceur 2002].

Como em outros trabalhos que resolvem o consenso de forma determinística, consideramos um sistema parcialmente síncrono para garantir a terminação [Dwork et al. 1988, Bravo et al. 2022], enquanto as propriedades de segurança são sempre satisfeitas no sistema. Neste modelo, os processos e a rede podem operar de forma assíncrona até um ponto desconhecido de estabilidade de tempo (*Global Stabilization Time*) GST, a partir do qual o sistema passa a operar de forma síncrona.

Os processos se comunicam por envio e recebimento de mensagens, e a camada de rede pode perder, duplicar ou reordenar as mensagens. No entanto, mensagens não podem ser indetectavelmente corrompidas (canais autenticados), e, se repetidamente reenviadas de um processo correto para outro, serão entregues em algum momento (canal justo). Os clientes fazem *multicast* de suas requisições para as réplicas, enquanto as réplicas enviam as respostas por meio de um canal ponto a ponto com cada cliente e também se comunicam entre si quando necessário, via um canal ponto a ponto.

3.2. USIG: Gerador de Identificadores Sequenciais Únicos

O USIG é um serviço seguro usado para atribuir um identificador único para uma mensagem, além de assiná-la. Os identificadores atribuídos às mensagens são: *(i)* únicos (um mesmo identificador nunca é atribuído a duas ou mais mensagens); *(ii)* monotônicos (o identificador atribuído a uma mensagem nunca é menor do que o anterior) e; *(iii)* sequenciais (o identificador atribuído a uma mensagem sempre é o sucessor do anterior). Uma instância do USIG deve ser criada em cada componente do sistema que precise gerar estes identificadores. A interface definida para acessar este serviço é a seguinte [Veronese et al. 2013]:

- `createUI(m)`: retorna um certificado `UI` que contém um identificador único (`UI.id`) e a prova (`UI.proof`) de que este identificador foi criado por este componente e atribuído a uma mensagem `m`. O identificador é basicamente um contador monotonicamente crescente que é incrementado a cada chamada desta função. Já a prova envolve criptografia e pode ser baseada em criptografia assimétrica, *i.e.*, uma assinatura digital. A chave privada usada para gerar esta assinatura é armazenada de forma segura dentro deste componente.
- `verifyUI(PK,UI,m)`: verifica se o identificador único `UI` é válido para uma mensagem `m`. Esta função recebe como parâmetro a chave pública `PK`, associada à respectiva instância do USIG, para verificar se a assinatura contida em `UI.proof` é válida para `UI.id` e `m`.

Com o uso deste componente, um sequenciador não consegue enviar duas mensagens diferentes para processos distintos com o mesmo identificador. Assim, basta que cada processo mantenha armazenado o identificador da última mensagem recebida de um sequenciador para saber qual é o próximo identificador esperado, e assim reduzem-se as ações de um sequenciador malicioso: ou envia a mesma mensagem (que pode conter qualquer valor) para todos os processos ou não envia nada.

Este componente pode ser implementado de duas formas [Veronese et al. 2013]: é possível usar apenas *hashes* ou empregar assinaturas digitais. Neste trabalho, adota-se a solução que emprega assinaturas digitais, de forma que a verificação pode ser realizada fora do componente seguro, bastando a chave pública correspondente. Além disso, diferentes níveis de isolamento podem ser empregados, utilizando máquinas virtuais ou até mesmo módulos seguros de hardware.

3.3. O Protocolo NsoBFT

O protocolo NsoBFT se beneficia de uma camada de rede com ordenação segura, viabilizada pelo uso do serviço USIG. A seguir, são apresentados os protocolos para uma execução normal e para a troca de visão do NsoBFT.

Execução normal. Uma execução normal segue o padrão de mensagens apresentado na Figura 2. Note que o serviço USIG é instanciado apenas no sequenciador.

- O protocolo inicia com o cliente realizando um *multicast* de uma requisição r, que deve passar pelo sequenciador antes de chegar às réplicas. Por exemplo, o sequenciador pode ser instalado em um *switch* de uma camada superior que conecta todas as réplicas.
- O sequenciador utiliza o USIG para adicionar um contador em r, executando `createUI(r)` para gerar e incluir na mensagem o contador `UI`.
- As réplicas recebem a requisição r juntamente com o `UI` e verificam se o contador é válido utilizando a função `verifyUI(PK,UI,r)`, onde `PK` é a chave pública do serviço conhecida por todos. Caso seja válido, também verificam se `UI.id` é o próximo número da sequência a ser entregue. Se ambas as condições forem atendidas, elas executam r, incluem r em um *log*, e enviam a resposta ao cliente.
- O cliente aguarda por $2f + 1$ respostas iguais para completar a operação.

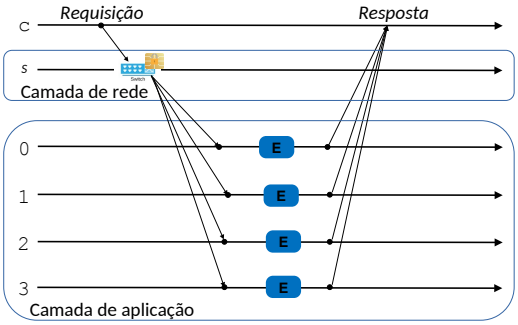


Figura 2. Execução normal do protocolo NsoBFT.

No protocolo descrito, caso uma réplica não receba uma determinada requisição, *i.e.*, o número de sequência recebido em `UI.id` resulta em uma lacuna na ordem de entrega. Assim, a réplica deve recuperar a requisição faltante de outras réplicas para prosseguir com a execução. Note que as réplicas mantêm as requisições armazenadas no *log*. Como o identificador `UI` é único para cada requisição, nenhuma ação maliciosa pode ser realizada durante este procedimento.

Troca de visão. Quando o sequenciador é suspeito de estar faltoso, é necessário escolher um novo sequenciador. Para isso, deve-se determinar até qual ponto no *log* as requisições devem ser preservadas. O protocolo proposto para a troca de visão assegura que todas as requisições que foram respondidas aos clientes por, no mínimo, $2f + 1$ réplicas, *i.e.*, que foram concluídas, sejam mantidas no *log*.

- Quando uma réplica suspeita que o sequenciador está faltoso, ela envia uma mensagem de `VIEW-CHANGE` para as outras réplicas.

- Quando uma réplica recebe uma mensagem de `VIEW-CHANGE`, ela retransmite essa mensagem para as demais réplicas.
- Quando uma réplica recebe $f + 1$ mensagens de `VIEW-CHANGE`, ela também envia uma mensagem de `VIEW-CHANGE`. Além disso, como é garantido que todas as réplicas corretas receberão essas mensagens, a réplica inicia um protocolo de consenso para determinar até qual ponto do *log* deve ser mantido, definindo, consequentemente, o que deve ser descartado:
  - Uma réplica líder propõe o seu *log*.
  - As outras réplicas só aceitam uma proposta caso o seu *log* não esteja à frente do *log* proposto. Para isso, o protocolo de consenso utilizado deve assumir um predicado definido para aceitar propostas [Vassantlal et al. 2022]. Isso garante que as requisições executadas e respondidas por pelo menos $2f + 1$ réplicas permanecerão no *log*.
  - Caso a proposta não seja aceita por pelo menos $2f + 1$ réplicas, um novo líder é escolhido e o processo se repete.
- Ao final da execução do consenso, o mesmo *log* l_d é decidido em cada réplica. Então, cada réplica consulta o seu próprio *log* e executa as requisições faltantes em l_d, ou realiza o *rollback* das requisições mais adiantadas em relação a l_d.

Note que um procedimento similar ao anteriormente descrito pode ser executado para realizar a limpeza periódica do *log* das réplicas.

Considerações adicionais. Um sequenciador malicioso pode escolher as requisições de um cliente para descartar. Como em outras implementações (por exemplo, o BFT-SMaRt [Bessani et al. 2014]), para evitar tal problema, os clientes devem enviar as requisições também para as réplicas, as quais iniciam um *timeout* para o recebimento de suas ordens a partir do sequenciador. Caso o *timeout* expire, a réplica deve enviar a requisição como se fosse o cliente e iniciar um novo *timeout*. Caso o novo *timeout* expire, a réplica suspeita do sequenciador e inicia uma troca de visão.

A implementação do componente USIG na camada de rede pode ser desafiadora. No entanto, é possível seguir abordagens já propostas, como a do NeoBFT [Sun et al. 2023], que utilizam um co-processador acoplado ao *switch* para executar operações mais complexas.

3.4. Variações do Protocolo: MinNsoBFT e MinZyzNsoBFT

Conforme apresentado na Tabela 1, existem duas variações possíveis para o NsoBFT, ambas reduzindo a quantidade de réplicas para apenas $2f + 1$. O aspecto fundamental a ser garantido é que requisições já completas não sejam removidas do *log* das réplicas durante uma troca de visão. Com f réplicas a menos, este desafio é superado da seguinte forma:

- **MinNsoBFT**: Esta variante funciona de forma similar ao MinBFT [Veronese et al. 2013], demandando um passo adicional de comunicação antes da execução de uma requisição. Assim, os quóruns são formados por apenas $f + 1$ réplicas, mas o passo adicional garante que mensagens respondidas por pelo menos $f + 1$ réplicas permanecerão no *log* durante a troca de visão.

- **MinZyzNsoBFT**: Esta variante funciona de forma similar ao MinZyzzyva [Veronese et al. 2013]. Neste caso, os clientes devem aguardar por todas as respostas $(2f+1)$ para completar uma operação. Em execuções onde a rede está lenta ou há réplicas maliciosas, é necessária uma etapa adicional, onde o cliente envia uma mensagem para todas as réplicas e aguarda as respostas. O objetivo dessas mensagens é garantir que pelo menos $f + 1$ réplicas executaram a requisição em ordem.

Note também que, para ambas as variantes, o protocolo de consenso utilizado para a troca de visão deve tolerar falhas bizantinas e considerar apenas $2f + 1$ réplicas. Um exemplo de protocolo que pode ser utilizado é o MinBFT [Veronese et al. 2013].

4. Avaliação Experimental

Esta seção descreve a avaliação experimental dos protocolos apresentados neste trabalho. Primeiramente, o ambiente e a metodologia empregados nos experimentos são detalhados. Em seguida, os resultados obtidos são descritos e discutidos.

4.1. Ambiente e Metodologia

Todos os protocolos que utilizam ordenação na camada de rede (NOPaxos, NeoBFT e NsoBFT) foram implementados em Java, e os experimentos foram realizados no Emulab [White et al. 2002]. O ambiente para a execução dos experimentos foi constituído por 6 máquinas *d430* (2.4 GHz E5-2630v3, com 8 núcleos e 2 *threads* por núcleo, 64GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. As réplicas do NOPaxos foram configuradas em 3 servidores, enquanto as réplicas do NeoBFT e do NsoBFT foram configuradas com 4 servidores, para tolerar até uma falha. Cada servidor executou em uma máquina dedicada, enquanto um número variado de clientes foi executado em outra máquina. O sequenciador também foi executado em uma máquina dedicada. Essa abordagem foi também uma das utilizadas na avaliação do NOPaxos.

O ambiente de *software* utilizado consistiu do sistema operacional Ubuntu 20 64-*bit* e máquina virtual Java de 64 *bits* versão 1.8_431. Também foi utilizada a biblioteca de criptografia *Bouncy Castle* para gerar e verificar assinaturas de 256 bits. Para verificar o desempenho das diferentes abordagens, foi implementado um serviço "vazio", *i.e.*, nada é processado nas réplicas. A partir desta implementação, o número de clientes foi variado e a vazão foi medido em uma das réplicas, enquanto a latência foi medida em um dos clientes. Além disso, o tamanho do *payload* variou entre 0B, 100B, 1kB e 4kB.

4.2. Resultados

A Figura 3 apresenta os resultados relacionados à latência e vazão para todas as abordagens estudadas. Conforme esperado, o NOPaxos apresenta um desempenho melhor por tolerar apenas falhas *crash* e demandar um número menor de réplicas. Além disso, esse protocolo não utiliza primitivas mais custosas de criptografia. Para o NOPaxos, a vazão passou de 25 kops/seg para *payloads* de 0B e 100B. Já para um *payload* de 1 kB, a vazão chegou a aproximadamente 24 kops/seg. Finalmente, para *payloads* maiores de 4kB, a vazão chegou a aproximadamente 9 kops/seg. Por outro lado, em todos os casos, a latência inicia relativamente baixa e aumenta gradativamente à medida em que o sistema satura e atinge o pico da vazão. A partir deste ponto, apenas a latência aumenta. Dentre os

protocolos que toleram falhas bizantinas, o NsoBFT apresentou um desempenho superior ao NeoBFT, principalmente por demandar um passo a menos de comunicação. Note que ambos os protocolos utilizam uma assinatura no sequenciador.

Além do *payload*, as mensagens ainda continham outras informações, como identificadores da mensagem e do cliente, além do número de sequência. As mensagens sem assinaturas possuíam um *overhead* de 190 bytes, enquanto que nas mensagens com assinaturas este valor foi de 536 bytes. A utilização da serialização do Java (interface *Serializable*) também contribuiu para este *overhead*. Também foram medidos os tempos médios de processamento dos algoritmos criptográficos. O tempo médio para o sequenciador assinar um pacote foi de aproximadamente $1,81$ ms, e para uma réplica verificar a sua validade foi de aproximadamente $0,22$ ms, resultando em um total de $2,03$ ms de *overhead* de processamento nas soluções que toleram falhas bizantinas.

A vazão do NsoBFT ultrapassou os 7 kops/seg para *payloads* de 0 B, enquanto que o NeoBFT apenas superou os 5 kops/seg; no outro extremo, para *payloads* de 4 kB, o NsoBFT alcançou uma vazão superior a 5 kops/seg, enquanto que no NeoBFT este valor ficou próximo dos 3 kops/seg. Assim como no NOPaxos, a latência permanece baixa até o ponto de saturação do sistema. Porém, é importante notar que, no NeoBFT, o sistema é saturado mais rapidamente, pois este protocolo exige um passo a mais de comunicação do que o NsoBFT.

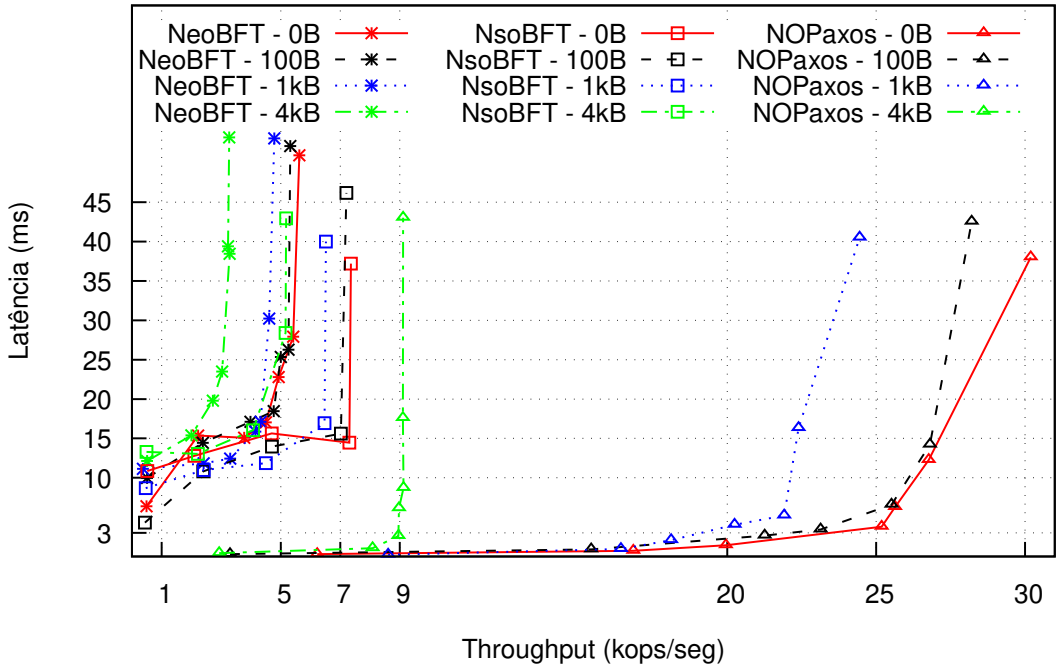


Figura 3. Relação entre latência (95 percentil) e vazão.

As Tabelas 2 e 3 apresentam, respectivamente, a vazão e a latência para as três soluções quando variamos a quantidade de clientes (1, 5, 10, 25, 50 e 100) e o tamanho do *payload* (0B, 100B, 1kB e 4kB). Ressalta-se que, para algumas configurações, a quantidade de clientes não chega a saturar o sistema, diferentemente do experimento cujos resultados são apresentados na Figura 3, em que foram usados clientes suficientes para

alcançar o ponto de saturação.

A vazão (Tabela 2) varia consideravelmente entre os algoritmos e conforme o número de clientes aumenta, evidenciando que a eficiência pode ser impactada pela carga no sistema. Observando os resultados, nota-se que, em geral, o algoritmo NOPaxos tende a apresentar vazão mais alta para a maioria das combinações de clientes e tamanhos de *payload*, especialmente em configurações com menor número de clientes e *payloads* maiores. Por outro lado, o NeoBFT e o NsoBFT mostram desempenhos inferiores em comparação ao NOPaxos, especialmente com cargas mais intensas. Este comportamento deve-se principalmente ao fato de o NOPaxos não empregar protocolos criptográficos e usar menos réplicas. Da mesma forma, o NsoBFT apresenta resultados superiores ao NeoBFT por demandar um passo a menos de comunicação.

	NOPaxos				NeoBFT				NsoBFT			
Clientes	0 B	100 B	1 kB	4 kB	0 B	100 B	1 kB	4 kB	0 B	100 B	1 kB	4 kB
1	6.22	5.30	5.60	5.93	0.49	0.51	0.37	0.51	0.51	0.43	0.46	0.48
5	16.85	15.43	16.43	8.08	2.22	2.38	2.41	2.00	2.09	1.99	1.84	1.84
10	19.96	21.25	18.11	8.95	3.77	3.98	3.30	2.72	4.69	4.70	4.49	4.07
25	25.18	23.13	20.25	8.97	4.50	4.76	4.15	3.03	7.29	7.01	6.46	4.57
50	25.63	25.52	21.90	9.12	4.93	5.00	4.32	3.28	7.35	7.20	6.51	5.18
100	26.76	26.81	22.39	9.10	5.62	5.96	4.61	3.23	7.40	7.56	6.46	5.35

Tabela 2. Vazão (kops/seg) variando o número de clientes e o tamanho do *payload*.

A latência (Tabela 3) também varia conforme a quantidade de clientes e o tamanho do *payload*. É possível observar que, para o algoritmo NOPaxos, a latência se mantém relativamente baixa em comparação aos demais algoritmos, mesmo com o aumento do número de clientes e do tamanho do *payload*, pois o sistema ainda não atinge o ponto de saturação. Para o NeoBFT, a latência aumenta consideravelmente, especialmente quando o número de clientes é elevado e o *payload* é maior. O NsoBFT também apresenta latências elevadas em cargas semelhantes, mas, de maneira geral, ainda inferiores às marcas verificadas para o NeoBFT.

	NOPaxos				NeoBFT				NsoBFT			
Clientes	0 B	100 B	1 kB	4 kB	0 B	100 B	1 kB	4 kB	0 B	100 B	1 kB	4 kB
1	0.29	0.30	0.23	0.42	9.39	9.99	11.14	13.13	8.85	8.30	8.68	12.29
5	0.73	0.90	0.94	1.05	15.35	14.44	12.83	15.38	12.79	10.81	10.99	13.03
10	1.42	2.66	2.07	2.60	15.09	17.11	20.43	19.20	15.63	13.95	11.87	16.03
25	3.79	3.43	4.04	6.14	17.04	18.46	25.85	33.50	14.45	15.60	16.94	28.39
50	6.27	6.59	5.20	8.70	32.78	45.34	37.10	48.45	37.19	40.18	40.01	42.96
100	12.26	14.21	16.30	17.58	47.92	47.27	55.23	99,40	47.37	46.08	52.96	84.08

Tabela 3. 95 percentil da latência (ms) variando o número de clientes e o tamanho do *payload*.

Os resultados experimentais apresentados permitem embasar a decisão pela adoção de um ou outro protocolo. Deve-se levar em consideração uma análise minuciosa dos requisitos, tanto em relação ao desempenho quanto à confiabilidade do sistema tolerante a falhas pretendido. Entretanto, fica clara a superioridade do NsoBFT para tolerância a intrusões.

5. Conclusão

O desenvolvimento de protocolos de consenso é crucial para a construção de sistemas distribuídos confiáveis, especialmente em ambientes sujeitos a intrusões. A proposta do

NsoBFT representa um avanço significativo em relação ao NeoBFT, ao introduzir uma solução que reduz a coordenação necessária entre réplicas e utiliza identificadores sequenciais únicos para mitigar o impacto de sequenciadores maliciosos. Essa abordagem não apenas melhora a eficiência ao demandar um passo a menos de comunicação, mas também mantém a robustez necessária para lidar com falhas bizantinas, contribuindo, assim, para a evolução dos sistemas tolerantes a falhas.

Adicionalmente, a análise experimental comparativa realizada e apresentada neste artigo fornece uma visão dos *trade-offs* entre desempenho e resiliência. Enquanto o NOPaxos destaca-se em desempenho, mas apenas tolerando falhas *crash*, a implementação do NsoBFT demonstra ser uma alternativa viável que equilibra eficiência e tolerância a falhas bizantinas, tornando-se uma opção atraente para aplicações críticas que requerem alta disponibilidade e integridade de dados. Como trabalhos futuros, pretende-se implementar o sequenciador junto ao *switch* e também realizar sua implementação com funções virtualizadas de redes. Além disso, devem ser implementadas e avaliadas as variações do NsoBFT (MinNsoBFT e MinZyzNsoBFT).

Referências

Alchieri, E. A. P., Bessani, A., Greve, F., and Fraga, J. d. S. (2018). Knowledge connectivity requirements for solving byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing*, 15(2):246–259.

Bessani, A., Sousa, J., and Alchieri, E. E. P. (2014). State machine replication for the masses with bft-smart. In *International Conference on Dependable Systems and Networks*, pages 355–362. IEEE.

Bravo, M., Chockler, G., and Gotsman, A. (2022). Making byzantine consensus live. *Distributed Computing*, 35(6).

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, pages 173–186, Berkeley, CA, USA. USENIX.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Douceur, J. R. (2002). The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, New York - USA.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

Li, C., Qiu, W., Li, X., Liu, C., and Zheng, Z. (2024). A dynamic adaptive framework for practical byzantine fault tolerance consensus protocol in the internet of things. *IEEE Transactions on Computers*, 73(7):1669–1682.

Li, J., Michael, E., Sharma, N. K., Szekeres, A., and Ports, D. R. (2016). Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *Symposium on Operating Systems Design and Implementation*, pages 467–483. USENIX.

Liu, X. and Yu, W. (2024). A review of research on blockchain consensus mechanisms and algorithms. In *International Conference on Intelligent Informatics and Biomedical Sciences*, volume 9, pages 1–10.

Saramago, R. Q., Alchieri, E. A., Rezende, T. F., and Camargos, L. (2018). On the impossibility of byzantine collision-fast atomic broadcast. In *International Conference on Advanced Information Networking and Applications*, pages 414–421. IEEE.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Singh, A., Kumar, G., Saha, R., Conti, M., Alazab, M., and Thomas, R. (2022). A survey and taxonomy of consensus protocols for blockchains. *Journal of Systems Architecture*, 127:102503.

Sun, G., Jiang, M., Khooi, X. Z., Li, Y., and Li, J. (2023). NeoBFT: Accelerating byzantine fault tolerance using authenticated in-network ordering. In *ACM Special Interest Group on Data Communications Conference*, page 239–254, New York, NY, USA. ACM.

Vassantlal, R., Alchieri, E., Ferreira, B., and Bessani, A. (2022). Cobra: Dynamic proactive secret sharing for confidential bft services. In *Symposium on Security and Privacy*, pages 1335–1353. IEEE.

Venâncio, G., Fulber-Garcia, V., Flauzino, J., Alchieri, E. A., and Duarte, E. P. (2024). Dependable virtual network services: An architecture for fault-and intrusion-tolerant sfcs. In *Conference on NFV and SDN*, pages 1–6. IEEE.

Venâncio, G., Turchetti, R. C., and Duarte Jr, E. P. (2022). Nfv-coin: unleashing the power of in-network computing with virtualization technologies. *Journal of Internet Services and Applications*, 13(1):46–53.

Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. (2013). Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30.

Vukolić, M. (2015). The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *IFIP WG 11.4 International Workshop Open Problems in Network Security*, pages 112–125. Springer.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Rev.*, 36(SI):255–270.

Zou, Y., Yang, L., Jing, G., Zhang, R., Xie, Z., Li, H., and Yu, D. (2024). A survey of fault tolerant consensus in wireless networks. *High-Confidence Computing*, 4(2):100202.