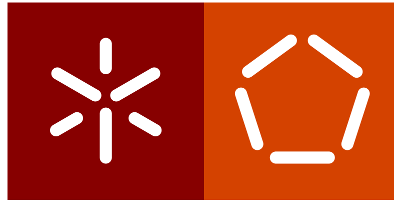
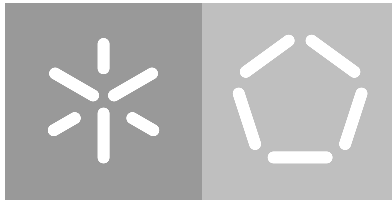


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Fábio Luís Baião da Silva

Acordo Bizantino Mutável para a Blockchain

novembro de 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Fábio Luís Baião da Silva

Acordo Bizantino Mutável para a Blockchain

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação de
José Orlando Pereira
Ana Nunes Alonso

novembro de 2019

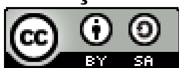
DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Atribuição-CompartilhaIgual
CC BY-SA
https://creativecommons.org/licenses/by-sa/4.0/

AGRADECIMENTOS

Aos meus orientadores, Professor José Orlando Pereira e Ana Nunes Alonso, pelo apoio prestado e pelos conselhos que deram.

Aos meus amigos e colegas pelo ambiente de trabalho proporcionado e pelos momentos de descontração fora dele.

Aos meus pais pela oportunidade que me concederam de estudar.

Este trabalho é financiado por fundos nacionais através da FCT – Fundação para a Ciência e a Tecnologia, I.P., no âmbito do projeto: UID/EEA/50014/2019.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

O principal componente de um sistema *blockchain* é o protocolo de acordo distribuído que tem de ser capaz de tolerar faltas bizantinas na chegada a decisões. Existem muitas implementações de *blockchain*, cada uma utilizando diferentes protocolos de acordo, porém todos eles revelam limitações. Implementações cujo protocolo é da categoria *Proof of*, apesar de escalarem, implicam compromissos entre desempenho e coerência. Protocolos ditos tradicionais (e.g. PBFT) são muito restritos na escalabilidade que oferecem, não conseguindo manter o desempenho ao aumentar o número de participantes. Para além disso, cada protocolo foca-se em características particulares com padrões de comunicação específicos, pelo que para alterar algum destes aspetos é necessário substituir o protocolo de acordo.

Neste trabalho propõe-se um protocolo que combina a tolerância a faltas bizantinas com as características do protocolo *Mutable Consensus* que admite diferentes padrões de comunicação aplicáveis a diferentes ambientes. Adicionalmente, um desses padrões que privilegia uma comunicação por difusão epidémica (*gossip*) oferece grande escalabilidade, permitindo assim construir um protocolo que também possa escalar.

Palavras chave: *blockchain*, acordo distribuído, faltas bizantinas

ABSTRACT

The main component of a blockchain system is the consensus protocolo that must tolerate byzantine faults. There are many blockchain implementations, each one using a distinct consensus protocol, though all of them have limitations. Some use a protocol from the Proof of family, that exhibit tradeoffs regarding consistency and peformance. Others rely in more traditional protocols (e.g. PBFT), whose biggest disadvantage is its poor scalability. Additionally, protocols have their own communication patterns and properties, and to change any of those it is necessary to replace the whole protocol.

This dissertation aims to build a protocol that combines byzantine fault tolerance with the features of the Mutable Consensus protocol which allows to build multiple communications patterns adaptable to different requirements. Moreover, one of those patterns, that spreads messages in an epidemic manner (gossip), offers great scalability, thus allowing to create a scalable protocol.

Keywords: blockchain, distributed consensus, byzantine faults

CONTEÚDO

1 INTRODUÇÃO 1
1.1 Problema 1
1.2 Objetivos 2
1.3 Estrutura da Dissertação 2
2 CONTEXTUALIZAÇÃO 3
2.1 Modelos 3
2.2 Tolerência a faltas 4
2.3 Protocolos que toleram faltas por paragem 5
2.3.1 Paxos 5
2.3.2 Mutable Consensus 6
2.4 Protocolos que toleram faltas bizantinas 7
2.4.1 Practical Byzantine Fault Tolerance 8
2.4.2 XPaxos 10
2.4.3 Outros 12
2.4.4 Protocolos adaptáveis 13
2.5 Acordo distribuído na blockchain 13
2.5.1 Proof of Work 13
2.5.2 Proof of Stake 14
2.5.3 Proof of Elapsed Time 14
2.6 Discussão 14
3 ANÁLISE DE ESCALABILIDADE 16
3.1 XPaxos ou XpensivePaxos 17
3.1.1 Algoritmo de gossip base 18
3.1.2 Envios condicionais 20
3.1.3 Retransmissões 21
3.1.4 Discussão 22
3.2 XPaxos ou PBFT 23
3.2.1 Mesma população 23
3.2.2 Mesma tolerância a faltas 24
3.3 Discussão 25
4 MUTABLE BFT 26
4.1 Alteração à condição prepared 27
4.1.1 Viabilidade da alteração 27

4.2 Especificação do protocolo 29
4.2.1 Canais *Stubborn* 32
4.2.2 Mutações 35
4.3 Otimizações 40
5 avaliação experimental 41
5.1 Implementação 41
5.2 Resultados 42
6 conclusão 47
6.1 Trabalho futuro 48

a resultados da análise de escalabilidade 51
b especificação completa do protocolo 57
b.1 Cliente 57
b.1.1 Transições 57
b.2 Processo 58
b.2.1 Estado 58
b.2.2 Funções auxiliares 58
b.2.3 Transições 59
b.2.4 Mutações 66
c resultados da avaliação experimental 69

LISTA DE FIGURAS

Figura 1	Comparação entre protocolos no modelo XFT	19
Figura 2	Comparação entre protocolos no modelo XFT com envios condicionais	20
Figura 3	Comparação entre protocolos no modelo XFT com retransmissões	22
Figura 4	Comparação entre XPaxos e PBFT para a mesma população	23
Figura 5	Comparação entre XPaxos e PBFT para a mesma tolerância	24
Figura 6	Algumas situações e respetivos comportamentos admitidos pelo protocolo Mutable BFT	26
Figura 7	Número médio de mensagens recebidas por processo em cada decisão e tamanho médio de cada mensagem	44
Figura 8	Latência média de cada decisão	45

LISTA DE ABREVIATURAS

BFT *Byzantine Fault Tolerance.*
PBFT *Practical Byzantine Fault Tolerance.*
PoW *Proof of Work.*
XFT *Cross Fault Tolerance.*

1 INTRODUÇÃO

A *blockchain* introduziu a possibilidade de confirmar transações de forma descentralizada, o que permite a remoção de intermediários. Adicionalmente, as transações são imutáveis não sendo, portanto, possível alterá-las após terem sido confirmadas. A sua popularidade surgiu com as criptomoedas, principalmente a *Bitcoin* [Nakamoto (2008)], que permitem efetuar pagamentos eletrónicos. No entanto, o potencial desta tecnologia gerou interesse em aplicá-la a outras áreas, como por exemplo, o *Everledger* [Everledger] que permite o registo de autenticidade e propriedade de itens, como diamantes.

Sendo as transações replicadas por todos os participantes torna-se necessário manter a coerência das mesmas. Para isso recorre-se a um protocolo de acordo distribuído que permite aos participantes de um sistema concordarem num dado valor. Porém, estes acordos devem ser possíveis mesmo que alguns participantes sejam faltosos. Pode-se distinguir dois tipos de faltas: faltas por paragem e faltas bizantinas. As primeiras apenas se aplicam a interrupções no processamento enquanto que as segundas também contemplam comportamentos maliciosos. Uma vez que os participantes na *blockchain* podem ser maliciosos é necessário tolerar faltas bizantinas.

O protocolo *Practical Byzantine Fault Tolerance* (*PBFT*) [Castro et al. (1999b)] é a referência na tolerância a faltas bizantinas. No entanto, o sucesso da *blockchain* fez surgir novos protocolos de acordo que toleram comportamentos maliciosos. A *Bitcoin* introduziu o *Proof of Work* (*PoW*) ao qual se seguiram outros protocolos do tipo *Proof of* e, de adaptações de protocolos clássicos como o já mencionado *PBFT*, emergiram protocolos com novos conceitos.

1.1 PROBLEMA

O acordo distribuído na *blockchain* é o que permite que transações possam ser confirmadas de forma descentralizada. No entanto os protocolos existentes apresentam problemas que impedem a utilização generalizada da *blockchain*. Protocolos clássicos, como o *PBFT*, revelam péssima escalabilidade, conseguindo desempenhos aceitáveis apenas para um pequeno número de participantes. Isto deve-se ao elevado número de mensagens que é necessário trocar para atingir um acordo. Os protocolos *Proof of* removem a troca de mensagens em

grande número, conseguindo assim escalar, porém possibilitam a apresentação de estados corretos divergentes em participantes distintos, para além de outras limitações particulares a cada protocolo.

Outro aspeto problemático reside nas diferenças entre os protocolos, nomeadamente dos padrões de comunicação. O desempenho de um determinado padrão depende das características do sistema em que executa. Uma vez que diferentes sistemas apresentam diferentes características, é necessária uma escolha cuidada do protocolo a integrar numa implementação de *blockchain*. Caso as características do sistema posteriormente se alterem, torna-se necessário alterar a implementação da *blockchain* para abrigar um novo protocolo.

Assim, é importante a existência de um protocolo de acordo distribuído por troca de mensagens que tolere faltas bizantinas e que, simultaneamente, seja escalável e adaptável.

1.2 OBJETIVOS

O protocolo *Mutable Consensus* [Pereira and Oliveira (2004)] permite alterar padrões de comunicação, chamadas mutações, sem afetar a correção do protocolo. Adicionalmente, uma dessas mutações, que privilegia a comunicação por difusão epidémica (*gossip*), apresenta uma elevada escalabilidade, conseguindo assegurar um bom desempenho com centenas de participantes. Assim, estas características permitem facilitar a adaptação às mudanças das características do sistema e aumentar a escalabilidade do acordo distribuído. No entanto, este protocolo apenas tolera faltas por paragem, não sendo evidente como torná-lo tolerante a faltas bizantinas, devido às diferentes implicações entre tolerar estes dois tipos de faltas.

Pretende-se então desenhar e implementar um protocolo de acordo distribuído que tolere faltas bizantinas e que, simultaneamente, seja adaptável. Para isso, será aplicada a capacidade de alteração dos padrões de comunicação a um protocolo tolerante a faltas bizantinas. Por sua vez, através do padrão de comunicação *gossip* espera-se conseguir fazer escalar o protocolo.

1.3 ESTRUTURA DA DISSERTAÇÃO

No capítulo 2 são enunciados alguns conceitos base relacionados com o acordo distribuído. e descreve-se alguns protocolos para diferentes modelos de faltas. O capítulo 3 contém uma análise do potencial de escalabilidade de dois dos protocolos descritos por forma a escolher a qual se aplica as características do protocolo *Mutable Consensus*. No capítulo 4 é apresentado o novo protocolo criado através das alterações realizadas ao protocolo escolhido. O capítulo 5 aborda a avaliação experimental do protocolo criado, comparando-a também à do protocolo escolhido. Por fim, no capítulo 6 é feita uma reflexão do trabalho realizado e enuncia-se potenciais caminhos para trabalho futuro.

2

CONTEXTUALIZAÇÃO

O acordo distribuído consiste em ter um conjunto de processos a concordar num determinado valor mesmo na presença de processos faltosos. Qualquer protocolo deve satisfazer as propriedades de validade, unanimidade e terminação [Coulouris et al. (2005)]. A **validade** requer que se algum processo decide um valor, então esse valor foi proposto por algum processo. A **unanimidade** afirma que nenhum par de processos decide valores diferentes. Por fim, a **terminação** define que, inevitavelmente, todos os processos corretos decidem algum valor. Estas propriedades asseguram a disponibilidade e coerência do serviço em que o protocolo é aplicado.

2.1 MODELOS

A implementação de um protocolo de acordo distribuído requer a especificação de um conjunto de atributos que modelem o sistema a considerar por forma a definir os comportamentos expectáveis do mesmo [Schneider (1993)].

Um dos aspetos a definir é o tipo de faltas a tolerar, uma vez que qualquer sistema está sujeito a ter componentes faltosos, sejam eles processos ou canais de comunicação. A tolerância a faltas por paragem considera que os componentes de um sistema apenas podem deixar de funcionar. Por exemplo, processos param de processar ou canais de comunicação deixam de entregar mensagens. Na tolerância a faltas bizantinas os componentes podem assumir comportamentos arbitrários, como por exemplo processos maliciosos que procuram corromper o sistema. Também os canais de comunicação podem apresentar adversidades, por exemplo alterando as mensagens.

Outro parâmetro a considerar na modelação é a sincronia quer dos processos quer dos canais de comunicação. Sistemas síncronos definem limites para o processamento e para a latência da rede, enquanto que sistemas assíncronos não impõem quaisquer limites nestes aspetos. Os sistemas assíncronos definem com maior precisão a maioria dos sistemas e para além disso, algoritmos definidos para sistemas assíncronos são universais, na medida em que funcionam independentemente da sincronia considerada. Por esta razão, assumir modelos que consideram o sistema assíncrono é mais apelativo.

No entanto, a impossibilidade de FLP [Fischer et al. (1982)] indica que não é possível resolver o acordo distribuído para um sistema assíncrono na presença de faltas por paragem. De forma a contornar este problema existem duas soluções: assumir algum sincronismo ou recorrer a detetores de falhas. A sincronia parcial consiste em considerar que em algum ponto os processos comunicarão sincronamente. Isto pode ser assumido apenas para garantir disponibilidade, não sendo necessário para garantir a coerência. Os detetores de falhas reforçam algoritmos para modelos assíncronos, porém podem suspeitar incorretamente da falha de processos.

Os detetores de falhas [Chandra and Toueg (1996)] são caracterizados por duas propriedades: totalidade e precisão. A totalidade indica que, inevitavelmente, um processo que falha é permanentemente suspeitado e a precisão garante que processos corretos nunca são suspeitados. Ambos são medidos em duas grandezas, forte e fraca, de acordo com o número de processos envolvidos. Relativamente à totalidade, esta é forte caso a suspeita seja feita por todos os processos corretos enquanto que é fraca se apenas for garantido que algum processo correto tenha essa suspeita. Para a precisão, ela é forte caso se aplique a todos os processos e é fraca se se aplicar apenas a algum. A precisão pode ainda ser relaxada para que apenas seja garantida inevitavelmente em qualquer uma das grandezas.

As várias combinações destas propriedades definem um conjunto de classes de detetores de faltas. Variam desde os inevitavelmente fracos, cujas propriedades são a totalidade fraca e precisão inevitavelmente fraca, até aos perfeitos que garantem totalidade forte e precisão forte.

2.2 TOLERÊNCIA A FALTAS

Protocolos para um modelo assíncrono que apenas toleram faltas por paragem garantem disponibilidade desde que a maioria das processos sejam corretos. enquanto que a coerência é garantida independentemente do número de processos faltosos. Ou seja, para tolerar um máximo de t faltas, neste modelo, são necessários 2t+1 processos.

Por sua vez, protocolos que toleram faltas bizantinas num modelo assíncrono necessitam de mais de dois terços de processos corretos para se manterem disponíveis. Relativamente à garantia de coerência, esta é mantida desde que existam, no máximo, um terço de processos maliciosos. Assim sendo, são necessários 3t+1 processos para tolerar t faltas. Este modelo é conhecido por tolerância a faltas bizantinas (*Byzantine Fault Tolerance* (*BFT*)) [Lamport et al. (1982)] em que é assumido um adversário muito forte que pode coordenar os processos faltosos e controlar a rede.

Existem porém modelos que relaxam algumas propriedades do modelo *BFT*. Por exemplo, o modelo *Cross Fault Tolerance* (*XFT*) [Liu et al. (2016)] tolera faltas bizantinas mas não assume um adversário capaz de controlar todas os processos faltosos e, simultaneamente, a

rede. Assim, atrasos na rede (assincronia) são consideradas faltas dos processos que ficam particionados. Desta forma, consegue-se tolerar t faltas com apenas 2t+1 processos porém, em qualquer momento, tem de haver uma maioria correta e que comunique sincronamente.

2.3 PROTOCOLOS QUE TOLERAM FALTAS POR PARAGEM

Apresenta-se de seguida dois protocolos que toleram faltas por paragem com comunicação assíncrona. Em primeiro lugar, o protocolo *Paxos*, o mais famoso para este modelo, e de seguida o protocolo *Mutable Consensus*.

2.3.1 *Paxos*

O protocolo *Paxos* [Lamport et al. (2001); Chandra et al. (2007)] processa-se em três fases. Na primeira fase ocorre a eleição de um coordenador em que os processos candidatam-se enviando uma mensagem Propose, que contém um número, aos restantes processos. Cada um desses processos responde com uma mensagem Promise caso o número recebido seja o mais alto até então e passam a rejeitar mensagens de coordenadores antigos. Quando o processo candidato receber mensagens Promise de uma maioria de processos torna-se no coordenador. A segunda fase inicia o acordo por um dado valor com o coordenador a enviar esse valor numa mensagem Accept aos restantes processos. Os processos respondem a essa mensagem caso o remetente seja o coordenador, tal como definido na primeira fase. O início da terceira fase dá-se quando o coordenador receber respostas de uma maioria de processes. Nessa altura chegou-se a acordo sobre o valor e o coordenador envia uma mensagem Commit a informar todas os processos desse acordo.

Quando ocorre a eleição de um novo coordenador já pode ter sido atingido um acordo anteriormente. Nesse caso é necessário informar o processo candidato desse valor, o que é feito incluindo na mensagem Promise esse mesmo valor juntamente com o número do coordenador correspondente. O processo candidato, ao receber essas mensagens, escolhe o valor correspondente ao coordenador mais recente.

O protocolo descrito apenas permite chegar a um acordo para um único valor. Para se chegar a acordo sobre uma sequência de valores basta aplicar o protocolo *Paxos* sucessivamente sobre os diferentes valores [Chandra et al. (2007)]. Uma otimização possível é a de o coordenador ser o mesmo nas várias iterações do protocolo, o que permite remover a primeira fase do protocolo em execuções consecutivas. A qualquer momento, no entanto, pode ser eleito um novo coordenador para tolerar a falha do anterior.

Para reduzir o número de mensagens trocadas no protocolo *Paxos* foi proposto o protocolo *Cheap Paxos* [Lamport and Massa (2004)]. Este protocolo considera t+1 processos principais e aos restantes t denomina-os de auxiliares. Apenas os processos principais participam na

chegada ao acordo por um valor. Desta forma, é necessário que todos os processos principais estejam disponíveis para que acordos possam ser alcançados. Quando algum dos processos principais falha é necessário alterar o conjunto de processos principais, altura em que os processos auxiliares também participam. O funcionamento normal do protocolo é o mesmo do *Paxos*, com a exceção de os processos para os quais as mensagens são enviadas serem apenas os que compõem o conjunto dos processos principais.

2.3.2 *Mutable Consensus*

O protocolo *Mutable Consensus* [Pereira and Oliveira (2004)] processa-se em rondas em que cada ronda admite um coordenador escolhido de forma determinística.

As mensagens trocadas neste protocolo incluem a ronda, o valor proposto e o conjunto de processos que concorda com o valor. Cada processo mantém um estado que inclui a ronda em que se encontra, o valor para essa ronda e o conjunto de processos que concorda com esse valor. No início de cada ronda o conjunto de processos é vazio.

O coordenador propõe um valor enviando a todos os processos uma mensagem cujo conjunto de processos apenas contém o próprio coordenador. Os processos, ao receberem uma mensagem, verificam se o conjunto de processos que se encontra na mensagem inclui processos que o seu estado ainda não considera. Caso isso se verifique, adiciona esses processos ao seu conjunto, incluindo-se a si próprio, e envia uma mensagem a todos os processos com o novo conjunto. Este comportamento é repetido até que o conjunto de processos seja uma maioria, momento em que é atingido o acordo.

Quando um processo suspeita da falha do coordenador este transita para a fase 2 enviando uma mensagem a todos os processos com essa indicação. Tal como acontece para atingir o acordo sobre um valor, são trocadas mensagens até que o conjunto de processos contenha uma maioria. Nesse momento, avança-se para a próxima ronda, substituindo assim o coordenador.

Caso os processos se encontrem em rondas atrasadas em relação a uma mensagem recebida, atualizam o seu estado de acordo com essa mensagem. Adicionalmente, mesmo que a ronda seja igual à de uma mensagem recebida, se a fase for diferente avançam também para a fase 2.

Este protocolo utiliza canais de comunicação insistentes (*stubborn*) [Guerraoui and Oliveira (1996)] em que o envio de uma mensagem não implica necessariamente a sua transmissão. Adicionalmente, um canal insistente retransmite periodicamente as últimas k mensagens enviadas. Neste protocolo apenas é guardada a última mensagem enviada em cada canal, portanto $k = 1$. De cada vez que uma mensagem é enviada, o temporizador de retransmissão é reposto.

As características dos canais insistentes permitem definir vários protocolos de comunicação, através do desenho de diferentes comportamentos para a transmissão e a retransmissão de mensagens. São apresentados de seguida os quatro padrões, chamadas mutações, já propostos.

Mutações

Na mutação *early*, cada nova mensagem que seja a primeira de uma ronda é imediatamente transmitida. Adicionalmente, quando for atingido o acordo, isto é, o conjunto de processos apresentar uma maioria, é também feita a transmissão. Numa retransmissão é submetida a última mensagem enviada, sem quaisquer restrições.

A mutação *centralized* admite comportamentos distintos para o coordenador e para os restantes processos. O coordenador assume o mesmo comportamento da mutação *early* enquanto que os restantes processos consideram as mesmas condições, no entanto, apenas transmitem se o destinatário for o coordenador. A retransmissão nesta mutação é realizada como a da mutação *early*, para todos os processos.

Na mutação *ring*, as condições de transmissão são semelhantes às dos restantes processos da mutação *centralized*. Porém, o destinatário aprovado não é o coordenador, mas sim o processo que sucede logicamente o remetente. Ao efetuar a retransmissão da última mensagem enviada, o destinatário aprovado é obtido de forma incremental, isto é, o sucessor do anterior destinatário.

Por fim, a mutação *gossip* revela o mesmo comportamento quer para as transmissões como para as retransmissões. Esta mutação não depende do conteúdo das mensagens, pelo que todas as mensagens enviadas são transmitidas. Porém cada mensagem apenas é transmitida para um conjunto restrito de processos, determinado por um valor de *fanout* e por uma permutação de todos os processos. Esse conjunto vai sendo alternado de forma cíclica, quer nas retransmissões, quer no envio de novas mensagens. Assim, obtém um padrão de comunicação de difusão epidémica que permite escalar o protocolo para um grande número de processos.

2.4 PROTOCOLOS QUE TOLERAM FALTAS BIZANTINAS

Nesta secção apresenta-se protocolos que toleram faltas bizantinas, começando pelo pioneiro para um modelo assíncrono, *PBFT*. De seguida faz-se um paralelismo com o protocolo *XPaxos*, no modelo *XFT*, terminando com alguns protocolos estado da arte.

2.4.1 *Practical Byzantine Fault Tolerance*

O protocolo *PBFT* [Castro et al. (1999b)], no modelo *BFT*, decorre numa sucessão de vistas em que cada vista define um coordenador deterministicamente. Cada vista pode ser dividida em duas partes: o caso normal, em que se processam pedidos de clientes; e a mudança de vista, onde os processos trocam entre si as decisões durante a vista anterior.

Todas as mensagens são assinadas pelos remetentes e cada mensagem aceite é incluída num registo. As mensagens guardadas nesse registo são utilizadas nas mudanças de vista permitindo provar as decisões tomadas.

Caso Normal

O cliente, que sabe a vista em que os processos se encontram, efetua um pedido enviando uma mensagem ao coordenador. O coordenador atribui um número de sequência distinto a cada pedido e envia a todas os processos uma mensagem Pre-Prepare. Essa mensagem inclui o número de sequência definido, a vista em que se encontra e o próprio pedido. Caso a vista que o cliente conhece seja errada, enviando assim o pedido para um processo que não é o coordenador, esse processo reencaminha o pedido ao coordenador.

Cada processo, ao receber o Pre-Prepare verifica se a vista corresponde à vista em que se encontra e se o número de sequência não foi definido a outro pedido diferente, nessa vista. Desta forma, um coordenador maligno não consegue fazer aprovar uma mensagem que não seja para a vista corrente ou para um número de sequência que entre em conflito com outro pedido. Se as verificações forem satisfeitas, o processo envia a todos os processos uma mensagem Prepare que contém a vista, o número de sequência e o *digest* do pedido. Esta mensagem serve para indicar o que foi recebido do coordenador, para que um coordenador maligno não possa enviar mensagens Pre-Prepare distintas para processos diferentes.

Os processos ao receberem estas mensagens apenas verificam se a vista é a atual. Ao conhecerem 2t mensagens Prepare de diferentes processos cujos *digests*, valores de vista e números de sequência correspondam ao de uma mensagem Pre-Prepare, atingem o estado *prepared*. Isto significa que uma maioria (de $\frac{2}{3}$) tomou conhecimento da mesma mensagem proveniente do coordenador e, com isso, chegou-se ao acordo para a vista corrente. Porém, um processo ao atingir este estado não sabe se outros processos também o atingiram, pelo que não tem a garantia de que o acordo se manterá numa vista futura. Assim, nesse momento, são enviadas mensagens Commit que incluem a vista, o número de sequência e o *digest* do pedido, para todos os processos. As mensagens deste tipo têm o intuito de informar que se atingiu o estado *prepared* relativamente ao pedido respetivo, nesta vista.

Tal como acontece com as mensagens Prepare, ao receber estas mensagens apenas é verificada a vista. Neste caso, a condição relativa ao valor da vista é mais relaxada, bastando que seja relativa à vista atual ou a uma vista passada. Assim, ao tomar conhecimento de

2t+1 mensagens Commit com os mesmos *digests*, valores de vista e números de sequência é atingido o estado *committed*. Nesta altura, um processo já conhece uma maioria de processos que atingiram o estado *prepared*, podendo assim considerar o acordo como atingido. O pedido pode então ser executado e é enviada uma resposta ao cliente que inclui o resultado. Os clientes apenas aceitam um resultado quando recebem t+1 respostas com o mesmo resultado de diferentes processos. As respostas incluem ainda a vista em que os processos se encontram, sendo desta forma que os clientes tomam conhecimento da vista.

Quando um cliente não consegue aceitar um resultado em tempo útil, reenvia o pedido a todos os processos. Estes reencaminham o pedido para o coordenador e, caso não executem esse pedido em tempo útil, iniciam uma mudança de vista.

Mudança de Vista

Quando um processo inicia uma mudança de vista deixa de aceitar mensagens do caso normal e envia uma mensagem View-Change a todas os processos. Esta mensagem contém a próxima vista, e todas as mensagens Pre-Prepare e Prepare cujos pedidos atingiram o estado *prepared* mais recentemente, isto é, de acordo com o valor da vista, para cada número de sequência.

Ao receber estas mensagens, para além de se verificar a vista, de forma a que seja superior à atual, também é necessário verificar o conjunto de mensagens Pre-Prepare e Prepare. Este conjunto de mensagens tem de ser correto, na medida em que todas as mensagens têm de pertencer a um conjunto que valide o estado *prepared*. O coordenador da próxima vista ao conhecer 2t+1 mensagens View-Change válidas, envia a todas os processos uma mensagem New-View. Esta mensagem contém a próxima vista, um conjunto com as mensagens View-Change conhecidas e um conjunto de mensagens Pre-Prepare. As mensagens Pre-Prepare são criadas para todos os pedidos que se encontram no conjunto de mensagens que resulta da junção dos conjuntos de mensagens Pre-Prepare e Prepare das mensagens View-Change consideradas. Desta forma, qualquer pedido que tenha atingido o estado *committed* em pelo menos um processo estará representado na junção de mensagens, pois atingiu o estado *prepared* em 2t+1 processos. Mesmo que t processos malignos não representem um pedido propositadamente e sejam escolhidas mensagens View-Change de outros t processos corretos que não tenham conhecimento desse pedido, existirá uma mensagem View-Change do processo 2t+1 que inclui esse pedido.

Na receção de mensagens New-View verifica-se se a vista é superior à atual, se todos as mensagens View-Change são corretas e se as mensagens Pre-Prepare foram criadas corretamente de acordo com as mensagens View-Change. Caso estas verificações se confirmem, são criadas mensagens Prepare para cada mensagem Pre-Prepare. Estas mensagens criadas são enviadas para todos os processos, terminando assim a mudança de vista e voltando a ser aceites mensagens do caso normal.

O envio das mensagens View-Change para os restantes processos apenas serve para garantir disponibilidade. Quando um processo conhece 2t+1 mensagens View-Change de processos diferentes deve concluir a mudança de vista em tempo útil. Caso isso não aconteça, inicia uma nova mudança de vista, o que permite garantir progresso mesmo quando existem vários coordenadores consecutivos faltosos.

2.4.2 *XPaxos*

O protocolo XPaxos [Liu et al. (2016)], no modelo *XFT*, também decorre numa sucessão de vistas com a mesma separação que existe no protocolo *PBFT*. Cada vista define, deterministicamente, um conjunto de processos ativos constituídos por um coordenador e t seguidores. Os restantes processos, passivos, não participam no caso normal.

Tal como no protocolo *PBFT* as mensagens são também assinadas e inseridas num registo.

Caso Normal

O cliente, que sabe a vista em que os processos se encontram, efetua um pedido enviando uma mensagem ao coordenador. O coordenador incrementa o número de sequência, atribui-o ao pedido e envia uma mensagem Prepare aos seguidores. Esta mensagem contém o número de sequência atribuído, a vista em que se encontra e o próprio pedido.

Cada seguidor ao receber uma mensagem Prepare verifica se a vista corresponde à vista em que se encontra e se o número de sequência foi incrementado. Se as verificações forem satisfeitas, o processo envia a todas os processos ativos uma mensagem Commit que contém a vista, o número de sequência e o *digest* do pedido. Estas mensagens correspondem às mensagens Pre-Prepare e Prepare do protocolo *PBFT*, tendo portanto os mesmos objetivos.

Quando um processo ativo tiver recebido mensagens Commit corretas de todos os seguidores, que correspondam à mensagem Prepare é atingido o acordo. Assim, o pedido pode ser executado e é enviado o resultado ao cliente. O cliente apenas aceita o resultado quando receber respostas corretas de todas os processos ativos.

Apesar de o protocolo considerar que os processos corretos comunicam de forma síncrona, sendo isso que reduz o número de processos necessários para tolerar o mesmo número de faltas relativamente ao *PBFT*, a participação de apenas uma maioria de processos permite desconsiderar essa componente no caso normal.

Neste protocolo não é necessária uma nova troca de mensagens, tal como acontece no protocolo *PBFT*, pois os clientes apenas aceitam o resultado ao receberem tantas respostas quanto o número de processos que participa no acordo. Pode-se pensar que o protocolo *PBFT* também podia fazer com que os clientes apenas aceitassem o resultado ao receberem 2t+1 respostas corretas, eliminando assim o segundo passo. No entanto, devido à forma como os processos iniciam mudanças de vista, ou seja, individualmente, um adversário

podia fazer com que alguns processos iniciassem uma mudança de vista enquanto que outros continuavam no caso normal e, desta forma, impedir qualquer progresso. No protocolo *XPaxos*, quando um processo inicia uma mudança de vista os restantes processos também a iniciam imediatamente. Note-se que isto não pode ser explorado por um adversário, pois apenas processos ativos podem iniciar mudanças de vista. Ou seja, caso seja um processo correto a iniciar a mudança de vista, os outros processos podem confiar nele e, caso seja um processo maligno a fazê-lo, este está a acusar-se a si próprio.

Mudança de Vista

As mudanças de vista contam com a participação de todos os processos e são iniciadas por um processo ativo que envia uma mensagem Suspect a todos os processos, deixando de aceitar mensagens do caso normal.

Cada processo ao receber uma mensagem Suspect deixa também de aceitar mensagens para o caso normal e envia uma mensagem View-Change a todas os processos ativos da próxima vista. Esta mensagem contém todas as mensagens Prepare e Commit cujos pedidos chegaram a acordo mais recentemente, isto é, de acordo com o valor da vista, para cada número de sequência.

Os futuros processos ativos esperam o tempo necessário para que tenham a garantia de que receberam mensagens View-Change de todos os processos corretos. Quando esse tempo expira, enviam uma mensagem VC-Final que contém o conjunto de mensagens View-Change recebidas, para todos os processos ativos da próxima vista.

O coordenador, ao receber as mensagens VC-Final de todos os futuros processos ativos, cria uma mensagem Prepare por pedido que tenha chegado ao acordo mais recentemente para cada número de sequência, do conjunto de mensagens que resulta da junção dos conjuntos de mensagens Prepare e Commit incluídas nas mensagens View-Change de cada mensagem VC-Final. Essas mensagens Prepare são incluídas numa mensagem New-View que é enviada a todos os seguidores.

Após os seguidores receberem as mensagens VC-Final de todos os futuros processos ativos e a mensagem New-View, verificam se as mensagens Prepare foram criadas corretamente de acordo com as mensagens VC-Final recebidas. Caso a verificação tenha sucesso, as mensagens Prepare são processadas tal como no caso normal, terminando assim a mudança de vista e voltando a ser aceites mensagens do caso normal.

O passo extra, relativamente ao protocolo *PBFT*, da troca de mensagens VC-Final e a sincronia na receção de mensagens View-Change são necessários para permitir reduzir o número de processos para tolerar o mesmo número de faltas. A troca de mensagens VC-Final entre os processos impede que um coordenador maligno possa selecionar meticulosamente as mensagens View-Change de forma a violar a coerência, o que poderia acontecer se fosse usada a mudança de vista do protocolo *PBFT*. Por sua vez, a espera na receção de mensagens

View-Change garante que são recebidas mensagens de todos os processos corretos, tal como definido no modelo *XFT*.

2.4.3 *Outros*

O protocolo *FireChain* [Mikalsen (2018)] combina um protocolo de *membership* em *gossip* com um protocolo de acordo baseado em *gossip*. O protocolo de *membership* é tolerante a faltas bizantinas e oferece escalabilidade através da difusão epidémica. Este protocolo define um conjunto de anéis que contêm todos os membros dispostos em ordens aleatórias. Cada membro tem, em cada anel, um antecessor e um sucessor, que constituem os vizinhos com quem fazem a troca de mensagens. Os membros monitorizam os seus sucessores, acusando aos vizinhos os que suspeitam terem falhado. Membros falsamente acusados defendem-se com as mensagens que enviam periodicamente. Por sua vez, o protocolo de acordo é baseado no trabalho apresentado em [van Renesse (2016)]. Neste protocolo cada decisão é dividida em várias rondas. Os participantes mantêm uma tabela onde cada participante é associado a uma ronda e um valor e, em cada ronda, cada participante pede a k outros participantes as suas tabelas. As tabelas recebidas são combinadas com a própria tabela de forma a que na tabela resultante cada participante esteja associado ao valor da ronda mais recente desse participante das várias tabelas. O valor do próprio participante é atualizado para o valor que se encontra em maioria nessa tabela resultante. À medida que o número de rondas aumenta há uma convergência relativamente ao valor proposto, sendo assim atingido o acordo.

O protocolo *Swirlds hashgraph* [Baird (2016)] assume o modelo *BFT*. Cada processo mantém uma estrutura, chamada *hashgraph*, que guarda informação sobre os todos as trocas de mensagens que ocorrem. Esta estrutura é difundida de forma epidémica juntamente com as transações, resultando num *overhead* muito reduzido. As estruturas recebidas são combinadas com a própria sendo ainda acrescentada informação sobre essas receções. Através da informação que consta nos *hashgraphs*, os membros conseguem calcular o voto de cada membro, atingindo dessa forma o acordo.

O protocolo *SBFT* [Gueta et al. (2019)] também assume o modelo *BFT*. Este protocolo parte do protocolo *PBFT* acrescentando e alterando algumas propriedades ao mesmo. As principais alterações são os usos de colecionadores e de *threshold signatures*. Com os colecionadores é feita uma alteração do padrão de comunicação deixando de haver comunicação todos para todos. Em vez disso, processos enviam as suas mensagens para um ou alguns colecionadores que tratam de enviar mensagens para todos. Esta alteração é complementada com o uso de *threshold signatures* que permitem reduzir o tamanho das mensagens enviadas pelos colecionadores. Em vez de as mensagens enviadas pelos colecionadores conterem o conjunto de mensagens recebidas, apenas é enviada uma mensagem que substitui esse conjunto.

2.4.4 *Protocolos adaptáveis*

Por fim, apresenta-se uma breve descrição dos dois protocolos tolerantes a faltas bizantinas adaptáveis conhecidos [Bahsoun et al. (2015), Guerraoui et al. (2010)]. Estes protocolos acomodam múltiplos protocolos tolerantes a faltas bizantinas num só, trocando entre os mesmos em diferentes momentos. Porém para efetuar essa troca, é necessário parar o protocolo, pois todos os processos têm de trocar para o mesmo protocolo.

2.5 ACORDO DISTRIBUÍDO NA BLOCKCHAIN

Os protocolos apresentados na secção anterior atingem decisões através de troca de mensagens entre os processos. Uma vez que necessitam que uma maioria chegue a acordo, estão sujeitos a ataques *Sybil* [Douceur (2002)] caso não haja algum tipo de controlo no conjunto de processos. Assim, as *blockchains* em que esses protocolos são utilizados denominam-se de *blockchains* permissionadas, já que processos necessitam de permissão para fazerem parte do conjunto de processos.

Por outro lado, nas *blockchains* não permissionadas, o conjunto de processos é aberto de tal forma que qualquer processo se pode fazer incluir no mesmo. Nesta secção são então apresentados alguns protocolos para este tipo de *blockchains*. Uma vez que estes protocolos são intrínsecos à *blockchain*, importa explicar o funcionamento básico desta. Todos os processos mantêm uma sequência de blocos replicada, em que cada bloco contém uma *hash* relativa ao bloco anterior. Um processo ao adicionar um bloco à sequência usa a *hash* do bloco mais recente, o que impossibilita qualquer alteração aos blocos presentes na sequência e à própria sequência sem que seja detetável através das *hashes*. O acordo distribuído permite então que os processos concordem na adição de cada bloco.

2.5.1 *Proof of Work*

No protocolo *PoW* [Nakamoto (2008)] a medida de tolerância não é o número de processos mas sim o poder computacional. Desta forma, este protocolo garante a correção desde que uma maioria de poder computacional seja correta.

Qualquer processo pode, em qualquer momento, submeter novos blocos porém, para serem aceites, a sua *hash* tem de cumprir um determinado requisito. Esse requisito só pode ser alcançado através de tentativa e erro o que implica um trabalho computacional. Adicionalmente, apenas são aceites blocos que levem à sequência mais comprida, o que é validado através da *hash* relativa ao bloco anterior.

No entanto, podem ocorrer submissões concorrentes que levam a diferentes sequências entre os processos. A política de sequência mais comprida resolve essa disparidade após

novas submissões serem efetuadas. É ainda este comportamento que permite a este protocolo tolerar faltas. Existindo uma maioria de poder computacional correta é impossível que uma minoria faltosa consiga criar uma sequência mais comprida que a construída pela maioria.

2.5.2 *Proof of Stake*

Como alternativa ao enorme gasto energético do protocolo *Proof of Work* foi proposto o protocolo *Proof of Stake*. Neste protocolo, os processos adicionam blocos à sequência com uma probabilidade proporcional à sua participação. A probabilidade é maior para os participantes que têm maior participação, pelo que vai contra o interesse dos próprios apresentar comportamentos maliciosos.

2.5.3 *Proof of Elapsed Time*

Outra alternativa ao protocolo *Proof of Work* é o protocolo *Proof of Elapsed Time* que simula o trabalho computacional do primeiro. Processos esperam tempos aleatórios, diferentes, e apenas ao fim desse tempo submetem um novo bloco, vencendo o primeiro bloco a ser submetido. Para garantir que o tempo a esperar é aleatório (e não um número escolhido minuciosamente) e que esse tempo foi realmente esperado, é necessária a utilização de hardware especializado (e.g. Intel SGX) que permite a qualquer processo fazer estas verificações.

2.6 DISCUSSÃO

As características do protocolo *Mutable Consensus* apenas são aplicáveis a protocolos em que o acordo é atingido através de trocas de mensagens. Para além disso, as limitações na latência e no débito dos protocolos *Proof of* dependem de fatores que não podem ser melhorados através de uma comunicação por difusão epidémica (*gossip*), como se propõe. Desta forma o protocolo a criar será aplicável a *blockchains* permissionadas.

O protocolo *Mutable Consensus* permite trocar de padrão de comunicação em cada processo de forma independente e sem interromper o protocolo. Porém, como já foi visto, este protocolo apenas tolera faltas por paragem, pelo que se justifica a criação de um protocolo tolerante a faltas bizantinas com as características do *Mutable Consensus*.

É então necessário escolher a qual protocolo tolerante a faltas bizantinas se irá aplicar as características do protocolo *Mutable Consensus*. Analisou-se com detalhe o protocolo *PBFT* por ser o pioneiro e, de alguma forma, todos se basearem neste e o protocolo *XPaxos* por necessitar de menos processos para tolerar t faltas, pelo que a escolha irá recair num destes. Essa escolha será feita de acordo com o que apresentar maior potencial de escalabilidade ao

aplicar-lhe as características do protocolo *Mutable Consensus*, em particular com a mutação *gossip*.

3

ANÁLISE DE ESCALABILIDADE

Nesta secção é feita uma análise do potencial de escalabilidade dos protocolos tolerantes a faltas bizantinas *PBFT* e *XPaxos*. Uma vez que o número de mensagens trocadas para atingir uma decisão é um dos principais fatores que afeta a escalabilidade [Vukolić (2015)], é dado um maior foco a esse aspeto.

O número de mensagens trocadas na chegada a uma decisão depende diretamente do número de participantes. Assim, os protocolos *XPaxos* e *PBFT* podem ser comparados, nesta métrica, em termos do número de faltas toleradas ou no número de participantes totais. Porém, em ambos os casos, o *XPaxos* necessita de menos participantes para atingir uma decisão. Para além disso, o protocolo *PBFT* apresenta um passo extra relativamente ao protocolo *XPaxos*, o que resulta num número muito maior de mensagens trocadas. Assim, na análise apresentada é considerada uma otimização ao protocolo *PBFT* que permite executar pedidos com apenas o primeiro passo, tal como acontece no *XPaxos*. Esta otimização, chamada *tentative execution*, não remove o segundo passo, uma vez que não é possível, mas permite corrê-lo em paralelo com as respostas aos clientes ou até acoplado ao primeiro passo do pedido seguinte.

Tendo em conta que se pretende aplicar ao protocolo a escolher as características do protocolo Mutable Consensus, o número de mensagens trocadas é analisado para cada mutação. Na mutação *early* cada processo envia as mensagens para todos os processos, pelo que quanto menor for o número de participantes menos mensagens são trocadas. Para a mutação *centralized* apenas o coordenador envia mensagens para todos os processos, enquanto que os restantes processos apenas enviam as suas mensagens ao coordenador. Assim, para esses processos o número de participantes não influencia o número de mensagens que enviam e recebem. Relativamente ao coordenandor, tal como acontece com a mutação *early*, quanto mais participantes existirem, mais mensagens são recebidas e enviadas por este. A mutação *ring* faz com que cada processo apenas receba mensagens de um processo e também envie a apenas um processo. No entanto, quanto maior for o número de participantes mais mensagens são necessárias para chegar a todos os processos. Para além disso, no protocolo *XPaxos*, enquanto a maioria que participa for correta, as mensagens são sempre entregues. Porém, no protocolo *PBFT*, podem existir mensagens a serem entregues a processos faltosos,

o que faz com que a troca de mensagens seja quebrada sendo necessária a ocorrência de retransmissões.

Assim, o protocolo *XPaxos* necessita de menos mensagens para atingir uma decisão em qualquer uma destas mutações, relativamente ao protocolo *PBFT*. No entanto, é a mutação *gossip* que permite atingir uma grande escalabilidade no protocolo Mutable Consensus. Nesta mutação, as mensagens são entregues a diferentes processos de forma probabilística, não sendo portanto trivial determinar o número de mensagens trocadas para atingir um acordo. Para além de ser necessário considerar o número de participantes, tal como nas mutações anteriores, existe ainda a questão de atingir uma maioria quando todos participam ou atingir todos quando apenas participa uma maioria. Por esta razão, realizaram-se simulações de forma a determinar o número de mensagens necessárias para atingir uma decisão em cada uma destas condições. Adicionalmente, começou-se por considerar um protocolo alternativo ao *XPaxos*, para o modelo *XFT*, em que todos os processos participam, de forma a compará-lo ao *XPaxos*, onde apenas participa uma maioria. Deu-se o nome *XpensivePaxos* a este protocolo em referência ao protocolo *Cheap Paxos* cuja comunicação se assemelha à do protocolo *XPaxos*.

3.1 XPAXOS OU XPENSIVEPAXOS

Protocolos para o modelo *XFT*, atingem o acordo quando uma maioria de processos tem confirmações de uma maioria de processos. Porém, para o protocolo *XPaxos* isto significa que todos os processos que participam têm de ter confirmações de todos os processos que participam, uma vez que só uma maioria de processos participa no acordo. Note-se ainda que num protocolo em que todos os processos participam, as maiorias que cada processo conhece não têm de ser necessariamente iguais.

Assim, cada simulação inicia com um coordenador a enviar a sua confirmação para processos que fazem parte de um conjunto de participantes. Os processos ao receberem confirmações, juntam-lhes as confirmações que já receberam e enviam-nas para processos desse mesmo conjunto. Uma vez que existe uma visão global de todos os processos, estes envios são realizados em iterações síncronas e, no final de cada iteração é verificado se existe um número suficiente de processos com esse mesmo número de confirmações.

O conjunto de participantes e o número de confirmações necessárias são definidos a partir do número de faltas t a tolerar e do nível de participação a considerar. Foram considerados os valores de 30, 60, 120, 240 e 480 como o número de faltas a tolerar, pelo que para os casos em que apenas participa uma maioria existem $t+1$ participantes e no caso em que todos os processos participam existem $2t+1$ participantes. Já o número de confirmações necessárias para atingir o acordo é $t+1$, independentemente do nível de participação.

Para além destes parâmetros, é ainda definido o valor de *fanout* que determina para quantos processos do conjunto de participantes cada mensagem (que contém as confirmações) é enviada. tendo sido considerados os valores de 2, 3 e 4. Em cada processo, os destinos são obtidos rotativamente de uma permutação aleatória do conjunto de participantes, de acordo com o valor de *fanout*.

Uma vez que estas simulações apenas se aplicam ao caso normal, não são assumidas faltas para o protocolo *XPaxos*. No entanto, para o protocolo *XpensivePaxos*, é necessário considerar a presença de processos faltosos. Assim, neste nível de participação, foram realizadas simulações para o melhor e para o pior caso, isto é, na ausência de faltas e na presença de t faltas (o máximo tolerável). Este pior caso implica que, do conjunto de participantes, apenas t+1 participam efetivamente na troca de mensagens, porém os restantes t processos continuam a ser destinos dessas mensagens.

Por cada configuração a simular efetuaram-se 100 repetições, pois o comportamento aleatório não permite tirar conclusões fiáveis com apenas uma execução. Em cada repetição são calculados o número de iterações necessário para atingir a decisão e o número total de mensagens trocadas. O número de iterações permite estimar a latência das decisões e através do número total de mensagens pode-se calcular o número médio de mensagens por processo, dividindo esse valor pelo número de participantes.

3.1.1 *Algoritmo de gossip base*

Os gráficos da Figura 1 apresentam o número médio de mensagens por processo e o número de iterações necessárias para atingir a decisão de acordo com o que foi descrito. Note-se que para o melhor caso do protocolo *XpensivePaxos*, ainda que o número total de mensagens possa ser superior comparativamente à participação de uma maioria, o número de mensagens por processo pode ser inferior já que o número de participantes é também maior. Já para o pior caso, uma vez que apenas os processos que não são faltosos é que contribuem, apenas estes são contabilizados como participantes, sendo assim o número de participantes o mesmo do protocolo *XPaxos*. São apresentados, em todos os gráficos, estes valores para o protocolo *XPaxos* original, que consistem em t+1 mensagens por processo e duas iterações, independentemente do número de participantes.

Relativamente ao número de mensagens (o maior indicador de escalabilidade) a participação de todos os processos apresenta claramente melhores resultados do que a participação de uma maioria, quer para o melhor caso, quer para o pior caso. Porém, no pior caso, uma vez que mensagens podem ser enviadas para processos faltosos (com probabilidade de 50%) pode nunca ser atingida qualquer decisão. Isto, aliás, acontece frequentemente com 98% dos casos para *fanout* 2 a não terminarem, 25% para fanout 3 e 10% para fanout 4. Assim,

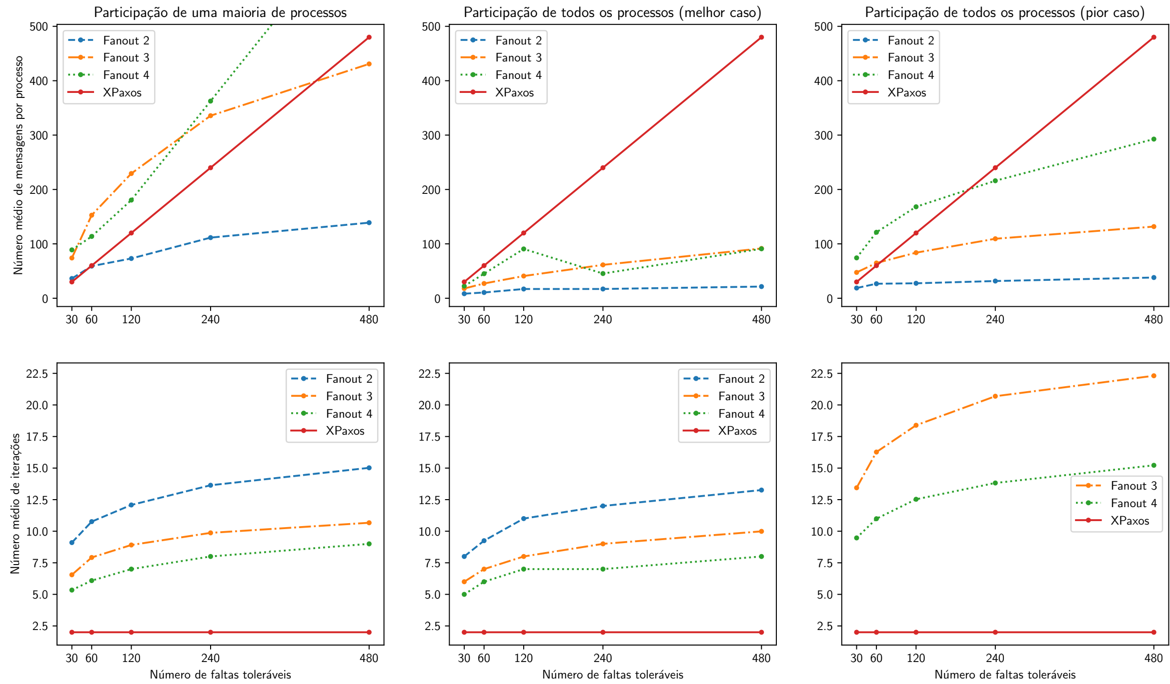


Figura 1: Comparação entre protocolos no modelo XFT

os resultados para o pior caso, apenas representam as repetições em que foram atingidas decisões, pelo que não são indicativos.

Podem ainda ser destacados outros comportamentos inesperados, relativamente ao número de mensagens. Por vezes, com *fanout* 4 consegue-se atingir o acordo com menos mensagens do que com *fanout* 3, o que se deve à diferença no número de iterações necessárias para alcançar um acordo entre estes valores de *fanout*. Também, pode ser observado que o aumento do número de participantes nem sempre resulta num aumento no número de mensagens por processo. Isto deve-se, novamente, ao número de iterações que, neste caso, não aumenta o que faz com que o número total de mensagens também não aumente. Assim, este dimiuição apenas acontece por o número total de mensagens ser dividido por mais participantes.

Olhando ao número de iterações necessário para atingir um acordo, é também fácil constatar que o melhor caso para o protocolo em que todos os processos participam necessita de menos iterações do que o protocolo em que apenas participa uma maioria. Importa ainda referir em relação ao pior caso na participação de todos os processos que, para além dos resultados não serem significativos pela razão anteriormente mencionada, os resultados para *fanout* 2 não são apresentados por saírem dos intervalos que permitem uma observação clara dos restantes resultados.

A principal razão para que o número de mensagens por processo seja significativamente maior na participação de uma maioria relativamente à participação de todos, deve-se a que

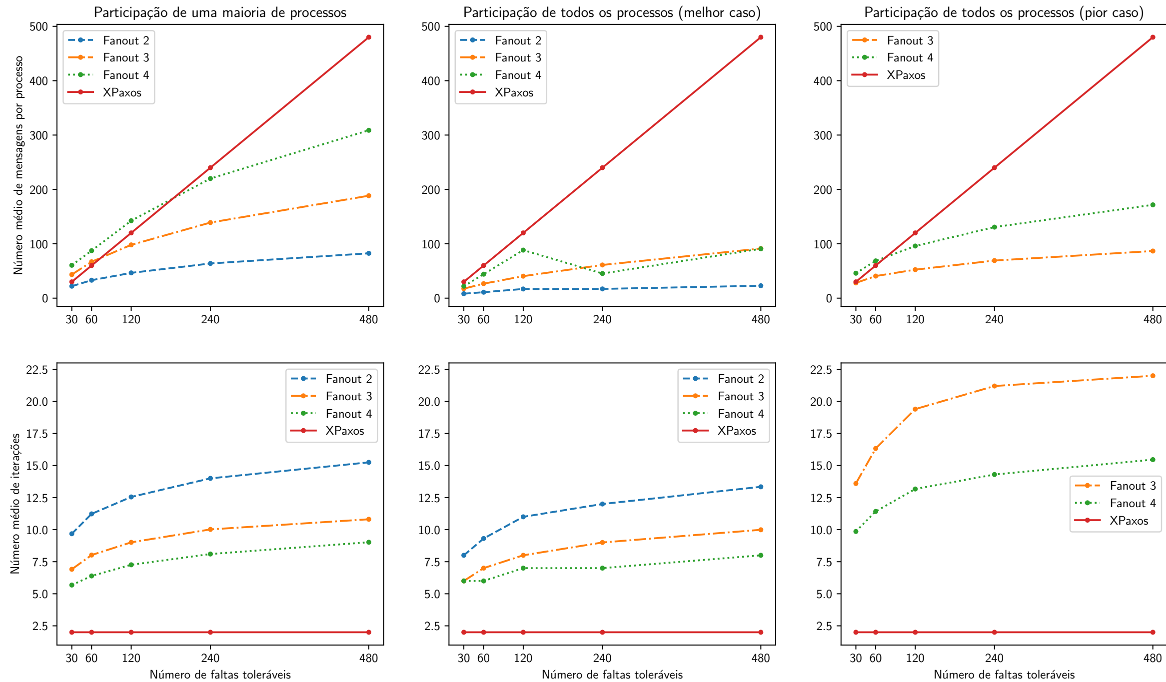


Figura 2: Comparação entre protocolos no modelo XFT com envios condicionais

para a maioria é necessário que todos os processos que participam tenham a confirmação de todos, o que implica que nas últimas iterações sejam trocadas imensas mensagens que não acrescentam confirmações aos destinatários mas que resultam no envio de ainda mais mensagens. Assim, aplicou-se uma característica do protocolo *Mutable Consensus*, em que apenas são enviadas mensagens se a mensagem recebida contiver novas confirmações, a todas as configurações propostas.

3.1.2 *Envios condicionais*

A Figura 2 contém os resultados obtidos para a modificação mencionada. Na participação de apenas uma maioria de processos verifica-se uma grande melhoria por comparação com os resultados anteriores. Porém esta melhoria acontece com o custo de passar a existir a hipótese de não se atingir decisões, nomeadamente de 30% nas repetições para *fanout* 2 e de 2% nas de *fanout* 3. Também em relação ao pior caso na participação de todos os processos se observou um aumento nas percentagens de não alcançar decisões. Para *fanout* 3 essa percentagem passou para 95% enquanto que para *fanout* 4 subiu para 35%. Adicionalmente, não foi possível completar qualquer repetição para *fanout* 2, sendo essa a razão por este valor não estar representado nos gráficos.

De forma a remover a possibilidade de não serem atingidas decisões, é necessário adicionar a retransmissão periódica de mensagens, tal como acontece no protocolo *Mutable Consensus*.

3.1.3 *Retransmissões*

Uma vez que nestas simulações não existe uma noção temporal, o período de retransmissão é definido em número de iterações. As retransmissões devem ocorrer relativamente à última mensagem enviada, sendo transmitida para os próximos *fanout* destinos, tal como acontece na ausência de retransmissões. Assim, sempre que uma nova mensagem for enviada o período deve ser redefinido, mesmo que isso implique reiniciar o período. É então necessário definir quantas iterações devem ser consideradas para que se justifique efetuar uma retransmissão, tendo em conta que quanto maior for o valor do período mais iterações serão necessárias para atingir uma decisão e vice-versa.

No protocolo em que apenas participa uma maioria de processos, não atingir uma decisão deve-se a serem recebidas mensagens que não contêm novas confirmações, resultando em não serem enviadas novas mensagens. Isto acontece principalmente nas últimas iterações, momento em que se encontram muitas mensagens em trânsito. Assim, se nessa altura um processo não transmitir mensagens durante uma ou duas iterações, por exemplo, significa que não recebeu mensagens com novas confirmações durante essas iterações, o que indica que pode-se justificar um período de retransmissão nesta ordem. Porém, nas iterações iniciais, não enviar mensagens durante uma ou duas iterações, pode ser normal, já que o número de mensagens trocadas é muito pequeno. Ainda assim, retransmitir com um período curto nas iterações iniciais pode permitir reduzir o número de iterações necessárias para atingir um acordo, com apenas um pequeno aumento no número de mensagens.

Relativamente à participação de todos os processos o risco de não atingir decisões aparece na presença de faltas e aumenta com o número de processos faltosos. Isto deve-se a mensagens serem enviadas para esses processos, o que quebra a propagação das mesmas. É nas iterações iniciais que existe uma maior suscetibilidade de todas as mensagens se perderem, já que o número de mensagens em trânsito é muito reduzido, pelo que se justifica um período de retransmissão pequeno. No entanto, este período de retransmissão deve-se manter em todas as iterações, caso contrário, o número de mensagens em trânsito irá diminuindo. É também necessário incluir as retransmissões nas simulações do melhor caso pois, num ambiente real, não é possível determinar se se está na presença de faltas ou não. Apesar de, neste caso, um período de retransmissão baixo não ser necessário, pode ser útil para as iterações iniciais, de forma a reduzir o número de iterações necessárias para atingir um acordo.

De acordo com o mencionado, optou-se então por realizar simulações com um período de retransmissão de duas iterações, estando os resultados obtidos apresentados na Figura 3. O número médio de mensagens por processos é claramente mais pequeno, na participação de todos os processos, quer para o melhor, quer para o pior caso, por comparação com a participação de uma maioria de processos. Por sua vez, o número médio de iterações para

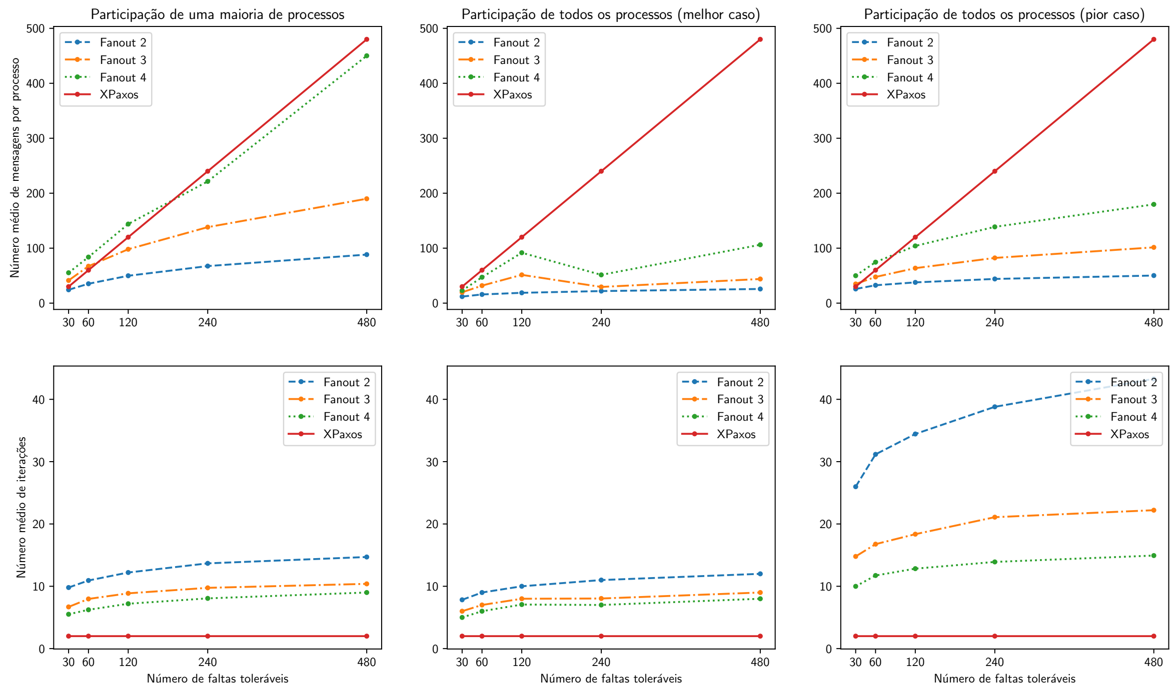


Figura 3: Comparação entre protocolos no modelo XFT com retransmissões

atingir uma decisão é mais pequeno no melhor caso da participação de todos os processos, sendo porém maior no pior caso, relativamente à participação de uma maioria de processos.

3.1.4 *Discussão*

A mutação *gossip* no protocolo *XpensivePaxos* apresenta, globalmente, maior potencial de escalabilidade do que a aplicação dessa mutação ao protocolo *XPaxos*, em que apenas participa uma maioria de processos. No entanto, este protocolo teria de incluir uma componente síncrona no caso normal, tal como acontece nas mudanças de vista do protocolo *XPaxos*, para garantir a correção do mesmo. Esta componente serviria para garantir que todos os processos corretos receberiam as confirmações de todos os processos corretos. Aplicar esta sincronia a uma difusão epidémica implicaria definir um número de iterações mínimo, para cada decisão, em que mensagens eram trocadas, o que resultava num aumento do número de mensagens para atingir as decisões. Para além disso, nunca se conseguiria garantir que ao fim dessas iterações os processos corretos teriam efetivamente recebido as confirmações de todos os processos corretos. A correção do protocolo tornar-se-ia, assim, probabilística, de tal forma que quanto maior o número de iterações, maior seria a probabilidade de ser correto.

Tendo isto em conta, o protocolo *PBFT* é comparado com os resultados obtidos para o protocolo *XPaxos*.

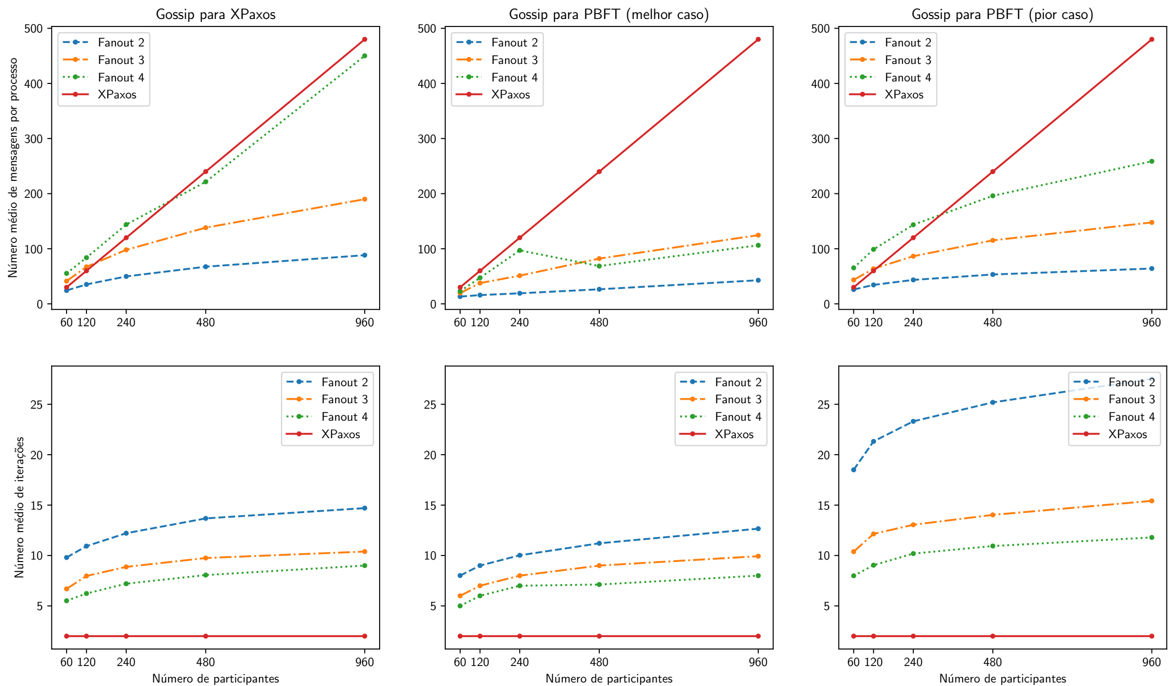


Figura 4: Comparação entre XPaxos e PBFT para a mesma população

3.2 XPAXOS OU PBFT

O facto de anteriormente se ter obtido melhores resultados na participação de todos os processos, motiva a realização de simulações para o protocolo *PBFT* uma vez que também neste protocolo todos os processos participam na chegada a cada decisão. Por serem necessários mais processos para tolerar o mesmo número de faltas, relativamente ao protocolo *XPaxos*, são feitas comparações para o mesmo número de participantes e também para o mesmo número de faltas toleráveis. Nesta secção avança-se diretamente para simulações em que apenas se envia mensagens caso a mensagem recebida contenha novas confirmações e em que se efetua retransmissões com período 2.

3.2.1 *Mesma população*

Considerar a mesma população para os protocolos *XPaxos* e *PBFT*, implica tolerar menos faltas no último. Adicionalmente, é necessário considerar dois terços de processos para se atingir um acordo, em oposição a um meio para o protocolo *XPaxos*. Relativamente ao pior caso do protocolo *PBFT*, isto é, na presença de t faltas, a probabilidade de uma mensagem ser enviada para um destes processos baixa para 33%, relativamente ao pior caso do protocolo para o modelo *XFT* em que todos os processos participam, analisado na secção anterior.

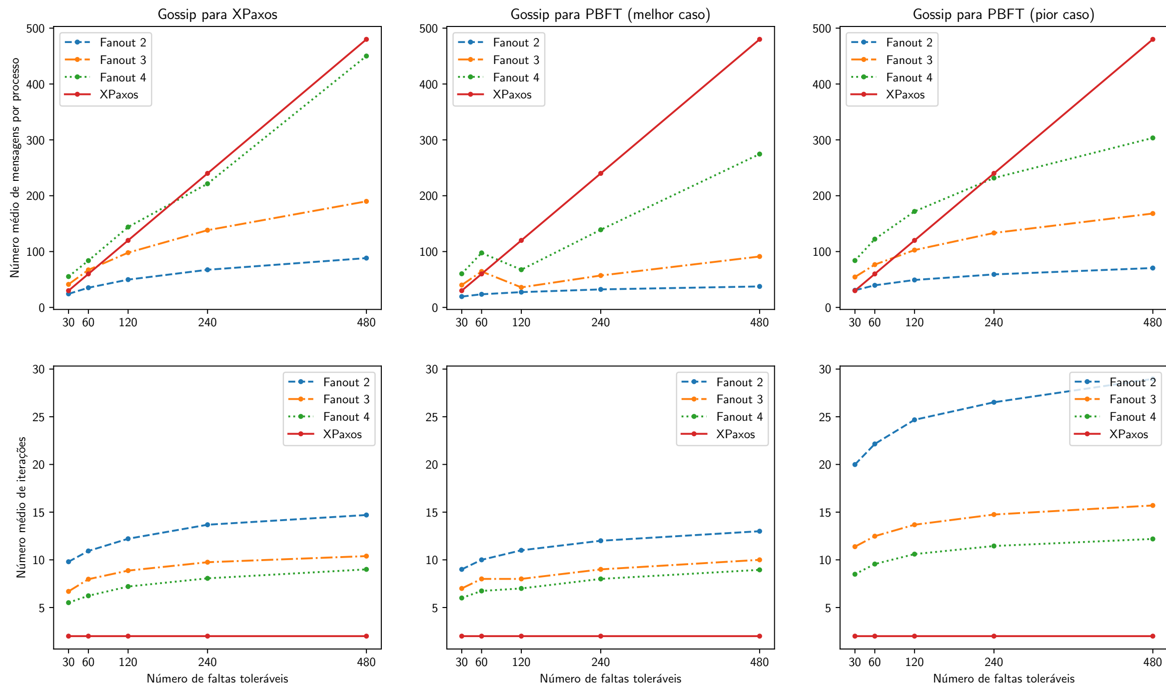


Figura 5: Comparação entre XPaxos e PBFT para a mesma tolerância

Assim, a Figura 4 apresenta o número médio de mensagens por processo e o número médio de iterações para chegar a um acordo, para os protocolos *XPaxos* e *PBFT*, com a distinção entre melhor e pior caso para este último. Note-se que o eixo horizontal nesta figura representa a população e não a tolerância como acontece nas restantes figuras.

Relativamente ao número de mensagens por processo, o melhor caso do protocolo *PBFT* revela melhores resultados por comparação com o protocolo *XPaxos*. Por outro lado, para o pior caso, à medida que o número de processos aumenta, a diferença no número de mensagens entre o *PBFT* e o *XPaxos* passa de ser nula para aumentar em favor do *PBFT*. Já em relação ao número de iterações, para o melhor caso do protocolo *PBFT* o seu valor é menor do que para o protocolo *XPaxos*, mas maior no pior caso.

3.2.2 *Mesma tolerância a faltas*

Ao definir a mesma tolerância para os protocolos *XPaxos* e *PBFT* importa salientar que, neste último, a população é 1.5 vezes maior e são necessários o dobro dos processos para atingir um acordo, relativamente ao protocolo *XPaxos*.

A Figura 5 apresenta então os resultados obtidos para as simulações nestas configurações. O número de mensagens por processo, para o melhor caso do protocolo *PBFT*, continua a ser mais baixo do que para o protocolo *XPaxos*, enquanto que para o pior caso, é maior quando o número de processos é menor, passando a ser menor para um número de processos grande.

Tal como anteriormente, o número de iterações é melhor para o melhor caso do protocolo *PBFT*, mas menor para o pior caso, relativamente ao protocolo *XPaxos*.

3.3 DISCUSSÃO

De acordo com todos os resultados obtidos pode-se concluir que, independentemente de ser considerada a mesma população ou a mesma tolerância, a aplicação da mutação *gossip* ao protocolo *PBFT* oferece maior escalabilidade, para o melhor caso, por comparação com a aplicação dessa mutação ao protocolo *XPaxos*. Adicionalmente, para o pior caso do protocolo *PBFT*, apesar de o número de iterações necessárias para atingir o acordo ser sempre maior, relativamente ao protocolo *XPaxos*, o número de mensagens trocadas é menor, por processo, para grandes populações, o que permite aumentar o número de participantes para valores superiores, em relação ao *XPaxos*.

Um aspeto não considerado nestas simulações, mas que também deve ter impacto na escolha do protocolo, é o das mudanças de vista. No protocolo *XPaxos*, todos os processos que fazem parte da maioria têm de ser corretos, pelo que, ocorrendo uma falha em algum desses processos, torna-se necessário efetuar uma mudança de vista. Isto resulta em mudanças de vista mais frequentes por comparação com o protocolo *PBFT*, já que neste apenas se efetua mudanças de vista quando o coordenador falha. As mudanças de vista do protocolo *XPaxos* são ainda mais demoradas, já que é necessário encontrar uma nova maioria correta, por oposição de encontrar um novo coordenador correto.

Assim, tendo em conta todos os fatores expostos, optou-se por tornar mutável o protocolo *PBFT*.

4

MUTABLE BFT

Nesta secção propõe-se o protocolo *Mutable BFT* baseado no protocolo *PBFT* com as características do protocolo *Mutable Consensus*. A principal diferença em relação ao protocolo *PBFT* é a de os processos, para além de enviarem as mensagens que criam, também enviarem as mensagens que aceitam de outros processos. Com isto torna-se possível definir diferentes padrões de comunicação já que as mensagens de um processo podem ser enviadas por outro processo. Para além disso, as mensagens são retransmitidas periodicamente permitindo assim que os padrões de comunicação não coloquem em causa as garantias de disponibilidade do protocolo.

A Figura 6 apresenta os comportamentos exibidos por um processo no protocolo *Mutable BFT* em diferentes situações, para uma população de 4 processos. Na situação (a), ao receber uma mensagem PrePrepare, é criada uma mensagem Prepare que é enviada, juntamente com a mensagem recebida, para todos os processos. Na situação (b), é recebida uma mensagem Prepare do processo 2 (para além da mensagem PrePrepare já conhecida) o que resulta na criação de uma mensagem Commit, sendo todas as mensagens enviadas para todos os processos. A situação (c) apresenta uma retransmissão das mesmas mensagens enviadas na situação (b), uma vez que o período de retransmissão, δ, expirou sem que tenham ocorrido novas transmissões. Na situação (d) não é recebida nenhuma nova mensagem pelo que não ocorre qualquer envio. Por fim, na situação (e) são recebidas mensagens que, juntamente com as mensagens já conhecidas, permitem atingir o estado *committed*.

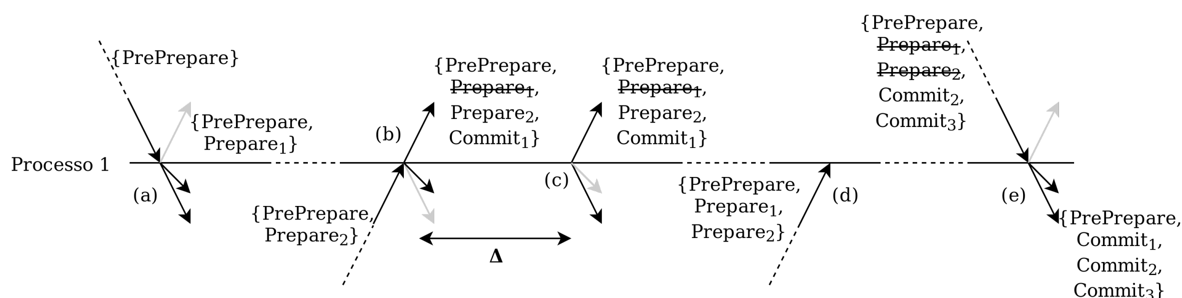


Figura 6: Algumas situações e respetivos comportamentos admitidos pelo protocolo Mutable BFT

Pode-se constatar que no envio de mensagens as mensagens apenas são transmitidas para alguns dos destinos (através da tonalidade das setas), mostrando assim a possibilidade de definir diferentes padrões de comunicação. Para além disso, algumas mensagens encontram-se rasuradas em referência à otimização que permite omitir o envio de mensagens Prepare.

No que resta deste capítulo começa-se por propor uma alteração à condição *prepared* de forma a que também possa ser avaliada com mensagens Commit, com vista a omitir o envio de mensagens *Prepare*. Apresenta-se ainda a especificação das características do protocolo *Mutable Consensus*, com base na especificação do protocolo *PBFT* [Castro et al. (1999a)], nomeadamente o envio de mensagens que são aceites, os canais *stubborn* e as mutações.

4.1 ALTERAÇÃO À CONDIÇÃO PREPARED

A condição *prepared* verifica se para um pedido são conhecidas uma mensagem Pre-Prepare e um conjunto com pelo menos 2t mensagens Prepare de processos diferentes, para a mesma vista e número de sequência. A alteração a efetuar à condição *prepared* para considerar mensagens Commit consiste em permitir conhecer um conjunto de pelo menos 2t mensagens sejam elas de Prepare ou Commit. No entanto, neste conjunto não pode estar incluída uma mensagem Commit do coordenador nem podem coexistir mensagens Prepare e Commit do mesmo processo.

$$prepared(m,v,n,M) \equiv \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}} \in M \land \exists R : (|R| \geq 2t \land primary(v) \notin R \land \forall k \in R : (\langle \text{Prepare}, v, n, D(m), k\rangle_{\sigma_k} \in M \lor \langle \text{Commit}, v, n, D(m), k\rangle_{\sigma_k} \in M))$$

Condição prepared alterada

De forma a determinar as implicações de alterar a condição *prepared* para também considerar mensagens Commit, procedeu-se a uma análise da prova do protocolo *PBFT* [Castro et al. (1999a)], nomeadamente dos invariantes em que a condição *prepared* é provada.

4.1.1 *Viabilidade da alteração*

O invariante 1.4. indica que em quaisquer dois processos corretos, para os mesmos valores de vista e número de sequência, a condição *prepared* de cada processo apenas é verdadeira para o mesmo pedido. A prova é feita por contradição, assumindo que a condição *prepared* é verdadeira em dois processos corretos, para os mesmos valores de vista e número de sequência, mas pedidos diferentes. Para isso acontecer, é necessário que pelo menos t+1 processos enviem mensagens com os mesmos valores de vista e número de sequência mas pedidos diferentes para os dois processos. Dado que apenas se tolera, no máximo, t faltas, terá de existir pelo menos um processo correto que envia essas mensagens contraditórias.

É provado que não é possível que uma réplica correta envie mensagens Pre-Prepare ou Prepare contraditórias, já que não as cria. Resta então provar que um processo correto também não envia mensagens Commit contraditórias.

$$\forall i,j \in \mathcal{R}, n,v \in \mathbf{N}, m,m' \in \mathcal{M} : ((\neg faulty_i \wedge \neg faulty_j \wedge n\text{-}faulty \leq f) \Rightarrow (prepared(m,v,n,i) \wedge prepared(m',v,n,j) \Rightarrow D(m) = D(m')))$$

Invariante 1.4.

Fazer essa prova não está no âmbito deste trabalho, ainda assim, faz-se um exercício que abre caminho para essa prova. Processos corretos enviam mensagens Commit quando a condição *prepared* é verdadeira. Um dos requisitos para que a condição *prepared* seja verdadeira num processo correto é que esse processo conheça uma mensagem Pre-Prepare respetiva. Uma vez que processos corretos não criam nem aceitam mensagens Pre-Prepare contraditórias, então também não podem enviar mensagens Commit contraditórias.

O invariante 1.5. constata uma propriedade semelhante, porém apenas para o conjunto de mensagens Pre-Prepare e Prepare incluídos numa mensagem New-View. É referido que a prova é semelhante à do invariante 1.4. pelo que pode-se assumir que esta alteração se aplica a esta prova da mesma forma.

$$\forall i \in \mathcal{R} : ((\neg faulty_i \wedge n\text{-}faulty \leq f) \Rightarrow (\forall \langle \text{New-View}, v, V, O, N \rangle_{\sigma_k} \in in_i, n, v' \in \mathbf{N} : (prepared(m,v',n,merge\text{-}P(V)) \wedge prepared(m',v',n,merge\text{-}P(V)) \Rightarrow D(m) = D(m'))))$$

Invariante 1.5.

Por fim, o invariante 1.8. assume que o conjunto de mensagens Pre-Prepare e Prepare que são inseridos numa mensagem View-Change têm de ser os mesmos em qualquer momento em que se reconstrua essa mensagem ao apenas considerar as mensagens com vista inferior à vista dessa mensagem. Este conjunto é obtido a partir da condição *last-prepared* que, por sua vez, verifica a condição *prepared*. Assim, para que o invariante seja válido, mensagens que influenciem a condição *prepared* não podem ser aceites em vistas futuras relativamente à vista a que se referem. Isto é verdadeiro para as mensagens Pre-Prepare e Prepare, no entanto, de acordo com o protocolo original, mensagens Commit podem ser aceites em vistas futuras. É então necessário alterar a condição em que mensagens Commit são aceites, passando apenas a aceitá-las se elas se referirem à vista atual. A condição original permite aceitar mensagens Commit se estas se referirem a vistas passadas, porém retirar essa possibilidade não afeta a correção pois ela não é usada para para provar qualquer invariante.

$\forall m, v, n, P : (\langle \text{View-Change}, v, P, i \rangle_{\sigma_i} \in in_i \Rightarrow \forall v' < v : (\textit{last-prepared-b}(m, v', n, i, v) \Leftrightarrow \textit{last-prepared}(m, v', n, P)))$

Em que *last-prepared-b* é definido por:

$\textit{last-prepared-b}(m, v, n, i, b) \equiv v < b \wedge \textit{prepared}(m, v, n, in_i) \wedge \nexists m', v' : ((\textit{prepared}(m', v', n, in_i) \wedge v < v' < b) \vee (\textit{prepared}(m', v, n, in_i) \wedge m \neq m'))$

Invariante 1.8.

Note-se que a aceitação de mensagens Commit referentes a vistas passadas pode ser mantida desde que se identifique a vista em que a mensagem é aceite. Desta forma, ao verificar a condição *prepared*, apenas se pode considerar as mensagens Commit que foram aceites na vista a que se referem. Para fazer o mínimo de alterações ao protocolo original, omitiu-se esta otimização na especificação.

Para além das alterações referidas é ainda necessário alterar o conjunto de de mensagens que são enviadas numa mensagem View-Change, de forma a incluir as mensagens Commit. Mesmo com esta alteração, neste conjunto de mensagens continua a ser suficiente a existência de apenas 2t+1 mensagens, em que uma delas terá de obrigatoriamente ser um Pre-Prepare e as restantes 2t poderão ser quer mensagens Prepare, quer mensagens Commit, com as mesmas restrições da condição *prepared*.

4.2 ESPECIFICAÇÃO DO PROTOCOLO

Tal como referido anteriormente, é necessário passar a enviar as mensagens que são aceites, para além das mensagens que são criadas pelo próprio processo. Em primeiro lugar, é eliminada a ação Send que envia mensagens que se encontrem no *buffer* de saída, removendo-as posteriormente. Mais à frente serão criadas novas ações para enviar estas mensagens, porém com condições e comportamentos diferentes.

As mensagens que um processo cria são inseridas no seu *buffer* de saída durante a criação, porém é necessário criar um mecanismo que também insira as mensagens que são aceites, tendo, para isso, sido definidas ações Relay. Dado que as mensagens que se encontram no *buffer* de saída serão retransmitidas continuamente é necessário definir pré-condições para inserir mensagens, para que não sejam transmitidas mensagens desnecessárias. Definiu-se ainda ações Drop que removem mensagens do *buffer* de saída quando se tornam obsoletas.

As pré-condições foram então construídas de acordo com os momentos em que mensagens se devem encontrar no *buffer* de saída, pelo que as pré-condições das ações Relay e Drop que correspondem são o inverso uma da outra. Existem três pares de ações Relay-Drop: para mensagens do caso normal (Pre-Prepare, Prepare e Commit), para mensagens de View-Change e para mensagens New-View. No caso normal apenas se transmite mensagens cuja vista seja igual à vista atual. Mensagens Prepare e Commit só se encontram no *buffer* de

saída se o Pre-Prepare correspondente for conhecido, uma vez que as mensagens Prepare e Commit podem ser aceites sem que exista uma mensagem Pre-Prepare correspondente. Adicionalmente, mensagens Commit substituem mensagens Prepare do mesmo processo. Por fim, estas mensagens apenas são transmitidas caso o pedido ainda não tenham sido executado. Mensagens View-Change apenas são transmitidas enquanto a própria mensagem View-Change para a vista atual se encontra no *buffer* de saída. Adicionalmente, apenas são incluídas mensagens View-Change cuja vista seja igual ou superior à vista atual. A própria mensagem View-Change é removida quando a condição *has-new-view* for verdadeira ou se a vista da mensagem for inferior à vista atual. As mensagens de New-View apenas se encontram no *buffer* de saída enquanto a vista da mensagem for a vista atual.

Relay-Pre-Prepare$(m, v, n)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \in in_i \wedge \langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \notin out_i \wedge n > last\text{-}exec_i \wedge in\text{-}v(v, i)$

Eff:

$out_i := out_i \cup \{\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}}\}$

Drop-Pre-Prepare$(m, v, n)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \in out_i \wedge (n \leq last\text{-}exec_i \vee \neg in\text{-}v(v, i))$

Eff:

$out_i := out_i \setminus \{\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}}\}$

Relay-Prepare$(m, v, n, j)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \in out_i \wedge \langle \text{Prepare}, v, n, D(m), j \rangle_{\sigma_j} \in in_i \wedge \langle \text{Prepare}, v, n, D(m), j \rangle_{\sigma_j} \notin out_i \wedge \langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j} \notin out_i \wedge n > last\text{-}exec_i \wedge in\text{-}v(v, i)$

Eff:

$out_i := out_i \cup \{\langle \text{Prepare}, v, n, D(m), j \rangle_{\sigma_j}\}$

Drop-Prepare$(m, v, n, j)_i$

Pre:

$\langle \text{Prepare}, v, n, D(m), j \rangle_{\sigma_j} \in out_i \wedge (\langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j} \in out_i \vee n \leq last\text{-}exec_i \vee \neg in\text{-}v(v, i))$

Eff:

$out_i := out_i \setminus \{\langle \text{Prepare}, v, n, D(m), j \rangle_{\sigma_j}\}$

Relay-Commit$(m, v, n, j)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \in out_i \wedge \langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j} \in in_i \wedge \langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j} \notin out_i \wedge n > last\text{-}exec_i \wedge in\text{-}v(v, i)$

Eff:

$out_i := out_i \cup \{\langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j}\}$

Drop-Commit$(m, v, n, j)_i$

Pre:

$\langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j} \in out_i \wedge (n \leq last\text{-}exec_i \vee \neg in\text{-}v(v, i))$

Eff:

$out_i := out_i \setminus \{\langle \text{Commit}, v, n, D(m), j \rangle_{\sigma_j}\}$

Ações Relay e Drop para mensagens do caso normal

Relay-View-Change$(v, j)_i$

Pre:

$\exists P : \langle \text{View-Change}, view_i, P, i\rangle_{\sigma_i} \in out_i \land \exists P' : (\langle \text{View-Change}, v, P', j\rangle_{\sigma_j} \in in_i \land \langle \text{View-Change}, v, P', j\rangle_{\sigma_j} \notin out_i) \land v \geq view_i$

Eff:

$out_i := out_i \cup \{\langle \text{View-Change}, v, P, j\rangle_{\sigma_j}\}$

Drop-View-Change$(v, j)_i$

Pre:

$\nexists P : \langle \text{View-Change}, view_i, P, i\rangle_{\sigma_i} \in out_i \lor (\exists P' : \langle \text{View-Change}, v, P', j\rangle_{\sigma_j} \in out_i \land (v < view_i \lor \textit{has-new-view}(v, i)))$

Eff:

$out_i := out_i \setminus \{\langle \text{View-Change}, v, P, j\rangle_{\sigma_j}\}$

Ações Relay e Drop para as mensagens View-Change

Relay-New-View$(v)_i$

Pre:

$\exists X, O, N, j : (\langle \text{New-View}, v, X, O, N\rangle_{\sigma_j} \in in_i \land \langle \text{New-View}, v, X, O, N\rangle_{\sigma_j} \notin out_i) \land \textit{in-v}(v, i)$

Eff:

$out_i := out_i \cup \{\langle \text{New-View}, v, X, O, N\rangle_{\sigma_j}\}$

Drop-New-View$(v)_i$

Pre:

$\exists X, O, N, j : \langle \text{New-View}, v, X, O, N\rangle_{\sigma_j} \in out_i \land v < view_i$

Eff:

$out_i := out_i \setminus \{\langle \text{New-View}, v, X, O, N\rangle_{\sigma_j}\}$

Ações Relay e Drop para as mensagens New-View

4.2.1 *Canais Stubborn*

Para enviar as mensagens que se encontram no *buffer* de saída, são criados canais de comunicação *stubborn*. Tal como acontece com as ações Relay e Drop, existem três tipos de canais *stubborn*: para mensagens do caso normal (Pre-Prepare, Prepare e Commit), para as mensagens View-Change e para mensagens New-View. Nas mensagens do caso normal existem canais distintos para cada tuplo que é definido por uma vista, um número de sequência e um pedido. Adicionalmente, cada tipo de canal é dividido em múltiplos canais, um para cada destino.

$t_i : \mathcal{CH} \times \mathcal{R} \rightarrow \mathbf{N}$, initially $\forall ch \in \mathcal{CH}, j \in \mathcal{R} : t_i(ch, j) = \infty$
$B_i : \mathcal{CH} \rightarrow 2^{\mathcal{M}}$, initially $\forall ch \in \mathcal{CH} : B_i(ch) = \{\}$

Novos estados para os canais *stubborn*

Especificar os canais *stubborn* envolveu criar ações Pre-Send e Send, existindo um para cada tipo de canal. Froam ainda criados novos estados, nomeadamente, o estado *timeout* que indica, para cada canal, o momento em que uma retransmissão deve acontecer, e o estado B que contém o conjunto de mensagens mais recentemente considerado para um dado canal global, isto é, o canal sem ter em conta a divisão para cada destino.

As ações Pre-Send marcam os estados *timeout* de todos os destinos para o envio de um conjunto de mensagens num canal e guardam o conjunto como o mais recente desse canal global. Por sua vez, as ações Send, para além de enviarem o conjunto de mensagens para um determinado destino, voltam a definir o *timeout* para esse destino. Os estados *timeout* consistem num incremento ao valor do relógio local com um valor que é múltiplo de um delta pré-definido, em que o fator de multiplicidade é determinado pela mutação.

As pré-condições das ações Pre-Send podem ser divididas em duas partes. Um dos requisitos tem a função de definir o conjunto de mensagens a ser transmitido de forma a que todas as mensagens que se encontram no *buffer* de saída, para o canal global a considerar, sejam incluídas. O segundo requisito serve para ativar a ação, o que acontece quando o conjunto de mensagens definido sofre alguma alteração em relação ao conjunto mais recente do canal global. Essas alterações consistem em adicionar ou remover mensagens do *buffer* de saída, por parte das ações Relay e Drop.

As pré-condições das ações Send também definem o conjunto de mensagens a enviar como sendo o conjunto mais recente, para que qualquer conjunto de mensagens transmitido passe pela ação Pre-Send antes de passar pela ação Send. Adicionalmente, são ativadas quando o valor de *timeout* para o destino considerado expira.

Uma vez que os diferentes tipos de canais são independentes, pode-se definir diferentes deltas para cada um deles.

Pre-Send$(M)_i$

Pre:

$\exists m, v, n : (\textit{prepared-certificate}(m, v, n, out_i) = M \wedge B_i(\langle m, v, n\rangle) \neq M$

Eff:

for each $j \in (\mathcal{R} \setminus \{i\})$ do

$t_i(\langle m, v, n\rangle, j) := clock_i + mutation_0(\langle m, v, n\rangle, M, i, j) \times \Delta$

$B_i(\langle m, v, n\rangle) := M$

$\textit{prepared-certificate}(m, v, n, M) \equiv \{c = \langle \text{Commit}, v, n, D(m), k\rangle_{\sigma_k} | c \in M\} \cup \{p = \langle \text{Prepare}, v, n, D(m), k\rangle_{\sigma_k} | p \in M\} \cup \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}}\}$

Send$(M, j)_i$ $(j \neq i)$

Pre:

$\exists m, v, n : (B_i(\langle m, v, n\rangle) = M \wedge t_i(\langle m, v, n\rangle, j) \leq clock_i) \wedge M \neq \{\}$

Eff:

$t_i(\langle m, v, n\rangle, j) := clock_i + mutation(\langle m, v, n\rangle, M, i, j) \times \Delta$

Ações Pre-Send e Send para os canais das mensagens do caso normal

Pre-Send$(M)_i$

Pre:

$M = \{m = \langle \text{View-Change}, v, P, k\rangle_{\sigma_k} | m \in out_i\} \wedge B_i(\text{View-Change}) \neq M$

Eff:

for each $j \in (\mathcal{R} \setminus \{i\})$ do

$t_i(\text{View-Change}, j) := clock_i + mutation_0(\text{View-Change}, M, i, j) \times \Delta$

$B_i(\text{View-Change}) := M$

Send$(M, j)_i$ $(j \neq i)$

Pre:

$B_i(\text{View-Change}, j) = M \wedge t_i(\text{View-Change}, j) \leq clock_i \wedge M \neq \{\}$

Eff:

$t_i(\text{View-Change}, j) := clock_i + mutation(\text{View-Change}, M, i, j) \times \Delta$

Ações Pre-Send e Send para os canais das mensagens View-Change

$\text{Pre-Send}(M)_i \ (j \neq i)$
Pre:
$M = \{m = \langle \text{New-View}, v, X, O, N \rangle_{\sigma_k} | m \in out_i\} \land B_i(\text{New-View}) \neq M$
Eff:
for each $j \in (\mathcal{R} \setminus \{i\})$ do
$t_i(\text{New-View}, j) := clock_i + mutation_0(\text{New-View}, M, i, j) \times \Delta$
$B_i(\text{New-View}) := M$

$\text{Send}(M, j)_i \ (j \neq i)$
Pre:
$B_i(\text{New-View}) = M \land t_i(\text{New-View}, j) \leq clock_i \land M \neq \{\}$
Eff:
$t_i(\text{New-View}, j) := clock_i + mutation(\text{New-View}, M, i, j) \times \Delta$

Ações Pre-Send e Send para os canais das mensagens New-View

Importa referir que estas ações, quando ativadas, podem ser executadas em ordens arbitrárias. Por exemplo, ao ser criada uma mensagem Prepare, na receção de uma mensagem Pre-Prepare, ela é adicionada ao *buffer* de saída. Porém, de seguida pode ser executada a ação Relay, que adiciona o Pre-Prepare aceite, ou a ação Pre-Send, uma vez que o conjunto de mensagens mudou. Adicionalmente, caso seja executada a ação *Pre-Send*, não é garantido que, de seguida, seja executada a ação Send, uma vez que a ação Relay continua ativada. Assim, apesar de algumas ordens de execução não serem ótimas, não se fez alterações à especificação para não a tornar complexa. Porém, numa implementação deste protocolo, deve-se ter em conta estas possibilidades de forma a serem escolhidos os comportamentos desejáveis.

4.2.2 *Mutações*

Os valores de *timeout* utilizados nas ações Pre-Send e Send são definidos por padrões de comunicação, aqui chamadas mutações. A especificação de uma mutação requer a implementação das primitivas $mutation_0$ e *mutation*. Estas primitivas recebem como argumentos o canal global, o conjunto de mensagens a enviar, a origem e o destino. Os resultados destas primitivas devem representar de forma relativa os momentos em que conjuntos de mensagens devem ser enviados. Por exemplo, o resultado 0 indica que a mensagem deve ser imediatamente transmitida. A primitiva $mutation_0$ é usada para definir o *timeout* na ação Pre-Send, ou seja, sempre que o conjunto de mensagens a enviar é alterado. Por sua vez, a primitiva *mutation* define o *timeout* na ação Send, o que acontece quando o *timeout* definido anteriormente expira, marcando assim uma retransmissão.

$fresh(B, M, i) \equiv fresh\text{-}nc(B, M, i) \lor fresh\text{-}vc(B, M, i) \lor fresh\text{-}nv(B, M, i)$
$fresh\text{-}nc(B, M, i) \equiv \exists m, v, n : (((\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} \in M \land \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} \notin B) \lor (\langle \text{Prepare}, v, n, D(m), i\rangle_{\sigma_i} \in M \land \langle \text{Prepare}, v, n, D(m), i\rangle_{\sigma_i} \notin B) \lor (\langle \text{Commit}, v, n, D(m), i\rangle_{\sigma_i} \in M \land \langle \text{Commit}, v, n, D(m), i\rangle_{\sigma_i} \notin B))$
$fresh\text{-}vc(B, M, i) \equiv \exists v, P : (\langle \text{View-Change}, v, P, i\rangle_{\sigma_i} \in M \land \langle \text{View-Change}, v, P, i\rangle_{\sigma_i} \notin B)$
$fresh\text{-}nv(B, M, i) \equiv \exists v, X, O, N : (\langle \text{New-View}, v, X, O, N\rangle_{\sigma_i} \in M \land \langle \text{New-View}, v, X, O, N\rangle_{\sigma_i} \notin B)$
$maj(B, M) \equiv \exists m, v, n : (\neg committed(m, v, n, B) \land committed(m, v, n, M))$

Condições $fresh$ e maj

Criou-se ainda as funções $fresh$ e maj para auxiliar as mutações na determinação dos valores de *timeout*. A função auxiliar $fresh$ verifica se existe uma mensagem do processo origem no conjunto a enviar que não exista no conjunto mais recente do canal. Por sua vez, a função maj verifica se a condição *committed* é verdadeira no conjunto de mensagens a enviar e não o é no conjunto mais recente.

Foram implementados os padrões de comunicação *early*, *centralized*, *ring* e *gossip*, tal como no protocolo Mutable Consensus.

Early

O padrão *early* imita o padrão do protocolo *PBFT* em que cada processo envia imediatamente as mensagens que cria para todas os processos. Neste caso, não são apenas enviadas as próprias mensagens uma vez que para criar diferentes padrões de comunicação é necessário retransmitir mensagens recebidas. Para implementar este padrão, o valor em $early_0$ é o se $fresh$ for verdadeiro. Caso contrário deve ser 1, que é também o valor que é devolvido pela primitiva $early$.

```
procedure early_0(ch, M, i, j)
    if fresh(B_i(ch), M, i) then
        return 0
    else
        return 1
procedure early(ch, M, i, j)
    return 1
```

Primitivas para a mutação *early*

Centralized

No padrão *centralized* as mensagens apenas são enviadas para o coordenador que é o único que transmite mensagens para todos os processos. Para determinar o coordenador é

necessário conhecer a vista. Para isso é usada a função *min-v* que calcula o menor valor de vista num conjunto de mensagens. Isto é necessário para os canais de mensagens View-Change, já que podem existir mensagens de vistas diferentes.

$$min\text{-}v(M) \equiv min(\{v | \langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_k} \in M\} \cup \{v | \langle \text{Prepare}, v, n, D(m), k \rangle_{\sigma_k} \in M\} \cup \{v | \langle \text{Commit}, v, n, D(m), k \rangle_{\sigma_k} \in M\} \cup \{v | \langle \text{View-Change}, v, P, k \rangle_{\sigma_k} \in M\} \cup \{v | \langle \text{New-View}, v, V, O, N \rangle_{\sigma_{primary(v)}} \in M\})$$

Condição *min-v*

Este padrão pode-se dividir em duas partes: quando o destino é o coordenador ou quando a origem é o coordenandor. No primeiro caso, o resultado de $centralized_0$ é o se $fresh$ for verdadeiro. Já para o segundo caso, o seu valor é o se $fresh$ ou maj forem verdadeiros. Usar apenas $fresh$ não seria suficiente pois dessa forma apenas o coordenador iria atingir a condição *committed*, já que as restantes réplicas não receberiam mensagens Commit. Caso nenhum dos processos envolvidos seja o coordenador, o resultado é infinito para que mensagens nunca sejam transmitidas nesse canal.

Para a primitiva *centralized* o resultado é sempre 1, já que os únicos canais em que não deve ocorrer retransmissão é nos que o resultado da primitiva $centralized_0$ foi infinito e, devido a esse resultado, esta primitiva nunca irá ser chamada para esses canais.

Nos canais em que são transmitidas mensagens View-Change, não transmitir mensagens em alguns destes pode impedir o progresso do protocolo numa mudança de vista. Assim, nesta situção, é necessário que a mutação degenere na mutação *early*. Para isso, o resultado da primitiva $centralized_0$ deve ser 1 quando o canal é o das mensagens View-Change e nenhum dos processos envolvidos é o coordenador.

```
procedure centralized_0(ch, M, i, j)
    if i = primary(min-v(M)) then
        if fresh(B_i(ch), M, i) ∨ maj(B_i(ch), M) then
            return 0
        else
            return 1
    else if j = primary(min-v(M)) then
        if fresh(B_i(ch), M, i) then
            return 0
        else
            return 1
    else if ch = View-Change then
        return 1
    else
        return ∞
procedure centralized(ch, M, i, j)
    return 1
```

Primitivas para a mutação *centralized*

Ring

Para o padrão *ring*, o resultado é calculado de acordo com a distância lógica entre a origem e o destino, de forma a que as transmissões ocorram de forma sucessiva. Na primitiva $ring_0$ a primeira transmissão ocorre imediatamente se $fresh$ ou maj forem verdadeiros. Esta primitiva reestabelece a ordem de transmissões de cada vez que é chamada para todos os processos. Na primitiva *ring*, o resultado é definido de forma a que a retransmissão apenas ocorra após todas as outras transmissões agendadas acontecerem. Assim, o seu valor é o número de destinos possíveis.

```
procedure ring_0(ch, M, i, j)
    if fresh(B_i(ch), M, i) ∨ maj(B_i(ch), M) then
        return ((j − i) mod |R|) − 1
    else
        return ((j − i) mod |R|)
procedure ring(ch, M, i, j)
    return |R| − 1
```

Primitivas para a mutação *ring*

$c_i : \mathcal{CH} \rightarrow \mathbf{N}$, initially $\forall ch \in \mathcal{CH} : c_i(ch) = 0$

Estado que contem a ronda em que um processo se encontra para um dado canal

Gossip

O padrão de comunicação *gossip* faz uma difusão epidémica das mensagens. As primitivas para este padrão não dependem dos conjuntos de mensagens, o que faz com que sempre que ocorra uma alteração nestes são feitas novas transmissões. Dá-se o nome de ronda a cada conjunto de transmissões que são efetuadas no mesmo momento. É criada uma permutação, u, de todos os destinos possíveis, que é percorrida de forma sequencial e rotativa, para cada canal global. Existe ainda um valor de $fanout$ que determina para quantos destinos é enviada uma mensagem numa dada ronda.

Os resultados seguem uma lógica semelhante à do padrão *ring*, na medida em que há uma sequencialização dos mesmos. Em vez de ser usado o valor destino para calcular a distância lógica é usado o índice em que esse destino se encontra na permutação. O ponto de referência no cálculo dessa distância também deixa de ser a origem, passando a ser um valor que varia à medida que as rondas se sucedem. O valor da distância não é usado de forma absoluta, já que dessa forma apenas se enviaria para um destino em cada ronda. Assim, é necessário dividir pelo valor de $fanout$ e arredondar para o maior inteiro que seja menor ou igual ao resultado obtido. Caso se trate de uma retransmissão, é necessário acrescentar 1 ao valor calculado para que não ocorra a transmissão para dois conjuntos de destinos na mesma ronda.

O valor que determina o ponto de referência é calculado a partir da ronda em que se encontra. Para isso é necessário guardar a ronda de um dado canal, o que é feito através do estado c. A manipulação deste estado acarreta um cuidado especial, já que tem de ser usado e alterado por todos os destinos de um dado canal. Sabe-se que a primitiva $gossip_0$ é chamada para todos os destinos, porém não há garantias em relação à primitiva $gossip$. Se tudo correr conforme esperado, esta primitiva é chamada para todos os destinos de uma ronda de forma sucessiva. Porém, pode dar-se o caso de apenas ser chamado para alguns, por ocorrer uma mudança no conjunto de mensagens, por exemplo. Assim, a atualização de c é feita adicionando-se $\frac{1}{fanout}$ de cada vez que a primitiva *gossip* é chamada. Após esta primitiva ser chamada para os $fanout$ destinos de uma ronda, c foi incrementado. O uso deste estado implica o arredondamento para o menor inteiro que seja maior ou igual ao seu valor. Esta solução seria suficiente caso não ocorressem interrupções na chamada à primitiva $gossip$. No entanto, essa situação é possível, o que deixa deixa o incremento da variável incompleto. Para resolver este problema, faz-se o arredondamento para o menor inteiro, maior ou igual ao seu valor, ao chamar a primitiva $gossip_0$. Assim, se o incremento de c tiver sido interrompido, este é ajustado na primeira chamada a esta primitiva. Caso contrário, o

procedure $gossip_0(ch, M, i, j)$
let $j = u_i(ch)[ind]$
$c_i(ch) := \lceil c_i(ch) \rceil$
return $\lfloor ((ind - fanout \times c_i(ch)) \bmod |\mathcal{R}|) \div fanout \rfloor$

procedure $gossip(ch, M, i, j)$
let $j = u_i(ch)[ind]$
$c_i(ch) := c_i(ch) + \frac{1}{fanout}$
return $\lfloor ((ind - fanout \times \lceil c_i(ch) \rceil) \bmod |\mathcal{R}|) \div fanout \rfloor + 1$

Primitivas para a mutação *gossip*

estado não sofre qualquer alteração. Isto é também verdade para as restantes chamadas a $gossip_0$, quer a variável seja alterada ou não.

4.3 OTIMIZAÇÕES

As otimizações propostas no protocolo *PBFT* podem também ser aplicadas a este novo protocolo. Através da técnica de *checkpointing* pode-se reduzir a complexidade das mudanças de vista, uma vez que deixa de ser necessário considerar todos os números de sequência para construir as mensagens relativas a esta fase. A otimização de *tentative execution*, mencionada anteriormente na escolha do protocolo tolerante a faltas bizantinas, permite executar e responder a pedidos no final da primeira fase. Já a otimização de *request batching* faz com que em cada decisão sejam servidos vários pedidos, reduzindo assim o número de mensagens trocadas por pedido.

A remoção de mensagens, relativas ao caso normal, do *buffer* de saída após um pedido ser executado, pode comprometer a aceitação da resposta por parte do cliente, pois deixa de haver retransmissões dessas mensagens. Porém, enquanto que nas mutações *early* e *centralized* isto só acontece se um grande número de mensagens se perderem, na mutação *gossip* e principalmente na mutação *ring*, uma vez que são enviadas poucas mensagens e, consequentemente, para poucos destinos, este comportamento pode ser mais frequente. Note-se que a condição de remoção definida é a mais drástica possível, podendo pois ser relaxada de forma a manter a transmissão de mensagens durante algum tempo adicional. Ainda assim, a otimização de *message retransmission* do protocolo *PBFT* é importante para que se reduza esta possibilidade, que consiste em processos enviarem, periodicamente, informação sobre o que conhecem de forma a que outros processos lhes enviem mensagens que estejam em falta.

Propõe-se ainda uma nova otimização em que os processos guardam informação sobre as mensagens que recebem de outros processos. Com isto, ao enviarem mensagens, analisam essa informação de forma a não enviarem mensagens que os destinos já conhecem.

5

AVALIAÇÃO EXPERIMENTAL

Neste capítulo descreve-se a implementação dos algoritmos *PBFT* e *Mutable BFT* e apresenta-se os resultados das respetivas avaliações experimentais por forma a aferir o potencial de escalabilidade da mutação *gossip*.

5.1 IMPLEMENTAÇÃO

Os protocolos *PBFT* e *Mutable BFT* foram implementados utilizando a versão 8 da linguagem Java, com o auxílio de primitivas de comunicação da *framework* Atomix que oferecem comunicação assíncrona por eventos.

A implementação do protocolo *Mutable BFT* foi realizada a partir da implementação do protocolo *PBFT*. As otimizações enunciadas no final do capítulo anterior não foram consideradas em qualquer uma das implementações.

A implementação a partir da especificação em *IO Automata* revelou alguns desafios. As ações de input são facilmente reproduzidas, através dos eventos que são definidos para a receção de mensagens. Porém as restantes ações, de output e internas, são ativadas por pré-condições que dependem do que contêm os estados. Assim, relativamente às ações de output do protocolo *PBFT*, uma vez que não foi implementado um *buffer* de saída, são chamados métodos que efetuam o envio de mensagens nas situações em que na especificação são adicionadas a esse *buffer*. Para as ações internas relativas ao protocolo *PBFT*, o seu comportamento foi criado implementando métodos que verificam a pré-condição e executam a ação. Estes métodos são chamados sempre que ocorre uma alteração que afete essas pré-condições.

Por sua vez, as ações criadas para o novo protocolo foram implementados de forma distinta. As ações Relay e Drop manipulam o *buffer* de saída que, como referido anteriormente, não foi implementado, pelo que não existe uma implementação direta destas ações. O que acontece é que as condições para mensagens serem enviadas, definidas por estas ações, são consideradas no envio de mensagens. Nas ações relativas aos canais *stubborn* é necessário detetar quando existem novas mensagens a enviar. Para isso, são chamados os métodos que processam o envio de mensagens quando novas mensagens são aceites ou criadas. Estes

métodos tratam então de construir os conjuntos de mensagens a enviar e definir os destinos para onde são enviados. Os conjuntos de mensagens contêm uma mensagem Pre-Prepare e assinaturas de mensagens Prepare e Commit conhecidas. Também o cálculo das mutações e computação dos respetivos valores de *timeout* é feito de forma diferente da especificação do protocolo. Em vez de serem calculados valores de *timeout* para cada destino, determina-se os destinos para os quais um conjunto de mensagens deve ser enviado e define-se um *timeout* igual ao delta. Quando esse *timeout* expira, volta-se a determinar os destinos para os quais o mesmo conjunto de mensagens deve ser transmitido e é novamente definido um *timeout* igual ao delta, sendo este processo repetido.

Porém, as maiores dificuldades surgiram na implementação do registo que guarda as mensagens e das funções auxiliares *last-prepared*, *correct-view-change* e *correct-new-view*. Uma vez que os acessos ao registo são muito frequentes é importante que não sejam um entrave ao desempenho. Assim, o registo consiste em várias estruturas de dados que permitem efetuar esses acessos de forma rápida, nomeadamente através do uso de tabelas *hash*. A função auxiliar *last-prepared*, usada na mudança de vista, indica para um valor de vista e um pedido se é nessa vista que o pedido atingiu a condição *prepared* mais recentemente. Na criação de mensagens View-Change são adicionadas todos as mensagens cujos valor de vista e pedido satisfazem a condição *last-prepared*. De forma a não fazer essa verificação exaustiva criou-se uma estrutura que guarda e substitui os valores de vista e pedidos à medida que vão atingindo a condição *prepared*. Assim, ao construir mensagens View-Change basta obter todos os dados dessa estrutura. Adicionalmente, a condição *last-prepared* também é usada na verificação de mensagens View-Change e New-View, nomeadamente através das funções *correct-view-change* e *correct-new-view*. Para fazer essas verificações é usada a mesma estrutura juntamente com as estruturas do registo, imitando o comportamento do caso normal e procurando por sobreposições, que não devem acontecer, na condição *last-prepared*.

5.2 RESULTADOS

Nesta secção efetua-se uma avaliação experimental das implementações realizadas por forma a aferir o potencial de escalabilidade da mutação *gossip*. Isto é feito comparando essa mutação com as restantes mutações e também com o próprio protocolo *PBFT*. Apenas se considera o melhor caso, uma vez que assumir faltas implica configurar períodos de retransmissão e lidar com mudanças de vista, não sendo estes fatores relevantes para estimar o potencial de escalabilidade.

Esta avaliação é realizada recorrendo a um sistema de simulação [Carvalho et al. (2011)] que permite executar um grande número de processos em apenas uma máquina, sem comprometer as medições a efetuar por uma eventual sobrecarga da máquina utilizada.

Sendo o principal fator que afeta a escalabilidade de um protocolo o número de mensagens trocadas [Vukolić (2015)], procura-se analisar o número de mensagens recebidas por processo. Uma vez que o conteúdo das mensagens trocadas nos dois protocolos implementados é significativamente diferente, também se calcula o tamanho de cada mensagem trocada. Resta ainda considerar o impacto que a forma como as mensagens são trocadas têm na escalabilidade. Para isso, observa-se então a latência e o débito das decisões explorando a capacidade do sistema de simulação de virtualizar o tempo. Por forma a determinar estas estatísticas guarda-se informação em ficheiros, à medida que as simulações são executadas, associando a cada registo um marco temporal. Relativamente aos processos é inserido um registo ao ser recebida uma mensagem, que inclui o tamanho da mensagem, e também ao executar um pedido. Já para os clientes são inseridos registos ao efetuar um pedido e ao aceitar uma resposta.

Foram obtidos resultados para populações de, aproximadamente, 100, 200, 300 e 400 processos. Por sua vez, o número de clientes, que representa o número de decisões em paralelo, foi definido em 2, já que inicialmente se testou com poucos processos e com 1 cliente obtinha-se aproximadamente a mesma latência mas metade do débito, enquanto que com 4 obtinha-se aproximadamente o mesmo débito mas o dobro da latência. Uma vez que apenas se obtém resultados para um único valor de número de clientes, não são apresentados os resultados para o débito, já que estes são inversamente proporcionais à latência média. Cada cliente realiza 100 pedidos consecutivamente, isto é, imediatamente após aceitar a resposta relativa ao pedido anterior. Durante os primeiros pedidos são realizadas várias inicializações, como por exemplo das estruturas de dados, pelo que estes não são considerados no cálculo da latência e do débito.

A figura 7 contém o número médio de mensagens recebida por processo em cada decisão e o tamanho médio de cada mensagem.

O número de mensagens pode ser calculado deterministicamente, exceto para a mutação *gossip*. Assim, na mutação *early* e também no protocolo *PBFT*, cada processo recebe duas mensagens de cada outro processo, ou seja, um total de duas vezes o número de processos. Para a mutação *centralized* deve ser feita uma distinção entre o número de mensagens recebidas pelo coordenador e pelos restantes processos. O coordenador recebe o mesmo número de mensagens como na mutação *early*, enquanto que os restantes processos recebem apenas 3 mensagens em cada decisão. Assim, o valor médio, 5, só é representativo após várias vistas, já que apenas assim é que a carga a que o coordenador está sujeito é distribuída por vários processos. Por fim, na mutação *ring*, o número de mensagens recebidas por qualquer processo não depende do número de processos, sendo aliás constante. Cada processo recebe entre 2 a 3 mensagens, dependendo da sua distancia lógica ao coordenador. Para a mutação *gossip* o número de mensagens recebidas é então calculado a partir dos registos que são inseridos nos ficheiros. Comparativamente à mutação *early* (e ao próprio

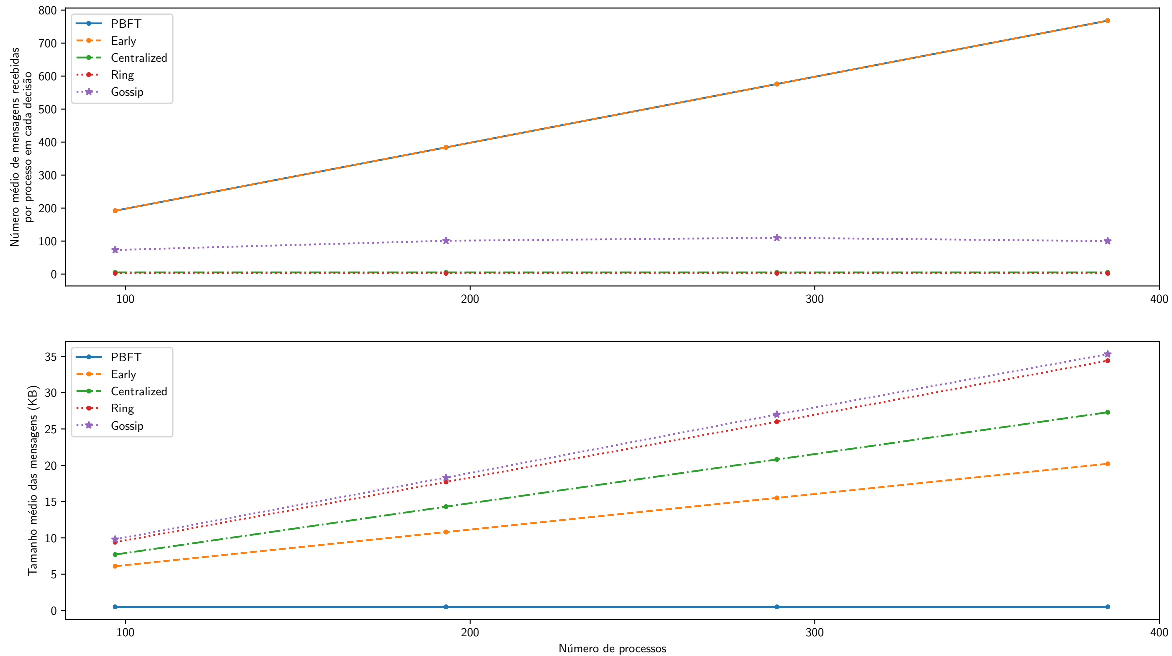


Figura 7: Número médio de mensagens recebidas por processo em cada decisão e tamanho médio de cada mensagem

protocolo *PBFT*) o número de mensagens por processo é significativamente menor nesta mutação. Já por comparação com as mutações *centralized* e *ring* este valor é um pouco superior.

Em relação ao tamanho das mensagens, para o protocolo *PBFT* este não é influenciado pelo número de processos, sendo sempre os mesmos 500 *bytes*, aproximadamente. Já no protocolo proposto, uma vez que ocorre uma acumulação de assinaturas, o tamanho das mensagens cresce quer com o número de processos, quer ao longo de cada decisão. O tamanho máximo destas mensagens corresponde à existência de 2t+1 assinaturas. Na mutação *early* a primeira ronda de mensagens apenas contêm uma mensagem Pre-Prepare e a assinatura de uma mensagem Prepare. Já na segunda ronda, as mensagens trocadas contêm 2t+1 assinaturas, em que uma delas é de uma mensagem Commit. Para a mutação *centralized* o tamanho médio das mensagens no coordenador é o mesmo da mutação *early*, enquanto que para os restantes processos é ligeiramente superior já que das três mensagens que recebem em cada decisão, duas delas contêm 2t+1 assinaturas. Na mutação *ring* o tamanho das mensagens vai crescendo até atingirem 2t+1 assinaturas de mensagens Prepare e, a partir desse momento, essas assinaturas vão sendo substituídas por assinaturas de mensagens Commit até existirem apenas assinaturas deste tipo de mensagens. Para a mutação *gossip* o tamanho das mensagens segue a mesma lógica da mutação *ring*, com um pequeno aumento na média relativamente a esta uma vez que as mensagens atingem o tamanho máximo mais rapidamente.

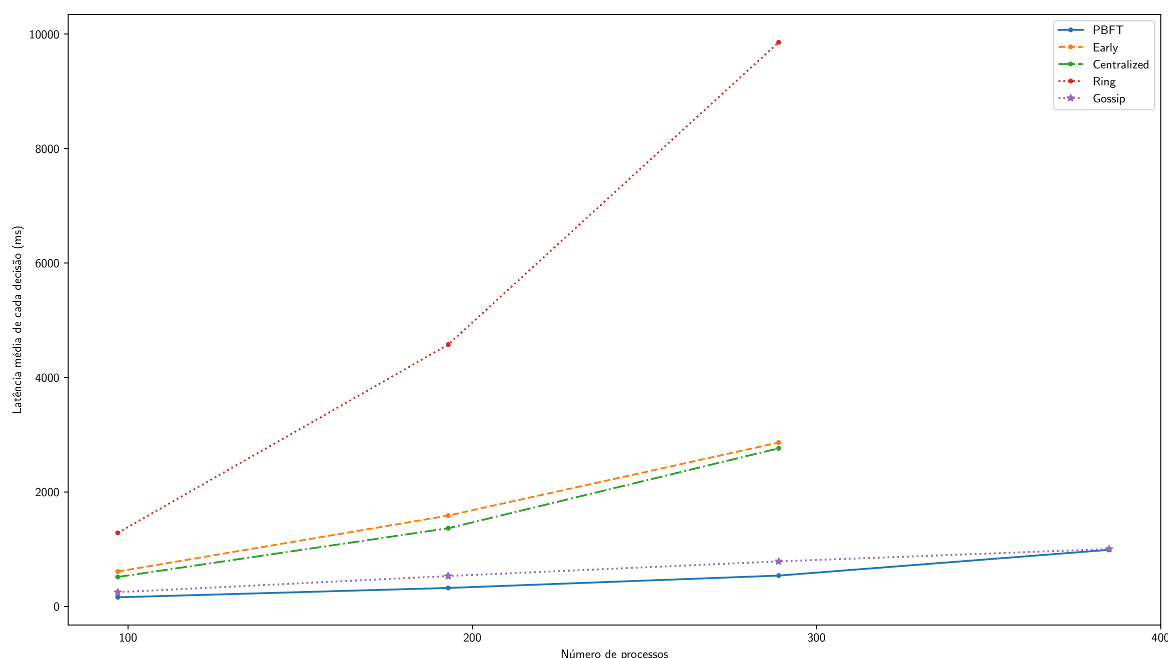


Figura 8: Latência média de cada decisão

A figura 8 apresenta a latência média de cada decisão. A latência média de cada decisão para a mutação *gossip* é menor do que em qualquer outra mutação, porém, por comparação com o protocolo *PBFT*, é ligeiramente superior. Apesar de nas mutações *centralized* e *ring* serem trocadas significativamente menos mensagens do que na mutação *gossip*, a forma como são trocadas influencia a demora na chegada a uma decisão. Apesar de o número médio de mensagens recebidas por processo, na mutação *centralized* ser pequeno, o facto de todas essas mensagens envolverem o mesmo processo, o coordenador, impede a escalabilidade desta mutação. Já para a mutação *ring* este resultado deve-se a não haver qualquer paralelismo em cada decisão. Isto pode porém ser aproveitado para correr um grande número de decisões diferentes em paralelo, o que permitiria aumentar significativamente o débito apesar do alto valor de latência. No entanto, na presença de faltas a latência iria subir para valores ainda mais altos, quebrando assim qualquer possibilidade de escalabilidade. Relativamente aos resultados da latência para o protocolo *PBFT*, estas são menores devido ao tamanho das mensagens trocadas.

Importa ainda analisar uma eventual carga computacional adicional em cada processo do protocolo *Mutable BFT* relativamente ao protocolo *PBFT*. Na receção das mensagens do protocolo *Mutable BFT*, é criada uma mensagem de decisão (Prepare ou Commit) para cada assinatura recebida, apesar de posteriormente apenas serem processadas as mensagens novas. Por sua vez, no envio de mensagens é necessário obter todas as assinaturas e criar a mensagem que as contém. São ainda calculados os destinos em cada envio, por oposição a simplesmente enviar para todos os destinos. Note-se no entanto que isto apenas se

aplica à implementação efetuada, pelo que uma implementação otimizada para o algoritmo *Mutable BFT* deverá conseguir mitigar qualquer diferença relativamente ao protocolo *PBFT*. Essa otimização não foi realizada uma vez que a implementação foi adaptada de uma implementação do protocolo *PBFT*.

6 CONCLUSÃO

O trabalho realizado nesta dissertação pode ser dividido em três partes: a escolha de um protocolo que tolera faltas bizantinas para dotá-lo com as características do protocolo *Mutable Consensus*; a extensão da especificação do protocolo escolhido para abrigar essas características, criando assim um novo protocolo; e a implementação do protocolo escolhido e do protocolo criado para efetuar a avaliação de ambos.

A escolha entre os protocolos *PBFT* e *XPaxos* foi feita de acordo com o que apresenta maior potencial de escalabilidade. Para isso, procurou-se perceber o comportamento dos dois protocolos com a mutação *gossip* do protocolo *Mutable Consensus*, uma vez que é com esta mutação que se espera atingir maior escalabilidade. A comparação foi realizada tendo em conta o número de mensagens trocadas e o número de iterações até ser atingida a decisão, valores esses que foram calculados através de simulações. Após a análise dos resultados obtidos e de serem consideradas várias características de cada protocolo, optou-se pelo protocolo *PBFT*.

À especificação do protocolo *PBFT* foram acrescentadas as características do protocolo *Mutable Consensus*. Em primeiro lugar cada processo deve enviar não só as mensagens que cria como também as mensagens que aceita de outros processos. De seguida, os processos deixam de enviar mensagens apenas quando as criam, passando a enviar sempre que tomam conhecimento de uma nova mensagem. Adicionalmente, ao contrário do que acontece no protocolo *PBFT*, as mensagens não são necessariamente transmitidas logo após serem enviadas pelo protocolo. Em vez disso, criou-se canais de comunicação, chamados *stubborn*, que transmitem e retransmitem as mensagens mediante algumas condições. Essas condições, constituídas pelas mutações, permitem a criação de diferentes padrões de comunicação.

Por fim, procedeu-se à implementação do protocolo *PBFT* e do protocolo *Mutable BFT* criado para os avaliar. Entre as mutações do protocolo *Mutable BFT*, a mutação *gossip* é, como esperado, a que permite atingir maior escalabilidade. Esta conclusão advém dos resultados obtidos para a latência média na chegada a uma decisão, já que é esta que considera todas as características das mutações, desde o número de mensagens trocadas à forma como são trocadas. A mutação *gossip* apresenta no entanto, pior latência do que o protocolo *PBFT*, que tem o mesmo padrão de comunicação que a mutação *early*. Esta

diferença deve-se essencialmente ao tamanho das mensagens trocadas, por serem enviadas mensagens de outros processos. Ainda assim, quer ao nível do protocolo, quer ao nível da implementação, foram tomadas medidas para reduzir o tamanho das mensagens. A alteração da condição *prepared* para considerar mensagens da segunda fase permite reduzir o tamanho das mensagens a metade durante a segunda fase, uma vez que não é necessário enviar mensagens das duas fases de um mesmo processo. Para além disso, na implementação do protocolo, para além da mensagem do coordenador, apenas são incluídas as assinaturas em vez das mensagens completas, já que é possível reconstruir as mensagens originais no destino a partir da mensagem do coordenador.

Em suma, conseguiu-se criar um protocolo tolerante a faltas bizantinas que contém as características do protocolo *Mutable Consensus* permitindo assim alterar padrões de comunicação sem ser necessário substituir protocolos e também sem colocar em causa a correção do protocolo. Adicionalmente, através da mutação *gossip* mostrou-se que existe um grande potencial de escalabilidade, já que a diferença para a mutação *early*, que imita o protocolo *PBFT*, é significativa.

6.1 TRABALHO FUTURO

O protocolo proposto como contribuição deste trabalho abre novos rumos de investigação como o desenho de novos protocolos de acordo e também a sua aplicação a sistemas *blockchain*.

O protocolo fornece uma base para que, recorrendo a técnicas criptográficas [Boneh (2011)], se consiga substituir os conjuntos de assinaturas que são enviados em assinaturas únicas. Aplicando estas técnicas ao protocolo *Mutable BFT* de tal forma que a mutação *early* tenha resultados semelhantes aos do protocolo *PBFT*, fará com que as restantes mutações também revelem uma melhoria em relação aos resultados obtidos. Desta forma, a mutação *gossip* será mais escalável do que o protocolo *PBFT*, da mesma forma que o é relativamente à mutação *early*.

A integração do protocolo proposto num sistema *blockchain* é outro caminho que pode ser explorado, por forma a efetuar uma avaliação prática do protocolo e da sua adaptabilidade.

BIBLIOGRAFIA

Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making bft protocols really adaptive. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 904–913. IEEE, 2015.

Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep.*, 2016.

Dan Boneh. *Aggregate Signatures*, pages 27–27. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_139. URL https://doi.org/10.1007/978-1-4419-5906-5_139.

Nuno A Carvalho, João Bordalo, Filipe Campos, and José Pereira. Experimental evaluation of distributed middleware with a virtualized java environment. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 3. ACM, 2011.

Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999a.

Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999b.

Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

Everledger. URL https://everledger.io.

Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1982.

Rachid Guerraoui and Rui Oliveira. Stubborn communication channels. In *LSE, D'epartement d'Informatique, Ecole Polytechnique F'ed'erale de*. Citeseer, 1996.

Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

Leslie Lamport and Mike Massa. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, pages 307–314. IEEE, 2004.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

Jon Foss Mikalsen. Firechain: An efficient blockchain protocol using secure gossip. Master's thesis, UiT Norges arktiske universitet, 2018.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

José Pereira and Rui Oliveira. The mutable consensus protocol. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 218–227. IEEE, 2004.

Fred B Schneider. What good are models and what models are good. *Distributed systems*, 2: 17–26, 1993.

Robbert van Renesse. A blockchain based on gossip?-a position paper. *Cornell University*, 2016.

Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.

RESULTADOS DA ANÁLISE DE ESCALABILIDADE

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	36,27	8,36	18,77
	3	74,03	17,90	47,50
	4	88,92	22,36	74,23
60	2	59,06	10,56	26,67
	3	152,71	27,10	64,79
	4	113,68	45,12	121,3
120	2	73,10	16,99	27,49
	3	229,35	40,83	83,79
	4	180,53	90,64	168,16
240	2	111,48	17,03	31,64
	3	335,67	61,38	109,40
	4	362,57	45,41	216,05
480	2	138,97	21,48	38,09
	3	430,89	91,55	131,77
	4	726,66	90,93	292,65

Tabela 1: Número médio de mensagens trocadas por processo (algoritmo de gossip base)

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	9,10	8,00	38,50
	3	6,55	6,00	13,44
	4	5,34	5,00	9,47
60	2	10,76	9,25	70,50
	3	7,92	7,00	16,27
	4	6,09	6,00	10,99
120	2	12,08	11,00	104,33
	3	8,91	8,00	18,39
	4	7,00	7,00	12,53
240	2	13,04	12,00	121,50
	3	9,87	9,00	20,69
	4	8,00	7,00	13,82
480	2	15,02	13,26	246,00
	3	10,67	9,99	22,31
	4	9,00	8,00	15,22

Tabela 2: Número médio de iterações para atingir decisão (algoritmo de gossip base)

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	21,98	8,19	-
	3	43,34	17,46	28,03
	4	60,48	21,95	45,89
60	2	33,01	10,94	-
	3	66,95	26,63	40,61
	4	87,33	44,08	68,58
120	2	46,49	16,89	-
	3	98,00	40,44	52,39
	4	142,43	88,39	95,90
240	2	63,78	17,01	-
	3	139,04	61,05	69,13
	4	220,02	45,31	130,66
480	2	82,50	22,83	-
	3	188,41	91,31	86,67
	4	308,76	90,77	171,63

Tabela 3: Número médio de mensagens trocadas por processo (envios condicionais)

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	9,68	8,00	-
	3	6,91	6,00	13,60
	4	5,68	6,00	9,86
60	2	11,23	9,31	-
	3	8,02	7,00	16,33
	4	6,39	6,00	11,42
120	2	12,55	11,00	-
	3	9,01	8,00	19,40
	4	7,26	7,00	13,17
240	2	14,00	12,00	-
	3	10,02	9,00	21,20
	4	8,10	7,00	14,30
480	2	15,25	13,34	-
	3	10,81	9,99	22,00
	4	9,02	8,00	15,46

Tabela 4: Número médio de iterações para atingir decisão (envios condicionais)

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	24,28	11,93	25,18
	3	41,29	19,70	34,88
	4	55,27	22,85	49,94
60	2	35,26	15,80	32,50
	3	67,18	31,87	47,64
	4	83,80	47,13	74,53
120	2	49,79	18,79	37,76
	3	98,04	51,46	63,63
	4	143,90	91,71	104,11
240	2	67,41	21,95	44,02
	3	138,27	29,45	82,26
	4	221,48	51,44	138,78
480	2	88,32	25,75	50,20
	3	189,93	43,90	101,48
	4	450,09	106,18	179,71

Tabela 5: Número médio de mensagens trocadas por processo (retransmissões)

		XPaxos	XpensivePaxos	
			melhor caso	pior caso
30	2	9,80	7,83	25,98
	3	6,69	6,00	14,79
	4	5,51	5,00	9,99
60	2	10,93	9,00	31,17
	3	7,97	7,00	16,76
	4	6,23	6,00	11,73
120	2	12,21	10,00	34,44
	3	8,87	8,00	18,36
	4	7,20	7,07	12,83
240	2	13,68	11,00	38,80
	3	9,75	8,04	21,09
	4	8,06	7,00	13,91
480	2	14,70	12,00	43,28
	3	10,39	9,00	22,21
	4	9,00	8,00	14,93

Tabela 6: Número médio de iterações para atingir decisão (retransmissões)

		XPaxos	PBFT	
			melhor caso	pior caso
60	2	24,28	13,22	26,02
	3	41,29	19,57	43,48
	4	55,27	22,91	65,35
120	2	35,26	15,91	34,38
	3	67,18	37,74	63,71
	4	83,80	47,30	98,88
240	2	49,79	19,17	43,48
	3	98,04	51,16	86,43
	4	143,90	97,02	143,53
480	2	67,41	26,44	53,30
	3	138,27	82,15	115,21
	4	221,48	68,59	196,04
960	2	88,32	42,83	64,11
	3	189,93	124,64	147,77
	4	450,09	106,25	258,72

Tabela 7: Número médio de mensagens trocadas por processo (mesma população)

		XPaxos	PBFT	
			melhor caso	pior caso
30	2	9,80	8,01	18,50
	3	6,69	6,00	10,38
	4	5,51	5,00	7,99
60	2	10,93	9,00	21,32
	3	7,97	7,00	12,14
	4	6,23	6,00	9,04
120	2	12,21	10,02	23,31
	3	8,87	8,00	13,05
	4	7,20	7,00	10,19
240	2	13,68	11,21	25,19
	3	9,75	9,00	14,03
	4	8,06	7,12	10,94
480	2	14,70	12,66	27,52
	3	10,39	9,93	15,42
	4	9,00	8,00	11,79

Tabela 8: Número médio de iterações para atingir decisão (mesma população)

		XPaxos	PBFT	
			melhor caso	pior caso
30	2	24,28	19,43	30,71
	3	41,29	40,08	54,45
	4	55,27	60,28	83,93
60	2	35,26	23,56	39,60
	3	67,18	64,37	76,64
	4	83,80	97,61	122,26
120	2	49,79	27,41	49,16
	3	98,04	35,90	102,68
	4	143,90	67,52	171,87
240	2	67,41	32,28	59,15
	3	138,27	57,13	133,34
	4	221,48	139,05	231,71
480	2	88,32	37,59	70,54
	3	189,93	91,18	168,14
	4	450,09	274,59	303,47

Tabela 9: Número médio de iterações para atingir decisão (mesma tolerância)

		XPaxos	PBFT	
			melhor caso	pior caso
30	2	9,80	9,00	19,99
	3	6,69	7,00	11,38
	4	5,51	6,00	8,49
60	2	10,93	10,00	22,15
	3	7,97	8,00	12,48
	4	6,23	6,74	9,56
120	2	12,21	11,00	24,68
	3	8,87	8,00	13,68
	4	7,20	7,00	10,60
240	2	13,68	12,00	26,52
	3	9,75	9,00	14,75
	4	8,06	8,00	11,45
480	2	14,70	13,00	28,97
	3	10,39	10,00	15,70
	4	9,00	8,95	12,19

Tabela 10: Número médio de iterações para atingir decisão (mesma tolerância)

B

ESPECIFICAÇÃO COMPLETA DO PROTOCOLO

B.1 CLIENTE

B.1.1 *Transições*

Request$(o)_c$

Eff:

$last\text{-}req_c := last\text{-}req_c + 1$

$out_c := \{\langle \text{Request}, o, last\text{-}req_c, c\rangle_{\sigma_c}\}$

$in_c := \{\}$

$retrans_c := false$

Send$(m, \{view_c \bmod |\mathcal{R}|\})_c$

Pre:

$m \in out_c \land \neg retrans_c$

Eff:

$retrans_c := true$

Send$(m, \mathcal{R})_c$

Pre:

$m \in out_c \land retrans_c$

Eff:

none

Receive$(\langle \text{Reply}, v, t, c, i, r\rangle_{\sigma_i})_c$

Eff:

if $out_c \neq \{\} \land last\text{-}req_c = t$ then

$in_c := in_c \cup \{\langle \text{Reply}, v, t, c, i, r\rangle_{\sigma_i}\}$

$\text{Reply}(r)_c$

Pre:

$out_c \neq \{\} \wedge \exists R : (|R| > f \wedge \forall i \in R : (\exists v : (\langle \text{Reply}, v, \textit{last-req}_c, c, i, r \rangle_{\sigma_i} \in in_i)))$

Eff:

$view_c := max(\{v | \langle \text{Reply}, v, \textit{last-req}_c, c, i, r \rangle_{\sigma_i} \in in_i\})$

$out_c := \{\}$

B.2 PROCESSO

B.2.1 *Estado*

$t_i : \mathcal{CH} \times \mathcal{R} \rightarrow \mathbf{N}$, initially $\forall ch \in \mathcal{CH}, j \in \mathcal{R} : t_i(ch, j) = \infty$

$B_i : \mathcal{CH} \rightarrow 2^{\mathcal{M}}$, initially $\forall ch \in \mathcal{CH} : B_i(ch) = \{\}$

B.2.2 *Funções auxiliares*

$primary(v) \equiv v \text{ mod } |\mathcal{R}|$

$primary(i) \equiv view_i \text{ mod } |\mathcal{R}|$

$\textit{in-v}(v, i) \equiv view_i = v$

$prepared(m, v, n, M) \equiv \langle \text{Pre-Prepare}, v, n, m \rangle_{\sigma_{primary(v)}} \in M \wedge \exists R : (|R| \geq 2t \wedge primary(v) \notin R \wedge \forall k \in R : (\langle \text{Prepare}, v, n, D(m), k \rangle_{\sigma_k} \in M \vee \langle \text{Commit}, v, n, D(m), k \rangle_{\sigma_k} \in M))$

$prepared(m, v, n, i) \equiv prepared(m, v, n, in_i)$

$committed(m, v, n, i) \equiv (\exists v' : (\langle \text{Pre-Prepare}, v', n, m \rangle_{\sigma_{primary(v)}} \in in_i) \vee m \in in_i) \wedge \exists R : (|R| \geq 2f + 1 \wedge \forall k \in R : (\langle \text{Commit}, v, n, D(m), k \rangle_{\sigma_k} \in in_i))$

$\textit{has-new-view}(v, i) \equiv v = 0 \vee \exists m : (m \in in_i \wedge \textit{correct-new-view}(m, v))$

$last\text{-}prepared(m,v,n,M) \equiv prepared(m,v,n,M) \wedge \nexists m',v' : ((prepared(m',v',n,M) \wedge v' > v) \vee (prepared(m',v,n,M) \wedge m \neq m')$
$last\text{-}prepared(m,v,n,i) \equiv last\text{-}prepared(m,v,n,in_i)$
$correct\text{-}view\text{-}change(m,v,j) \equiv \exists P : (m = \langle \text{View-Change}, v, P, j\rangle_{\sigma_j} \wedge \forall \langle \text{Pre-Prepare}, v', n, m'\rangle_{\sigma_{primary(v')}} \in P : (last\text{-}prepared(m',v',n,P) \wedge v' < v))$
$merge\text{-}P(V) \equiv \{m | \exists \langle \text{View-Change}, v, P, k\rangle_{\sigma_k} \in V : m \in P\}$
$max\text{-}n(M) \equiv max(\{n | \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} \in M\})$
$correct\text{-}new\text{-}view(m,v) \equiv \exists V,O,N,R : (m = \langle \text{New-View}, v, V, O, N\rangle_{\sigma_{primary(v)}} \wedge |V| = |R| = 2f+1 \wedge \forall k \in R : (\exists m' \in V : (correct\text{-}view\text{-}change(m',v,k))) \wedge O = \{\langle \text{Pre-Prepare}, v, n, m'\rangle_{\sigma_primary(v)} | \exists v' : last\text{-}prepared(m',v',n,merge\text{-}P(V))\} \wedge N = \{\langle \text{Pre-Prepare}, v, n, null\rangle_{\sigma_primary(v)} | n < max\text{-}n(O) \wedge / \exists v',m',n : last\text{-}prepared(m',v',n,merge\text{-}P(V))\})$

$prepared\text{-}certificate(m,v,n,M) \equiv \{c = \langle \text{Commit}, v, n, D(m), k\rangle_{\sigma_k} | c \in M\} \cup \{p = \langle \text{Prepare}, v, n, D(m), k\rangle_{\sigma_k} | p \in M\} \cup \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}}\}$
$fresh(B,M,i) \equiv fresh\text{-}nc(B,M,i) \vee fresh\text{-}vc(B,M,i) \vee fresh\text{-}nv(B,M,i)$
$fresh\text{-}nc(B,M,i) \equiv \exists m,v,n : (((\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} \in M \wedge \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} \notin B) \vee ((\langle \text{Prepare}, v, n, D(m), i\rangle_{\sigma_i} \in M \wedge \langle \text{Prepare}, v, n, D(m), i\rangle_{\sigma_i} \notin B) \vee ((\langle \text{Commit}, v, n, D(m), i\rangle_{\sigma_i} \in M \wedge \langle \text{Commit}, v, n, D(m), i\rangle_{\sigma_i} \notin B))$
$fresh\text{-}vc(B,M,i) \equiv \exists v,P : ((\langle \text{View-Change}, v, P, i\rangle_{\sigma_i} \in M \wedge \langle \text{View-Change}, v, P, i\rangle_{\sigma_i} \notin B)$
$fresh\text{-}nv(B,M,i) \equiv \exists v,X,O,N : ((\langle \text{New-View}, v, X, O, N\rangle_{\sigma_i} \in M \wedge \langle \text{New-View}, v, X, O, N\rangle_{\sigma_i} \notin B)$
$maj(B,M) \equiv \exists m,v,n : (\neg committed(m,v,n,B) \wedge committed(m,v,n,M))$

B.2.3 *Transições*

Receive$(\langle \text{Request}, o, t, c\rangle_{\sigma_c})_i$
  Eff:
    let $m = \langle \text{Request}, o, t, c\rangle_{\sigma_c}$
      if $t = last\text{-}rep\text{-}t_i(c)$ then
        $out_i := out_i \cup \{\langle \text{Reply}, view_i, t, c, i, last\text{-}rep_i(c)\rangle_{\sigma_i}\}$
      else if $t > last\text{-}rep\text{-}t_i(c)$ then
        $in_i := in_i \cup \{m\}$
        if $primary(i) \neq i$ then
          $out_i := out_i \cup \{m\}$

Send$(\langle \text{Reply}, v, t, c, i, r\rangle_{\sigma_i}, \{c\})_i$

Pre:

$\langle \text{Reply}, v, t, c, i, r\rangle_{\sigma_i} \in out_i$

Eff:

$out_i := out_i \setminus \{\langle \text{Reply}, v, t, c, i, r\rangle_{\sigma_i}\}$

Send$(m, \{primary(i)\})_i$

Pre:

$m \in out_i \wedge tag(m, \text{Request})$

Eff:

$out_i := out_i \setminus \{m\}$

Send-Pre-Prepare$(m, v, n)_i$

Pre:

$primary(i) = i \wedge seqno_i = n - 1 \wedge in\text{-}v(v, i) \wedge has\text{-}new\text{-}view(v, i) \wedge \exists o, t, c : (m = \langle \text{Request}, o, t, c\rangle_{\sigma_c} \wedge m \in in_i) \wedge \nexists \langle \text{Pre-Prepare}, v, n', m\rangle_{\sigma_i} \in in_i$

Eff:

$seqno_i := seqno_i + 1$

let $p = \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i}$

$out_i := out_i \cup \{p\}$

$in_i := in_i \cup \{p\}$

Receive$(\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_j})_i$ $(j \neq i)$

Eff:

if $j = primary(i) \wedge in\text{-}v(v, i) \wedge has\text{-}new\text{-}view(v, i) \wedge \; / \exists d : (d \neq D(m) \wedge \langle \text{Prepare}, v, n, d, i\rangle_{\sigma_i} \in in_i)$ then

let $p = \langle \text{Prepare}, v, n, D(m), i\rangle_{\sigma_i}$

$in_i := in_i \cup \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_j}, p\}$

$out_i := out_i \cup \{p\}$

else if $\exists o, t, c : (m = \langle \text{Request}, o, t, c\rangle_{\sigma_c})$ then

$in_i := in_i \cup \{m\}$

Receive$(\langle \text{Prepare}, v, n, d, j\rangle_{\sigma_j})_i$ $(j \neq i)$

Eff:

if $j \neq primary(i) \wedge in\text{-}v(v, i)$ then

$in_i := in_i \cup \{\langle \text{Prepare}, v, n, d, j\rangle_{\sigma_j}\}$

Send-Commit$(m, v, n)_i$

Pre:

$prepared(m, v, n, i) \land \langle \text{Commit}, v, n, D(m), i \rangle_{\sigma_i} \notin in_i$

Eff:

let $c = \langle \text{Commit}, v, n, D(m), i \rangle_{\sigma_i}$

$out_i := out_i \cup \{c\}$

$in_i := in_i \cup \{c\}$

Receive$(\langle \text{Commit}, v, n, d, j \rangle_{\sigma_j})_i$ $(j \neq i)$

Eff:

if $in\text{-}v(v, i)$ then

$in_i := in_i \cup \{\langle \text{Commit}, v, n, d, j \rangle_{\sigma_j}\}$

Execute$(m, v, n)_i$

Pre:

$n = last\text{-}exec_i + 1 \land committed(m, v, n, i)$

Eff:

$last\text{-}exec_i := n$

if $m \neq null$ then

let $\langle \text{Request}, o, t, c \rangle_{\sigma_c} = m$

if $t \geq last\text{-}rep\text{-}t_i(c)$ then

if $t > last\text{-}rep\text{-}t_i(c)$ then

$last\text{-}rep\text{-}t_i(c) := t$

$(last\text{-}rep_i(c), val_i) := g(c, o, val_i)$

$out_i := out_i \cup \{\langle \text{Reply}, view_i, t, c, i, last\text{-}rep_i(c) \rangle_{\sigma_i}\}$

$in_i := in_i \setminus \{m\}$

Send-View-Change$(v)_i$

Pre:

$v = view_i + 1$

Eff:

$view_i := v$

let $P' = \{\langle m, v, n\rangle | \textit{last-prepared}(m, v, n, i)\}$

$P = \bigcup_{\langle m,v,n\rangle \in P'} (\{c = \langle \text{Commit}, v, n, D(m), k\rangle_{\sigma_k} | c \in in_i\} \cup \{p = \langle \text{Prepare}, v, n, D(m), k\rangle_{\sigma_k} | p \in in_i\} \cup \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}}\})$

$m = \langle \text{View-Change}, v, P, i\rangle_{\sigma_i}$

$out_i := out_i \cup \{m\}$

$in_i := in_i \cup \{m\}$

Receive$(\langle \text{View-Change}, v, P, j\rangle_{\sigma_j})_i$ $(j \neq i)$

Eff:

let $m = \langle \text{View-Change}, v, P, j\rangle_{\sigma_j}$

if $v \geq view_i \wedge \textit{correct-view-change}(m, v, j)$ then

$in_i := in_i \cup \{m\}$

Send-New-View$(v, V)_i$

Pre:

$primary(v) = i \wedge v \geq view_i \wedge v > 0 \wedge V \subseteq in_i \wedge |V| = 2f + 1 \wedge \neg \textit{has-new-view}(v, i) \wedge \exists R : (|R| = 2f + 1 \wedge \forall k \in R : (\exists P : (\langle \text{View-Change}, v, P, k\rangle_{\sigma_k})))$

Eff:

$view_i := v$

let $O = \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_i} | \exists v' : \textit{last-prepared}(m, v', n, \textit{merge-P}(V))\}$

$N = \{\langle \text{Pre-Prepare}, v, n, null\rangle_{\sigma_i} | n < \textit{max-n}(O) \wedge \not\exists v', m, n : \textit{last-prepared}(m, v', n, \textit{merge-P}(V)\}$

$m = \langle \text{New-View}, v, V, O, N\rangle_{\sigma_i}$

$seqno_i := \textit{max-n}(O)$

$in_i := in_i \cup O \cup N \cup \{m\}$

$out_i := \{m\}$

Receive$(\langle \text{New-View}, v, X, O, N\rangle_{\sigma_j})_i \ (j \neq i)$

Eff:

let $m = \langle \text{New-View}, v, X, O, N\rangle_{\sigma_j}$

$P = \{\langle \text{Prepare}, v, n', D(m'), i\rangle_{\sigma_i} | \langle \text{Pre-Prepare}, v, n', m'\rangle_{\sigma_j} \in (O \cup N)\}$

if $v > 0 \wedge v \geq \mathit{view}_i \wedge \mathit{correct\text{-}new\text{-}view}(m, v) \wedge \neg \mathit{has\text{-}new\text{-}view}(v, i)$ then

$\mathit{view}_i := v$

$\mathit{in}_i := \mathit{in}_i \cup O \cup N \cup \{m\} \cup P$

$\mathit{out}_i := P$

Relay-Pre-Prepare$(m, v, n)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}} \in \mathit{in}_i \wedge \langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}} \notin \mathit{out}_i \wedge n > \mathit{last\text{-}exec}_i \wedge \mathit{in\text{-}v}(v, i)$

Eff:

$\mathit{out}_i := \mathit{out}_i \cup \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}}\}$

Drop-Pre-Prepare$(m, v, n)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}} \in \mathit{out}_i \wedge (n \leq \mathit{last\text{-}exec}_i \vee \neg \mathit{in\text{-}v}(v, i))$

Eff:

$\mathit{out}_i := \mathit{out}_i \setminus \{\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}}\}$

Relay-Prepare$(m, v, n, j)_i$

Pre:

$\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_{primary(v)}} \in \mathit{out}_i \wedge \langle \text{Prepare}, v, n, D(m), j\rangle_{\sigma_j} \in \mathit{in}_i \wedge \langle \text{Prepare}, v, n, D(m), j\rangle_{\sigma_j} \notin \mathit{out}_i \wedge \langle \text{Commit}, v, n, D(m), j\rangle_{\sigma_j} \notin \mathit{out}_i \wedge n > \mathit{last\text{-}exec}_i \wedge \mathit{in\text{-}v}(v, i)$

Eff:

$\mathit{out}_i := \mathit{out}_i \cup \{\langle \text{Prepare}, v, n, D(m), j\rangle_{\sigma_j}\}$

Drop-Prepare$(m, v, n, j)_i$

Pre:

$\langle \text{Prepare}, v, n, D(m), j\rangle_{\sigma_j} \in \mathit{out}_i \wedge (\langle \text{Commit}, v, n, D(m), j\rangle_{\sigma_j} \in \mathit{out}_i \vee n \leq \mathit{last\text{-}exec}_i \vee \neg \mathit{in\text{-}v}(v, i))$

Eff:

$\mathit{out}_i := \mathit{out}_i \setminus \{\langle \text{Prepare}, v, n, D(m), j\rangle_{\sigma_j}\}$

Relay-Commit$(m, v, n, j)_i$

Pre:

$\langle$Pre-Prepare$, v, n, m\rangle_{\sigma_{primary(v)}} \in out_i \wedge \langle$Commit$, v, n, D(m), j\rangle_{\sigma_j} \in in_i \wedge \langle$Commit$, v, n, D(m), j\rangle_{\sigma_j} \notin out_i \wedge n > last\text{-}exec_i \wedge in\text{-}v(v, i)$

Eff:

$out_i := out_i \cup \{\langle$Commit$, v, n, D(m), j\rangle_{\sigma_j}\}$

Drop-Commit$(m, v, n, j)_i$

Pre:

$\langle$Commit$, v, n, D(m), j\rangle_{\sigma_j} \in out_i \wedge (n \leq last\text{-}exec_i \vee \neg in\text{-}v(v, i))$

Eff:

$out_i := out_i \setminus \{\langle$Commit$, v, n, D(m), j\rangle_{\sigma_j}\}$

Relay-View-Change$(v, j)_i$

Pre:

$\exists P : \langle$View-Change$, view_i, P, i\rangle_{\sigma_i} \in out_i \wedge \exists P' : (\langle$View-Change$, v, P', j\rangle_{\sigma_j} \in in_i \wedge \langle$View-Change$, v, P', j\rangle_{\sigma_j} \notin out_i) \wedge v \geq view_i$

Eff:

$out_i := out_i \cup \{\langle$View-Change$, v, P, j\rangle_{\sigma_j}\}$

Drop-View-Change$(v, j)_i$

Pre:

$\nexists P : \langle$View-Change$, view_i, P, i\rangle_{\sigma_i} \in out_i \vee (\exists P' : \langle$View-Change$, v, P', j\rangle_{\sigma_j} \in out_i \wedge (v < view_i \vee has\text{-}new\text{-}view(v, i)))$

Eff:

$out_i := out_i \setminus \{\langle$View-Change$, v, P, j\rangle_{\sigma_j}\}$

Relay-New-View$(v)_i$

Pre:

$\exists X, O, N, j : (\langle$New-View$, v, X, O, N\rangle_{\sigma_j} \in in_i \wedge \langle$New-View$, v, X, O, N\rangle_{\sigma_j} \notin out_i) \wedge in\text{-}v(v, i)$

Eff:

$out_i := out_i \cup \{\langle$New-View$, v, X, O, N\rangle_{\sigma_j}\}$

Drop-New-View$(v)_i$

Pre:

$\exists X, O, N, j : \langle \text{New-View}, v, X, O, N \rangle_{\sigma_j} \in out_i \wedge v < view_i$

Eff:

$out_i := out_i \setminus \{\langle \text{New-View}, v, X, O, N \rangle_{\sigma_j}\}$

Pre-Send$(M)_i$

Pre:

$\exists m, v, n : (prepared\text{-}certificate(m, v, n, out_i) = M \wedge B_i(\langle m, v, n \rangle) \neq M$

Eff:

for each $j \in (\mathcal{R} \setminus \{i\})$ do

$t_i(\langle m, v, n \rangle, j) := clock_i + mutation_0(\langle m, v, n \rangle, M, i, j) \times \Delta$

$B_i(\langle m, v, n \rangle) := M$

Send$(M, j)_i$ $(j \neq i)$

Pre:

$\exists m, v, n : (B_i(\langle m, v, n \rangle) = M \wedge t_i(\langle m, v, n \rangle, j) \leq clock_i) \wedge M \neq \{\}$

Eff:

$t_i(\langle m, v, n \rangle, j) := clock_i + mutation(\langle m, v, n \rangle, M, i, j) \times \Delta$

Pre-Send$(M)_i$

Pre:

$M = \{m = \langle \text{View-Change}, v, P, k \rangle_{\sigma_k} | m \in out_i\} \wedge B_i(\text{View-Change}) \neq M$

Eff:

for each $j \in (\mathcal{R} \setminus \{i\})$ do

$t_i(\text{View-Change}, j) := clock_i + mutation_0(\text{View-Change}, M, i, j) \times \Delta$

$B_i(\text{View-Change}) := M$

Send$(M, j)_i$ $(j \neq i)$

Pre:

$B_i(\text{View-Change}, j) = M \wedge t_i(\text{View-Change}, j) \leq clock_i \wedge M \neq \{\}$

Eff:

$t_i(\text{View-Change}, j) := clock_i + mutation(\text{View-Change}, M, i, j) \times \Delta$

$\text{Pre-Send}(M)_i\ (j \neq i)$

Pre:

$M = \{m = \langle \text{New-View}, v, X, O, N \rangle_{\sigma_k} | m \in out_i\} \wedge B_i(\text{New-View}) \neq M$

Eff:

for each $j \in (\mathcal{R} \setminus \{i\})$ do

$t_i(\text{New-View}, j) := clock_i + mutation_0(\text{New-View}, M, i, j) \times \Delta$

$B_i(\text{New-View}) := M$

$\text{Send}(M, j)_i\ (j \neq i)$

Pre:

$B_i(\text{New-View}) = M \wedge t_i(\text{New-View}, j) \leq clock_i \wedge M \neq \{\}$

Eff:

$t_i(\text{New-View}, j) := clock_i + mutation(\text{New-View}, M, i, j) \times \Delta$

B.2.4 *Mutações*

procedure $early_0(ch, M, i, j)$

if $fresh(B_i(ch), M, i)$ then

return 0

else

return 1

procedure $early(ch, M, i, j)$

return 1

$min\text{-}v(M) \equiv min(\{v|\langle \text{Pre-Prepare}, v, n, m\rangle_{\sigma_k} \in M\} \cup \{v|\langle \text{Prepare}, v, n, D(m), k\rangle_{\sigma_k} \in M\} \cup \{v|\langle \text{Commit}, v, n, D(m), k\rangle_{\sigma_k} \in M\} \cup \{v|\langle \text{View-Change}, v, P, k\rangle_{\sigma_k} \in M\} \cup \{v|\langle \text{New-View}, v, V, O, N\rangle_{\sigma_{primary(v)}} \in M\})$

procedure $centralized_0(ch, M, i, j)$
    if $i = primary(min\text{-}v(M))$ then
        if $fresh(B_i(ch), M, i) \vee maj(B_i(ch), M)$ then
            return 0
        else
            return 1
    else if $j = primary(min\text{-}v(M))$ then
        if $fresh(B_i(ch), M, i)$ then
            return 0
        else
            return 1
    else if $ch = \text{View-Change}$ then
        return 1
    else
        return ∞
procedure $centralized(ch, M, i, j)$
    return 1

procedure $ring_0(ch, M, i, j)$
    if $fresh(B_i(ch), M, i) \vee maj(B_i(ch), M)$ then
        return $((j - i) \bmod |\mathcal{R}|) - 1$
    else
        return $((j - i) \bmod |\mathcal{R}|)$
procedure $ring(ch, M, i, j)$
    return $|\mathcal{R}| - 1$

$c_i : \mathcal{CH} \to \mathbf{N}$, initially $\forall ch \in \mathcal{CH} : c_i(ch) = 0$

procedure $gossip_0(ch, M, i, j)$

  let $j = u_i(ch)[ind]$

    $c_i(ch) := \lceil c_i(ch) \rceil$

    return $\lfloor((ind - fanout \times c_i(ch)) \bmod |\mathcal{R}|) \div fanout\rfloor$

procedure $gossip(ch, M, i, j)$

  let $j = u_i(ch)[ind]$

    $c_i(ch) := c_i(ch) + \frac{1}{fanout}$

    return $\lfloor((ind - fanout \times \lceil c_i(ch) \rceil) \bmod |\mathcal{R}|) \div fanout\rfloor + 1$

C

RESULTADOS DA AVALIAÇÃO EXPERIMENTAL

	PBFT	Early	Centralized	Ring	Gossip
97	192	192	5	2,3	73
193	384	384	5	2,3	101
289	576	576	5	2,3	110
385	768	768	5	2,3	100

Tabela 11: Número médio de mensagens recebidas por processo

	PBFT	Early	Centralized	Ring	Gossip
97	0,5	6,1	7,7	9,4	9,8
193	0,5	10,8	14,3	17,7	18,3
289	0,5	15,5	20,8	26,0	27,0
385	0,5	20,2	27,3	34,4	35,3

Tabela 12: Tamanho médio das mensagens (KB)

	PBFT	Early	Centralized	Ring	Gossip
97	161	611	516	1287	250
193	324	1588	1367	4574	532
289	540	2864	2763	9855	788
385	990				1004

Tabela 13: Latência média de cada pedido