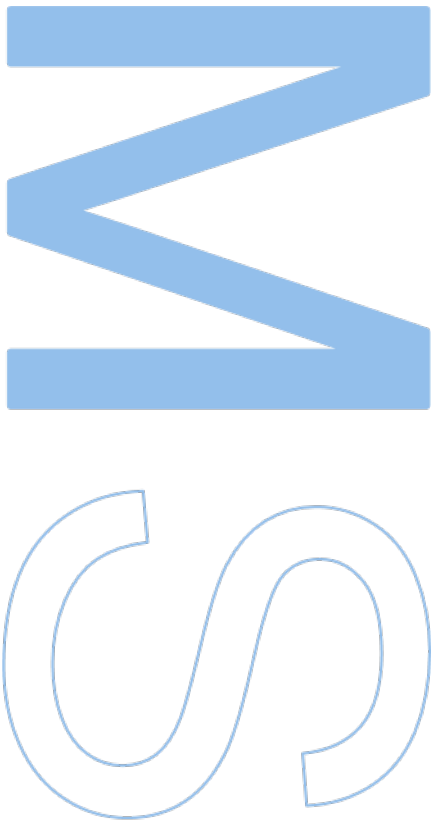
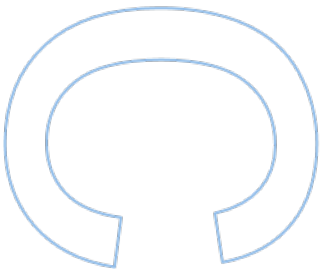
Quando a probabilidade diminui – Consenso probabilístico em ambientes com faltas



Beatriz Meireles
Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2023

Orientador
João Paulo da Conceição Soares, Professor Auxiliar Convidado,
Faculdade de Ciências da Universidade do Porto




U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Agradecimentos

Quero expressar os meus agradecimentos ao meu orientador Professor João Soares, que durante todos este tempo da realização da minha dissertação me ajudou bastante a nível académico e me proporcionou uma experiência enriquecedora onde aprendi imenso no que toca a este tema.
Queria ainda agradecer aos meus pais e à minha família que sempre me incentivou a ser melhor e a lutar por aquilo que desejo, proporcionaram-me as condições necessárias para evoluir o máximo possível a nível académico.
Por último quero agradecer aos meus amigos que tiveram comigo durante todo este caminho e me ajudaram a superar os desafios que foram aparecendo.

Resumo

Os protocolos de consenso, garantem que um grupo de máquinas concordem numa mesma decisão, realizando as mesmas operações. O problema do consenso é muito falado em *computer science* como sendo um dos problemas mais clássicos.
Um desafio que as redes distribuídas enfrentam é assegurar que todos os nós cheguem ao mesmo estado, o estado deve apenas mudar de acordo com um conjunto de regras e que uma vez alterado todos os nós não faltosos eventualmente alcançam este novo estado.
Esse conjunto de regras é especificamente o protocolo de consenso, que para além de assegurar que todos os nós não faltosos chegam ao mesmo estado, também tem certas propriedades que permitem provar que o protocolo é correto.
O desafio mais natural sobre este tópico é fazer com que o consenso seja o mais rápido possível e ao mesmo tempo o mais seguro. Para isso ser possível foi criado o primeiro protocolo *atomic broadcast* prático assíncrono proposto por Andrew Miller et al. chamado HoneyBadgerBFT [6] que demonstra uma performance bastante favorável e se encontra nos protocolos resistentes a faltas bizantinas.
Com o passar do tempo surgiram outros protocolos semelhantes e que conseguiram melhorar o HoneyBadger, é aí que entra o Dumbo [8] (com dois protocolos, Dumbo1 e Dumbo2) proposto por Bingyong Guo et al. que se basea no HoneyBadger (HB) mas consegue ter melhorias tanto a nível de latência quanto a nível de taxa de transferência.
Os autores descrevem os protocolos e sub-protocolos do Dumbo e as suas várias especificações como por exemplo, as faltas que consegue suportar, a sua resiliência, entre outros.
Nesta dissertação, esses protocolos e sub-protocolos foram analisados exaustivamente identificando possíveis vulnerabilidades caso um ou mais nós não respeitem o protocolo. O resultado de cada vulnerabilidade foi testado na prática e comparado com o resultado esperado tendo em conta a especificação do protocolo.

Abstract

Consensus protocols ensure that a group of machines agree on the same decision, performing the same operations. The consensus problem is often talked about in *computer science* as being one of the most classic problems.
A challenge that distributed networks face is to ensure that all nodes arrive at the same state, the state must only change according to a set of rules and once is changed all non-faulting nodes eventually reach this new state.
This set of rules is specifically the consensus protocol, which in addition to ensuring that all non-faulting nodes arrive at the same state, also has certain properties that allow proving that the protocol is correct.
The most natural challenge about this topic is to make consensus as fast as possible and at the same time as safe as possible. For this to be possible, the first practical asynchronous *atomic broadcast* protocol was proposed by Andrew Miller et al. and it's called HoneyBadgerBFT [6], HoneyBadger (HB) demonstrates a very favorable performance and is a Byzantine fault resistant protocol.
Over time, other similar protocols emerged that managed to improve HoneyBadger, that's where Dumbo [8] (with two protocols, Dumbo1 and Dumbo2) proposed by Bingyong Guo et al. came in, dumbo is based on HB but manages to improve both latency and *throughput*.
The authors specify Dumbo's protocols and sub-protocols and its various specifications such as the faults it can withstand and its resilience.
In this dissertation, these protocols and sub-protocols were exhaustively analyzed identifying possible vulnerabilities if one or more nodes do not respect the protocol. The result of each vulnerability was tested in practice and compared with the expected result given the protocol specification.

Conteúdo

Agradecimentos i

Resumo iii

Abstract v

Conteúdo ix

Lista de Tabelas xii

Lista de Figuras xiv

Acrónimos xvi

1 Introdução 1

2 Background 3

2.1 Consenso 3

2.2 Protocolos de Consenso 4

2.2.1 Modelo de Comunicação 4

2.2.2 Modelo de Faltas 5

2.2.3 Protocolos de Consenso Bizantinos 6

3 Estado da Arte 9

3.1 *From Consensus to Atomic Broadcast* . . . 9
3.1.1 Arquitetura . . . 10
3.1.2 *Multi-valued Consensus* . . . 12
3.1.3 *Vector Consensus* . . . 14
3.1.4 *Atomic Broadcast* . . . 14
3.2 HoneyBadger . . . 15
3.2.1 *Atomic Broadcast* . . . 16
3.2.2 Construção do protocolo HoneyBadgerBFT . . . 17
3.3 Dumbo . . . 20
3.3.1 *Asynchronous Binary Agreement* . . . 21
3.3.2 Esquema de Encriptação *Threshold* . . . 22
3.3.3 *Multi-valued Byzantine Agreement* . . . 22
3.3.4 $(1,K,\epsilon) - CommitteeElection(CE)$ . . . 22
3.3.5 *Consistent Broadcast* . . . 23
3.3.6 Dumbo1-ACS . . . 23
3.3.7 Dumbo2-ACS . . . 24
3.3.8 *Atomic Broadcast* . . . 26

4 Análise do Problema **27**
4.1 Desenho da Análise . . . 27
4.1.1 Atomic Broadcast . . . 27
4.1.2 ACS . . . 28
4.1.3 Modificações . . . 29
4.2 Análise do Código . . . 29
4.2.1 Atomic Broadcast . . . 30
4.2.2 MVBA . . . 32
4.2.3 CBC . . . 34

5 Testes e Validação **37**

5.1 *Throughput e Latência* 37
5.2 Injeção de faltas 38
5.2.1 *Cluster* com 4 nós 38
5.2.2 *Cluster* com 8 nós 43
5.2.3 *Cluster* com 16 nós 46
5.2.4 Análise 50

6 Conclusões **53**

6.1 Trabalho Futuro 54

Bibliografia **55**

Lista de Tabelas

5.1 *Throughput* e latência sem injeção de faltas 38

5.2 *Throughput* e latência com injeção de faltas - PRBC 39

5.3 *Throughput* e latência com injeção de faltas - PRBC(2) 39

5.4 *Throughput* e latência com injeção de faltas - CBC(send) 40

5.5 *Throughput* e latência com injeção de faltas - CBC(echo) 40

5.6 *Throughput* e latência com injeção de faltas - CBC(Final) 41

5.7 *Throughput* e latência com injeção de faltas - CBC(Entrega) 41

5.8 *Throughput* e latência com injeção de faltas - Coin 41

5.9 *Throughput* e latência com injeção de faltas - ABA(1) 42

5.10 *Throughput* e latência com injeção de faltas - ABA(2) 42

5.11 *Throughput* e latência sem injeção de faltas 43

5.12 *Throughput* e latência com injeção de faltas - PRBC(2) 44

5.13 *Throughput* e latência com injeção de faltas - CBC 45

5.14 *Throughput* e latência com injeção de faltas - COIN 45

5.15 *Throughput* e latência com injeção de faltas - ABA(1) 46

5.16 *Throughput* e latência com injeção de faltas - ABA(2) 46

5.17 *Throughput* e latência sem injeção de faltas 47

5.18 *Throughput* e latência com injeção de faltas - PRBC(2) 47

5.19 *Throughput* e latência com injeção de faltas - CBC 48

5.20 *Throughput* e latência com injeção de faltas - ABA(1) 49

5.21 *Throughput* e latência com injeção de faltas - ABA(2) 49

5.22 Modificações com 4 nós 50
5.23 Modificações com 8 nós 51
5.24 Modificações com 16 nós 51

Lista de Figuras

2.1 Protocolo de *Agreement do PBFT* 7

3.1 Propriedades dos protocolos propostos 10

3.2 Arquitetura do protocolo. 11

3.3 Explicação do Multi-valued Consensus 12

3.4 Explicação do Multi-valued Consensus 13

3.5 HoneyBadger 16

3.6 Asynchronous Common Subset 19

3.7 HoneyBadgerBFT 20

3.8 Dumbo 21

3.9 CBC 23

3.10 Dumbo1 24

3.11 Dumbo2 25

3.12 AtomicBroadcast 26

4.1 AtomicBroadcast 28

4.2 Modificações no protocolo PRBC 31

4.3 Modificações no protocolo PRBC(2) 31

4.4 Modificações no protocolo PRBC(3) 32

4.5 Modificações no protocolo MVBA - *Consistent Broadcast* 33

4.6 Modificações no protocolo MVBA(2) 33

4.7 Modificações no protocolo MVBA(3) - *Asynchronous Binary Agreement* 34

4.8 Modificações no protocolo MVBA(4) 34
4.9 Modificações no protocolo CBC(1) 35
4.10 Modificações no protocolo CBC(2) 35
4.11 Modificações no protocolo CBC(3) 36
4.12 Modificações no protocolo CBC(4) 36

5.1 Resultados das modificações nos diferentes subprotocolos do Dumbo (n=4; f=1) . 43
5.2 Resultados das modificações nos diferentes subprotocolos do Dumbo (n=8; f=2) . 46
5.3 Resultados das modificações nos diferentes subprotocolos do Dumbo (n=16; f=5) 49

Acrónimos

BFT Bizantine Fault Tolerante - Tolerante a faltas bizantinas

PBFT Pratical Bizantine Fault Tolerante

DCC Departamento de Ciência de Computadores

FCUP Faculdade de Ciências da Universidade do Porto

RBC Reliable Broadcast

ABA Asynchronous Binary Agreement

BC Binary Consensus

MVC Multi-valued Consensus

VC Vector Consensus

ACS Asynchronous Common Subset

ABC Atomic Broadcast

MVBA Multi-valued Validated Agreement

TPKE Threshold Encryption Scheme

BA Binary Agreement

CE Committe election

HB HoneyBadger

id identificador

PRBC Provable Reliable Broadcast

CBC Consistent Broadcast

CE Committee Election

Capítulo 1

Introdução

Protocolos de consenso, em sistemas distribuídos, é um tópico bastante estudado e com muita aplicabilidade prática, têm um papel crucial na construção de sistemas distribuídos de larga escala. Permitem essencialmente que um conjunto de máquinas chegue a um acordo em relação a um valor, ou seja, sobre uma mesma decisão.
Um exemplo prático, e bastante usado nos dias de hoje visto ser um dos tópicos mais falados, é o consenso estar no núcleo/centro das *blockchains*, usa-se por exemplo para ordenar as transações mesmo que alguma parte seja faltosa. Tem que se garantir para isso duas propriedades: *safety*, garantir que nada malicioso acontece, e *liveness*, onde se verifica o progresso, se houver um pedido este deve ser atendido [10].
Estes protocolos podem lidar com faltas *fail-stop*, nas quais as máquinas falham porque param de responder, ou seja, param de participar do protocolo, ou lidar com falhas bizantinas, nas quais as máquinas podem apresentar comportamento arbitrário, ou seja, podem continuar a participar da entrega do protocolo, com por exemplo, resultados errados que podem corromper o estado da aplicação.

Os protocolos de consenso tolerantes a faltas têm ainda duas abordagens possíveis: determinística e probabilística. Em protocolos determinísticos, como o PBFT [5] e BFT-SMART [3], normalmente todos os elementos do grupo são conhecidos 'a priori', um dos membros do grupo tem o papel de líder, sendo ele o único responsável por propor valores para votar. Em protocolos probabilísticos, o tipo de protocolo estudado nesta dissertação, como o Honey Badger [6], Dumbo [8] e FPC [9], não há necessidade de conhecer todos os elementos do grupo 'a priori', não há noção de líder, pois cada membro pode propor valores para votação, e o protocolo não tem um número pré-definido de fases. Ao contrário dos determinísticos, que requerem períodos de sincronia na comunicação para garantir a terminação, os probabilísticos eliminam essa necessidade garantindo a terminação mesmo na ausência de sincronia na comunicação.

Focando nos protocolos probabilísticos resistentes a falhas bizantinas e depois de estudados exaustivamente foi escolhido o **Dumbo** [8], inicialmente o objetivo era testar as vulnerabilidades

do HB, mas a atualização mais recente da implementação foi há 5 anos atrás e deu bastantes problemas então foi decidido trocar para uma mais recente, neste caso o Dumbo que é baseado no HB e teóricamente conseguiu melhorar certos parâmetros. Isto porque, propôs um novo protocolo ACS (Asynchronous Common Subset) que utiliza o protocolo MVBA trazendo mais eficiência ao ACS se usado com cuidado.

O objetivo desta dissertação foi estudar e compreender diversos protocolos probabilísticos resistentes a faltas bizantinas e compara-los antes de escolher um. Depois disso e de escolhido então o **Dumbo**, foi analisado exaustivamente de maneira a se conseguir perceber de forma inequívoca a sua implementação, seguidamente veio a identificação de possíveis vulnerabilidades caso um ou mais nós não respeitassem o protocolo e por último testou-se na prática o resultado das vulnerabilidades identificadas e comparou-se o resultado prático obtido com o resultado esperado tendo em conta a especificação do protocolo.
Dito isto, conseguimos validar a execução do Dumbo e perceber quais as suas principais vulnerabilidades, entendendo a fundo como é que elas afetam a performance do protocolo e percebemos no final que ele nem sempre se comporta de acordo com as suas especificações em todo o protocolo e sub-protocolos.

No capítulo 2, são apresentados diversos conceitos que tiveram que ser bem entendidos e estudados antes da análise real dos protocolos e sub-protocolos. No capítulo 3, são explicados os protocolos de consenso probabilísticos que foram analisados por serem necessários na realização desta dissertação. Seguidamente, no capítulo 4, encontra-se a análise exaustiva do protocolo escolhido (o Dumbo) e dos seus sub-protocolos e a identificação das possíveis vulnerabilidades. Consequentemente, no capitulo 5, demonstra-se os testes/resultados das vulnerabilidades, a comparação dos mesmos e que métodos foram utilizados na prática. Por último, no capítulo 6 encontra-se as conclusões retiradas depois da realização prática e teórica do problema e trabalhos futuros que poderão existir.

Capítulo 2

Background

Este capítulo apresenta alguns fundamentos do trabalho feito e serve para que haja um melhor entendimento sobre terminologias e expressões usadas nos próximos capítulos. Começa por ser explicado o conceito de consenso que é utilizado nos protocolos estudados e também os modelos existentes, tanto a nível de faltas como de comunicação.
Para ser possível estudar a fundo os protocolos de consenso bizantinos e os seus sub-protocolos foi necessário uma pesquisa intensiva sobre o *background* dos mesmos e conseguir perceber tanto as semelhanças quanto as diferenças entre os variados modelos.

2.1 Consenso

O contexto desta dissertação gira em torno de um conceito chamado **consenso**. O consenso consiste num conjunto de participantes concordar ou decidir sobre um valor (ou um conjunto de valores). Pode-se atingir o consenso quando a maioria dos membros que participam da votação votam na mesma decisão ou então quando um único líder diz aos participantes o que fazer.
Para haver esta votação e o consenso ser alcançado tem que haver mais do que um participante, cada um dos participantes votar apenas numa das opções disponíveis e depois da votação não haver alteração dos votos.
Outro conceito que surge é o **consenso distribuído**, os participantes não estão fisicamente juntos e comunicam com mensagens. Resumidamente, os votos são enviados por mensagem, alguém entrega e conta os votos e os participantes recebem os resultados também por mensagem. Sem o consenso não é possível que o valor escolhido seja compartilhado por todos os membros da rede, precisa-se que que todos entrem em acordo ou pelo menos cheguem a uma maioria.

2.2 Protocolos de Consenso

Depois de entendido o conceito de **consenso** é preciso perceber o que são **protocolos de consenso**. Têm como objetivo garantir que um conjunto de máquinas chega a acordo sobre uma mesma decisão onde todos executam as mesmas operações.
Tipicamente, um elemento do grupo, que é o líder, propõe um valor ou decisão para consenso e todos os elementos do grupo participam nas votações de forma a decidir sobre essa proposta. Uma proposta é aceite se a maioria do grupo aceitar a mesma, sendo rejeitada caso contrário. A decisão aceite é executada por todos os membros (máquinas) do grupo. Estes protocolos são cada vez mais importantes hoje em dia, sendo usados não só para melhorar o desempenho das redes das criptomoedas [10] que estão em ascensão neste momento, como também por gestores de recursos distribuídos, gestores de membros de grupo, entre outros que estão na base das infraestruturas de *cloud* [1].

2.2.1 Modelo de Comunicação

Dentro dos protocolos de consenso podemos encontrar diversos modelos. No caso dos modelos de comunicação há vários conceitos que vão ser abordados nas secções seguintes, nomeadamente a **sincronia da rede**.
A sincronia estabelece o nível de coordenação entre os vários processos [10]. Y. Xiao et al. tanto em [10] quanto em [11] especifica 3 níveis de sincronia de rede, nomeadamente síncrona, parcialmente síncrona e assíncrona.
Défago et al. tem uma abordagem com 5 níveis de sincronia para dois parâmetros que são abordados no artigo, sendo eles o *process speed interval* e o *communication delay* [18]:

- Limite superior conhecido que se mantém sempre
- Limite superior desconhecido que se mantém sempre
- Limite superior desconhecido que eventualmente se mantém para sempre
- Limite superior conhecido que eventualmente se mantém para sempre
- Não há nenhum limite sobre o valor do parâmetro

Em sistemas onde o *process speed interval*, que é a diferença entre a velocidade do processo mais lento com o mais rápido, e o *communication delay*, que é o intervalo de tempo entre o envio e a receção das mensagens, cumprem com o primeiro ponto falado em cima, Défago et al. intitula de **sistema síncrono** [18].
Y. Xiao et al. intitula de **sistema síncrono**, um sistema onde as diversas operações dos processos têm rondas que são coordenadas entre si e onde existem restrições de tempo. Há também uma execução de operações igualitária de todos os processos e a entrega das mensagens é assegurada dentro de um atraso fixos [10] [11].

Conseguimos daí subentender então que um modelo de comunicação síncrono tem um limite superior conhecido para o atraso de entrega das mensagens, ou seja, a latência de entrega tem um valor máximo conhecido e as mensagens são sempre entregues e nunca descartadas.

Por outro lado, em sistemas onde o *process speed interval* e o *communication delay* não têm nenhum limite associado, que é o caso do último ponto que se encontra anteriormente, são então intitulados de **sistemas assíncronos** por Défago et al [18].
Y. Xiao et al. aborda um **sistema assíncrono**, onde as diversas operações dos processos não são coordenadas, ou seja, cada processo executa as suas próprias operações sem haver garantia de entrega [10] [11].
Em resumo, num modelo de comunicação assíncrono, o atraso na entrega das mensagens tem um limite superior desconhecido, ou seja, a latência tem um valor máximo desconhecido e as mensagens podem ser descartadas.

Por último, e entre os dois extremos dos pontos referidos anteriormente por Défago et al. encontram-se vários modelos de sistemas **parcialmente síncronos** [18].
Y. Xiao et al. fala de redes **parcialmente síncronas** tendo elas uma flexibilidade na coordenação das operações dos processos onde a entrega das mensagens é garantida mas com uma quantidade incerta de atrasos, ou seja a entrega é assegurada embora possa não ser em tempo hábil [10] [11].

Dentro dos modelos de comunicação há também os conceitos de **modelo de comunicação confiável**, onde as mensagens nunca são corrompidas, ou seja, o conteúdo da mensagem é preservado durante a comunicação, e **modelo de comunicação não confiável**, onde as mensagens podem ser corrompidas, ou seja, o conteúdo da mensagem pode ser alterado durante a comunicação.

Défago et al. define então dois tipos de canais, os ***Reliable Channels*** que garantem que se um processo que é correto envia uma mensagem a outro processo correto então quer dizer que o último vai eventualmente receber a mensagem, e os ***Lossy Channels*** onde pode haver perdas por colisões de rede, interferências, entre outros [18].

2.2.2 Modelo de Faltas

Dentro dos protocolos de consenso há diferentes abordagens dependendo do modelo de faltas que se vai lidar.
Défago et al. indica 4 tipos de faltas, as faltas ***crash***, quando um processo trava e deixa de fazer qualquer tipo de atividade, as faltas por **omissão**, onde um processo omite algumas ações que deve fazer, as **temporais**, em que há uma violação temporal (não existe em sistemas assíncronos) e por último as **bizantinas** que podem ter comportamentos arbitrários [18].

Seguidamente vai ser apenas abordado dois tipos de faltas, que já abrangem as anteriormente referidas e são os conceitos mais falados em protocolos de consenso no que toca a vulnerabilidades.

2.2.2.1 Modelo de faltas *Fail-stop*

As faltas, podem ser *'fail-stop'*, em que as máquinas falham deixando de responder (e portanto deixando de participar no protocolo).
No caso do modelo *Fail-stop*, os participantes faltosos ficam indisponíveis, ou seja, não participam do protocolo. Os participantes tanto podem estar disponíveis quanto indisponíveis no momento seguinte.
Resumidamente, neste modelo os participantes e links falham ao ficarem indisponíveis (as mensagens são ou não entregues) e os participantes não faltosos respeitam sempre o protocolo.

2.2.2.2 Modelo de faltas Bizantinas ou Arbitrárias

Nas faltas bizantinas as máquinas falham exibindo um comportamento arbitrário, podendo continuar a participar no protocolo, mas dando por exemplo, resultados errados que podem corromper o estado da aplicação.
No modelo bizantino, os participantes faltosos podem apresentar um comportamento arbitrário, ou seja, não seguir o protocolo.
Um exemplo é enviar mensagens/resultados erróneos, participar mais de uma vez em cada votação, modificar os seus votos, entre outros.
Por outras palavras, os participantes e links podem falhar arbitrariamente e as mensagens podem ficar corrompidas durante o trânsito.

Os requisitos dependem sempre do modelo de falta, diferentes protocolos de consenso foram propostos para diferentes modelos de faltas.
Dentro destes modelos de faltas pode haver modelos **baseados em líder** e uma abordagem **sem líder**. No caso dos *fail-stop*, há o Paxos e o Raft [12] [15] como modelos baseados em líder e o Ben-Or [7] sem líder. Nos bizantinos, o PBFT [5] é um exemplo de modelo baseado em líder e sem líder os protocolos HoneyBadger [6] e Dumbo [8].

Quando existe líder, ele é eleito entre os participantes onde todos são candidatos e só ele pode propor valores, todos votam nas propostas recebidas.

2.2.3 Protocolos de Consenso Bizantinos

Esta dissertação foca-se em protocolos de consenso bizantinos e probabilísticos, por consequente há a necessidade de perceber as duas abordagens possíveis e as suas diferenças, sendo elas: determinística e probabilística.

2.2.3.1 Determinísticos

Nos protocolos determinísticos tipicamente todos os elementos do grupo são conhecidos à partida, um dos membros do grupo tem o papel de líder, sendo este o único responsável por propor

valores para votação como acontece por exemplo no PBFT. O processo de votação divide-se essencialmente em 3 fases como podemos ver na figura 2.1:

- o líder propõe o valor (pre-prepare)
- cada elemento do grupo envia o valor proposto e a sua decisão para todos os membros (prepare)
- cada elemento do grupo segue a decisão da maioria, definida no passo anterior (commit)

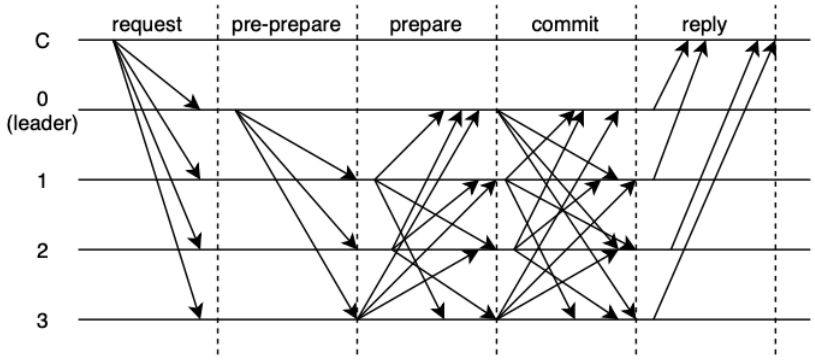


Figura 2.1: Protocolo de *Agreement* do PBFT, figura retirada de [5].

2.2.3.2 Probabilísticos

Nos protocolos probabilísticos (ou não determinísticos), tipicamente, não há necessidade de se conhecer todos os elementos do grupo, cada membro do grupo pode propor valores (não existindo a noção de líder) e o protocolo não tem um número de fases pré-definida, parando quando uma possível maioria dos membros do grupo tenham participado na votação.
Estes protocolos costumam usar mecanismos de troca de mensagens epidêmicos ou gossip-based, em que cada membro em cada ronda de troca de mensagens envia mensagens aos membros que conhece, parando o protocolo quando as últimas n rondas não contactaram novos membros.

Os conteúdos apresentados neste capítulo são extremamente necessários para ser possível atingir o objetivo de explorar as vulnerabilidades do protocolo escolhido, o Dumbo [8]. Sem entender as diferenças entre os vários modelos de comunicação e das faltas existentes seria impossível a compreensão total do protocolo e dos sub-protocolos.
No próximo capítulo, estado da arte, são abordados tanto os artigos de protocolos probabilísticos relevantes incluídos no estudo durante a pesquisa quanto o protocolo principal escolhido, o Dumbo.

Capítulo 3

Estado da Arte

Neste Capítulo será discutida a literatura necessária para a realização deste trabalho, são abordados 3 artigos focados em protocolos de consenso probabilísticos resistentes a faltas bizantinas.
A análise começou pelo trabalho proposto por Correia et al. [14] que foi a base de tudo e teve uma importância fulcral. Isto porque, consegue explicar detalhadamente e de maneira simples as propriedades dos protocolos e os diversos sub-protocolos que depois são também usados no HoneyBadger [6] e no Dumbo [8], sem o estudo aprofundado deste artigo considera-se que não seria possível a total compreensão dos dois seguintes.
Esta dissertação foca-se no Dumbo, o protocolo mais recente entre os 3 estudados e que se encontra detalhado na última secção deste Capítulo.

3.1 *From Consensus to Atomic Broadcast*

O artigo "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures"proposto por Correia et al. [14] não fala de um protocolo isolado mas sim de uma *stack* com 3 protocolos resistentes a faltas bizantinas, sendo eles o *multi-valued consensus*, *vector consensus* e o *atomic broadcast*.
Os nós dos protocolos são representados por n e os faltosos por f.
Na figura 3.1 estão identificadas as **propriedades comuns** dos 3 protocolos propostos:

- **Sem assinaturas digitais** construidas com criptografia de chave publica, a criptografia assimétrica usa pares de chaves o que torna a performance mais lenta;
- ***Time-Free*** - São assíncronos ou seja, não fazem nenhuma suposição temporal;
- As decisões são feitas de maneira **descentralizada**, ou seja, sem coordenadores/lideres;
- Por último, têm **resiliência ótima**, visto que suportam f = [(n-1)/3] nós maliciosos no total de n nós. A resiliência de um protocolo defini-se como o número máximo de falhas que o protocolo suporta comportando-se ainda de acordo com as suas especificações.

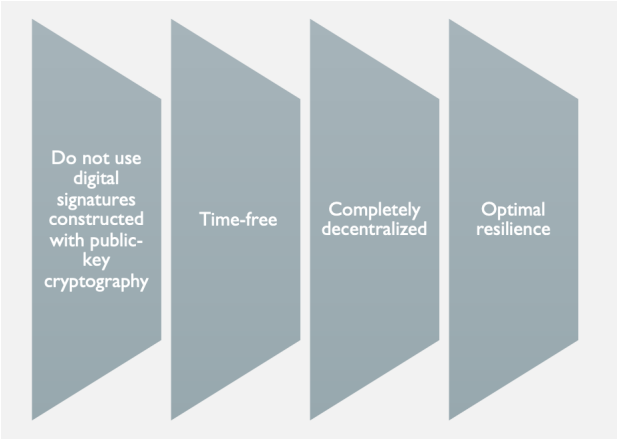


Figura 3.1: Propriedades dos protocolos propostos.

Dos 3 protocolos propostos por Correia et al., conseguimos distinguir 2 tipos de consenso: o ***multi-valued consensus*** onde basicamente há uma decisão em valores que têm tamanho arbitrário e o ***vector consensus*** onde a decisão é feita a partir de um vetor que contem os valores propostos por vários processos. O terceiro protocolo proposto é então o ***atomic broadcast***, um protocolo de comunicação, que faz a entrega das mensagens a todos os processos na mesma ordem, isto para que todos mantenham o mesmo estado.

3.1.1 Arquitetura

Como se pode ver na figura 3.2, Correia et al. apresenta na base da arquitetura **canais confiáveis** que interligam os processos e onde é feita a transmissão confiável das mensagens.

Por cima desses canais confiáveis que conseguem assegurar a transmissão confiável das mensagens estão o ***binary consensus*** e o ***reliable broadcast*** que são também utilizados no HB e no Dumbo.

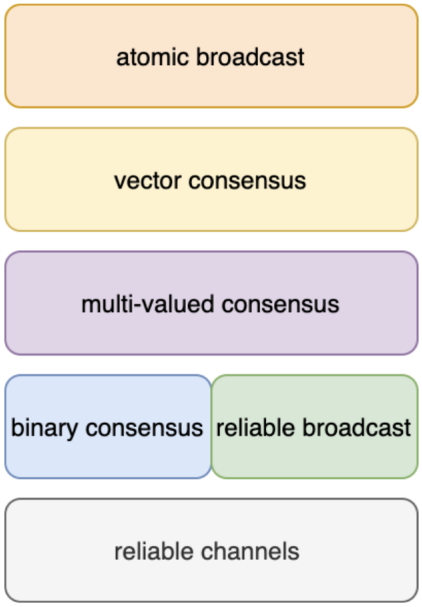


Figura 3.2: Arquitetura do protocolo, figura adaptada do artigo do Correia et al. [14].

Reliable Broadcast

O RBC tal como nome indica, faz a transmissão confiável dos valores, resumidamente garante que todas as mensagens consigam chegar a todos os nós, na ordem correta e sem haver duplicados (mensagens diferentes com o mesmo id).
É apresentado por Correia et al. com as seguintes propriedades:

- **Validação** - Sempre que há a transmissão de uma mensagem por um processo não corrupto então essa mensagem vai ser entregue por algum processo não corrupto;
- ***Agreement*** - Sempre que um dos processos que não é corrupto entrega alguma mensagem, essa mensagem vai ser entregue por todos os processos;
- **Integridade** - "For any identifier ID, every correct process p delivers at most one message M with identifier ID, and if sender(M) is correct then M was previously broadcast by sender(M)" [14], resumidamente o id é único para cada mensagem, não há duas mensagens entregues com o mesmo id.

Binary Consensus

Lado a lado com o RBC temos o BC onde todos os processos recebem um valor de input binário, b $\in$ 0,1. Depois de receberem esse valor, todos devem concordar com o valor de input dos processos.
É apresentado por Correia et al. com as seguintes propriedades:

- **Validação** - Sempre que há uma decisão de todos os nós/processos não corruptos sobre um valor (b), então qualquer nó não corrupto que tenha que decidir, vai decidir esse valor;

- ***Agreement*** - Todos os nós não corruptos decidem o mesmo, ou seja, não existem dois nós corretos que decidem coisas diferentes;
- **Terminação** - Todos os processos corretos decidem;

3.1.2 *Multi-valued Consensus*

A correr em cima do BC e do RBC chega o primeiro protocolo da *stack* proposta por Correia et al. dos 3 protocolos resistentes a faltas bizantinas, o ***Multi-valued Consensus***.
Este protocolo resulta numa decisão sobre valores que têm tamanho arbitrário e foi fulcral para conseguir compreender como o consenso funciona.

É apresentado por Correia et al. com as seguintes propriedades:

- **Validação1** - Se o mesmo valor for proposto por todos os processos, então quando um processo for decidir vai decidir esse valor;
- **Validação2** - Se um processo correto for decidir um certo valor então esse valor foi proposto por algum processo;
- **Validação3** - Se houver algum valor que foi proposto por processos maliciosos, os processos não maliciosos que decidam nunca decidem esse valor;
- ***Agreement*** - Todos os processos não maliciosos que decidam, decidem o mesmo valor;
- **Terminação** - Todos os processos não maliciosos acabam por decidir;

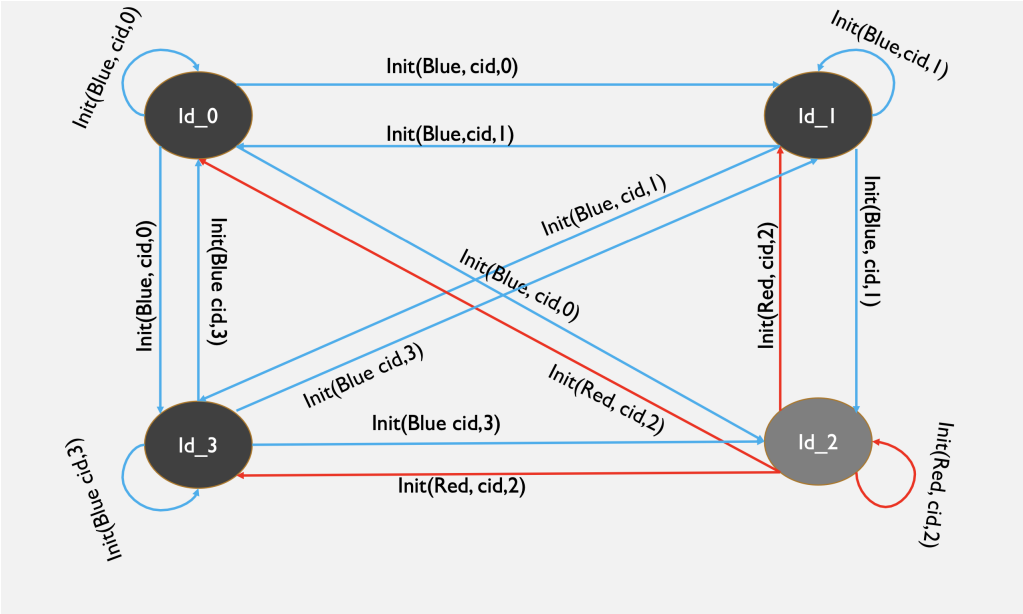


Figura 3.3: Demonstração do Multi-valued Consensus.

Demonstração:

A figura 3.3 mostra, de maneira simplificada e resumida, como funciona o Multi-valued Consensus. Nesta demonstração encontram-se 4 nós (Id_0, Id_1, Id_2, Id_3) que têm que decidir sobre um valor.
Todos os nós transmitem uma mensagem Init através do RBC com um valor. Cada nó espera pela receção de (n-f) mensagens Init, neste caso como são 4 nós (f = 1), espera pela receção de 3 mensagens Init (incluindo a própria).
Os valores propostos ficam guardados num vetor, cada nó tem o seu próprio vetor com os valores guardados (Init(Blue, cid,0); Init(Blue, cid,1); Init(Red, cid,2) e Init(Blue, cid,3)).
O próximo passo é escolher o valor, como o valor proposto é >= (n-2f) (especificação do protocolo para continuar o consenso) , neste caso se for >= 2 esse valor é o escolhido.
Como se consegue ver, Blue é o valor escolhido porque aparece mais do que duas vezes apesar do nó 2 propor o valor Red, o valor escolhido é então guardado em wi que aparece na figura 3.4 (wi = Blue).

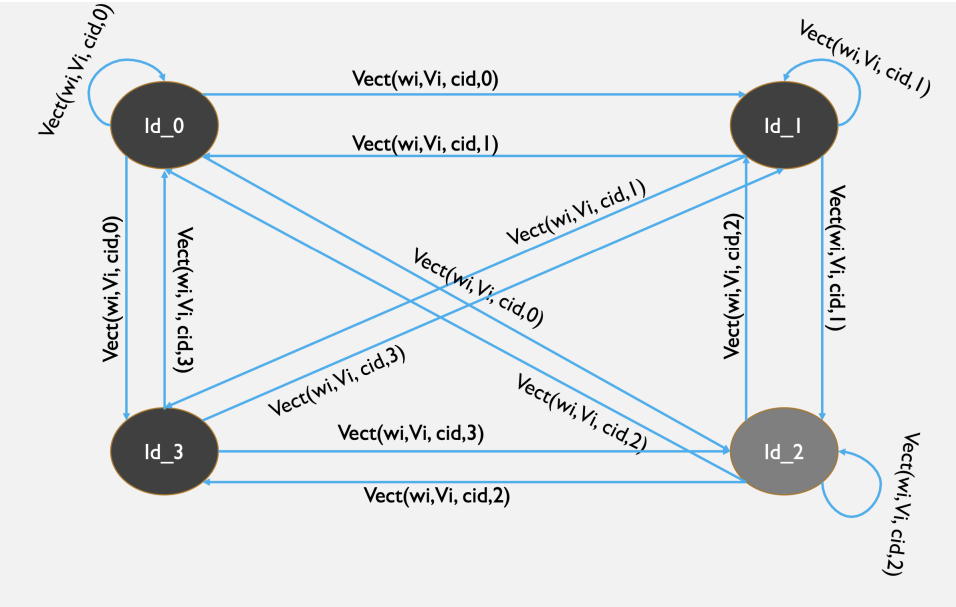


Figura 3.4: Continuação da demonstração do Multi-valued Consensus.

Demonstração (continuação): Na segunda parte do protocolo, como se pode ver na figura 3.4, todos os nós transmitem a mensagem Vect, utilizando o RBC, com o valor escolhido anteriormente (wi) e como vetor Vi que justifica a seleção, porque tem todos os vetores de todos os nós com as suas propostas.
Há então a espera por (n-f) mensagens Vect válidas que contenham o valor escolhido. Como não receberam duas mensagens Vect com valores diferentes w != w' e receberam pelo menos (n-2f) mensagens com w o BC propõe 1 para todos os casos deste exemplo (se houvesse algum que tivesse mensagens diferentes já iria propor 0).
Como todas as (n-2f) mensagens Vect são válidas e têm o valor escolhido , neste caso o Blue que

foi escolhido inicialmente então o Blue foi o valor decidido pelo Multi-valued Consensus.

Estas demonstrações foram importantes para entender como funciona o consenso, os protocolos de consenso que veremos a seguir funcionam de forma semelhante mas com outras propriedades.

3.1.3 *Vector Consensus*

O segundo protocolo de consenso resistente a faltas bizantinas proposto por Correia et al. foi o ***Vector Consensus***. Este protocolo corre em cima do Multi-valued consensus como se pode ver na figura 3.3 e a decisão dele é feita a partir de um vetor que contem os valores propostos pelos vários processos. É representado pelas seguintes propriedades:

- **Validação do vetor** - Qualquer processo não malicioso que vá decidir, decide sobre um vetor (pelo menos (f+1) elementos do vetor vieram de processos corretos);
- ***Agreement*** - Todos os processos não maliciosos decidem o mesmo valor;
- **Terminação** - Todos os processos não maliciosos acabam por decidir;

Explicação do protocolo: Resumidamente, o protocolo transmite com o RBC uma mensagem com o valor proposto pelo processo. Depois disso, o algoritmo começa cada ronda à espera da receção de (n-f+r) mensagens (sendo r o número da ronda).
Seguidamente, há a construção de um vetor com os valores recebidos, esse vetor é proposto para o ***Multi-valued Consensus***.
Por último, quando houverem processos suficientes a propor o mesmo vetor para o ***Multi-valued Consensus*** então esse vetor é decidido por esse protocolo e imediatamente a seguir pelo ***Vector Consensus***.

3.1.4 *Atomic Broadcast*

O último protocolo proposto por Correia et al. foi o ***Atomic Broadcast***. Este protocolo corre em cima do ***Vector Consensus*** e é o último protocolo da arquitetura como se pode ver na figura 3.2.
O ***Atomic Broadcast*** basicamente faz a entrega das mensagens tal como o RBC mas garante que sejam entregues na mesma ordem a todos os processos. É também usado tanto no HB quanto no Dumbo.
É representado pelas seguintes propriedades:

- **Validação** - Sempre que há a transmissão de uma mensagem por um processo não corrupto então essa mensagem vai ser entregue por algum processo não corrupto;
- ***Agreement*** - Sempre que um dos processos que não é corrupto entrega alguma mensagem, essa mensagem vai ser entregue por todos os processos;

- **Integridade** - "For any identifier ID, every correct process p delivers at most one message M with identifier ID, and if sender(M) is correct then M was previously broadcast by sender(M)" [14], resumidamente o id é único para cada mensagem, não há duas mensagens entregues com o mesmo id;
- ***Total Order*** - Garante que as mensagens são entregues sempre na mesma ordem.

Explicação do protocolo: Resumidamente, o protocolo chama o **atomic broadcast** que vai transmitir as mensagens para todos os processos com o RBC. Sempre que uma mensagem é entregue pelo RBC é posta num conjunto, quando não está vazio há uma tentativa entre os processos em concordar na entrega das mensagens que estão no conjunto.
De seguida, constrói-se um vetor que tem a *hash* das mensagens que estão no conjunto falado, esse vetor é proposto pelos processos para o ***vector consensus***. O ***vector consensus*** vai decidir algum vetor que tem pelo menos (2f+1) vetores de diferentes nós (a *hash* tem que aparecer em pelo menos (f+1) desses vetores para se ter a certeza que foi proposta por um processo não malicioso).
Depois de todas as mensagens que vão ser entregues estarem no conjunto falado anteriormente, estas são finalmente entregues por ordem e removidas desse conjunto.

Concluindo, depois da leitura e análise destes protocolos propostos pro Correia et al. foi possível ter uma visão mais clara sobre os protocolos de consenso resistentes a faltas bizantinas. Através destas descrições houve um maior entendimento sobre conceitos que foram realmente importantes para a realização desta dissertação e por consequente, sobre a perceção do Dumbo.

3.2 HoneyBadger

Como foi falado no Capítulo 1, à medida que *blockchains* evoluíam, o interesse nos protocolos de consenso surgiu de forma mais intensiva sendo que, numa fase inicial todas as instituições que desenvolviam protocolos de consenso focavam-se e baseavam-se no PBFT (um dos primeiros protocolos principais no tópico de resolver o consenso no que toca a faltas bizantinas) [5].
Os protocolos mais conhecidos neste tópico de consenso, tais como Paxos, Raft e PBFT dependem de suposições de sincronia [12] [15] [5] e falham quando alguns parâmetros não são atendidos. Esta questão traz algumas desvantagens em relação aos protocolos assíncronos que não necessitam de seguir nenhuns parâmetros temporais e conseguem deste modo fazer progressos mesmo que as condições da rede não sejam as melhores.
Visto haver a necessidade de construir um protocolo robusto e assíncrono não só teoricamente mas também prático para obedecer a certos parâmetros surge então o primeiro protocolo BFT *atomic broadcast* assíncrono prático que garante *liveness* sem a necessidade de fazer suposições temporais, o HoneyBadgerBFT [6].
O HoneyBadger usa como principal bloco de construção o ACS e também usa o TPKE, uma primitiva criptográfica que permite que cada parte encripte uma mensagem para uma chave

pública master de maneira a que os nós da rede trabalhem juntos para a desencriptar.
O artigo proposto por Andrew Miller et al. aborda 4 protocolos que vão ser explicados nesta secção, sendo eles o ACS, BA/ABA , RBC e por último o protocolo principal, HoneyBadgerBFT.

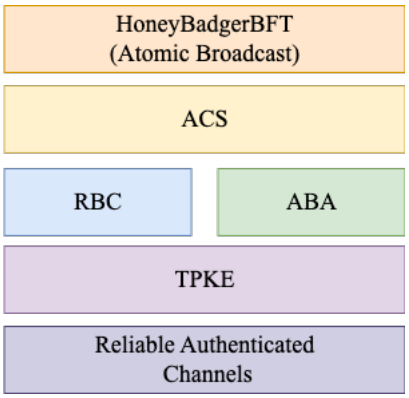


Figura 3.5: Arquitetura do protocolo HoneyBadgerBFT.

Como se pode ver na figura 3.5, necessária para representar como funcionam os protocolos, consegue-se perceber como estão organizados e que podem correr várias instâncias dos que estão a baixo na stack sendo que o HoneyBadgerBFT corre o ACS que por sua vez corre o RBC e o ABA que estão por cima de criptografia *threshold* e de *reliable channels*.

3.2.1 *Atomic Broadcast*

Primeiramente é explicado o problema do Atomic Broadcast e da rede envolvente que foram criados para o HoneyBadgerBFT.
Os nós vão recebendo transações e no final o propósito é chegar a um *agreement* consensual sobre a ordem das transações (que são strings únicas). São os clientes que geram e enviam as transações.
O sistema proposto por Andrew Miller et al. faz algumas suposições, sendo elas:

- **Rede puramente assíncrona** - É assumido também que são usados **canais confiáveis** e autenticados entre os nós. Todas as mensagens que são enviadas por nós corretos são eventualmente entregues e os nós têm *buffers* que não apresentam limite para conseguirem processar todas as mensagens;
- **Faltas bizantinas** - Pode haver um potencial controlo por alguém malicioso de até f nós maliciosos. O limite inferior neste cenário é 3f + 1 <= N;
- **Setup confiável** - Inicialmente, é assumido que os nós interagem com uma entidade confiável para estabelecer chaves públicas e *shares* secretas.

O protocolo ***Atomic Broadcast*** já foi abordado anteriormente por Correia et al. e volta a

ser utilizado neste artigo de Andrew Miller et al. na construção do HoneyBadger (que é um protocolo atomic broadcast) e apresenta as seguintes propriedades:

- ***Agreement*** - Se algum participante não malicioso fizer output de uma operação então todos os outros não maliciosos fazem exatamente a mesma coisa;
- ***Total Order*** - Garante que as mensagens são entregues sempre na mesma ordem por todos os participantes corretos;
- ***Censorship Resilience*** - Se houver uma operação que é *input* de N-f nós não maliciosos então todos os nós não maliciosos vão dar essa operação como *output*.

3.2.2 Construção do protocolo HoneyBadgerBFT

O protocolo **HoneyBadgerBFT** proposto por Andrew Miller et al. gira em torno de uma primitiva chamada **ACS** (***Asynchronous Common Subset***) onde cada participante propõe um valor e o *output* de cada é um vetor com as propostas de *input* de pelo menos N-2f participantes não maliciosos.
De forma a evitar que alguma entidade maliciosa saiba quais os participantes e quais as operações propostas por eles é utilizado o ***Threshold Encryption***.

3.2.2.1 *Threshold Encryption*

A **criptografia de chave pública/privada** tradicional baseia-se na posse da chave privada correspondente à chave pública usada para a encriptação. Os dados podem ser desencriptados por qualquer participante que possua a chave privada, permitindo que uma única entidade tenha acesso total aos dados.
Por outro lado, a ***Threshold Encryption*** baseia-se no esforço colaborativo de várias entidades que detêm cada uma, uma parte da *decryption key* . Os dados podem ser desencriptados apenas quando um *threshold* específico dos vários participantes combinam as suas partes da chave, o que impede que qualquer entidade individual acesse os dados sozinha.

A primitiva de ***Threshold Encryption*** previne que possíveis atacantes descubram sobre as transações propostas pelos participantes isto porque qualquer uma das partes consegue encriptar a mensagem com uma chave pública *master* mas no caso da desencriptação os participantes têm que o fazer juntos com as suas partes da chave, esta abordagem que é utilizada no HoneyBadger é baseada na proposta de J. Baek e Y. Zheng [2].

3.2.2.2 *Reliable Broadcast*

O RBC garante que todas as mensagens cheguem a todos os nós e que todos nós não maliciosos entreguem a mesma mensagem. Andrew Miller et al. propõe o RBC baseado no trabalho de Gabriel Bracha [4] e refere as propriedades dos canais assíncronos *Reliable Broadcast*:

- ***Agreement*** - Os participantes que entreguem, entregam os mesmos valores;
- **Totalidade** - Quando um participante não malicioso entrega um valor, então todos os outros participantes não maliciosos entregam esse valor;
- **Validação** - Quando um participante não malicioso receber *input* de um valor todos os outros que são corretos vão entregar esse valor.

3.2.2.3 *Binary Agreement*

O BA que pode também ser chamado de ABA como vamos ver a seguir, faz com que os participantes decidam sobre um único bit (0/1), Andrew Miller et al. propõe esta primitiva e depois implementa-a com base no trabalho de Mostefaoui et al.
Segue as seguintes propriedades:

- ***Agreement*** - Se os participantes não maliciosos fizerem *output* de um bit então todos os outros participantes não maliciosos dão *output* desse bit;
- **Terminação** - Sabendo que todos os participantes não maliciosos recebem *input* então todos participantes não maliciosos fazem *output*;
- **Validação** - Sabendo que um participante que é correto fez *output* de um bit então pode-se dizer que pelo menos um participante correto recebeu esse bit.

Esta primitiva permite que um bit seja decidido se todos os participantes não maliciosos receberem esse esse mesmo bit como *input*.

3.2.2.4 *Asynchronous Common Subset*

O ACS é o principal bloco de construção do HoneyBadgerBFT e apresenta as seguintes propriedades:

- **Validação** - Se um participante não malicioso dá como *output* um conjunto, o tamanho do conjunto é >= n-f sendo que esse conjunto contem os *inputs* de N-2f participantes não maliciosos;
- ***Agreement*** - Se um participante não malicioso fizer *output* de um valor, então todos os outros participantes fazem *output* desse valor;

- **Totalidade** - Se n-f participantes não maliciosos receberem alguma transação como *input* então quer dizer que todos os participantes da rede vão gerar um *output*.

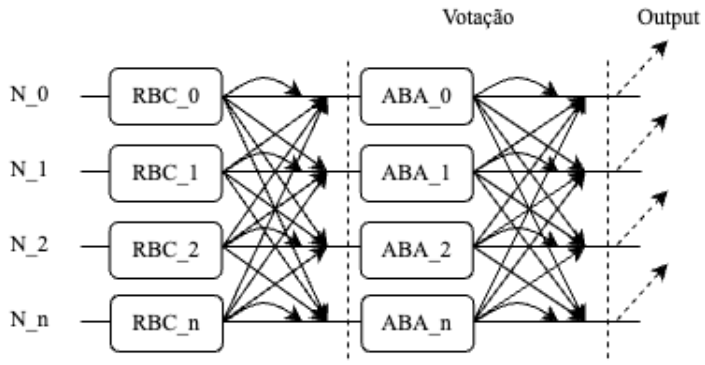


Figura 3.6: Asynchronous Common Subset do HoneyBadgerBFT.

Como principal bloco de construção, o **ACS** é baseado no trabalho proposto por Ben-Or et al. [7] e utiliza as primitivas referidas nas subsecções anteriores, o ACS e BA/ABA como se pode ver na figura 3.6.
Resumidamente, cada participante chama o RBC que irá transmitir o seu valor de *input* para todos os outros participantes e seguidamente participa nas instâncias do ABA concordando sobre qual subconjunto de *inputs* se vai incluir. Basicamente, os participantes que recebem alguma transação de outros participantes entram nas instâncias ABA com o valor 1, em seguida, quando n-f instâncias fizerem *output* de 1, são começadas novas instâncias com o valor 0 (isto para transações que ainda não recebeu), no final há o *output* das várias transações que tiveram valor 1 nas instâncias ABA.

3.2.2.5 HoneyBadgerBFT

Por último, é proposto então o protocolo principal, o HoneyBadgerBFT que como se pode ver na figura 3.5 aparece no topo da *stack* e utiliza os protocolos que se seguem da *stack*.
Como se pode verificar na figura 3.7, o HoneyBadgerBFT começa com o ACS que recebe como *input* um *ciphertext* que foi selecionado a partir de um conjunto de transações pelos nós que o cifraram.
De seguida, na parte azul da figura 3.7, no RBC os participantes recebem um conjunto que adveio do ACS e aqui criam uma *share* vinda da desencriptação do que receberam, por último enviam então para todos os participantes uma mensagem que contém a *share* e outras informações (índice da ronda e da *share* e identificador do participante).
Na última parte do protocolo, parte verde da figura 3.7, é quando um participante recebe f+1 mensagens enviadas pelo RBC e junta as *shares* (conjunto de *shares*) recebidas para decifrar então o *plaintext* que é basicamente o conjunto de transações inicialmente propostos que vão ser então usadas (escolhidas).

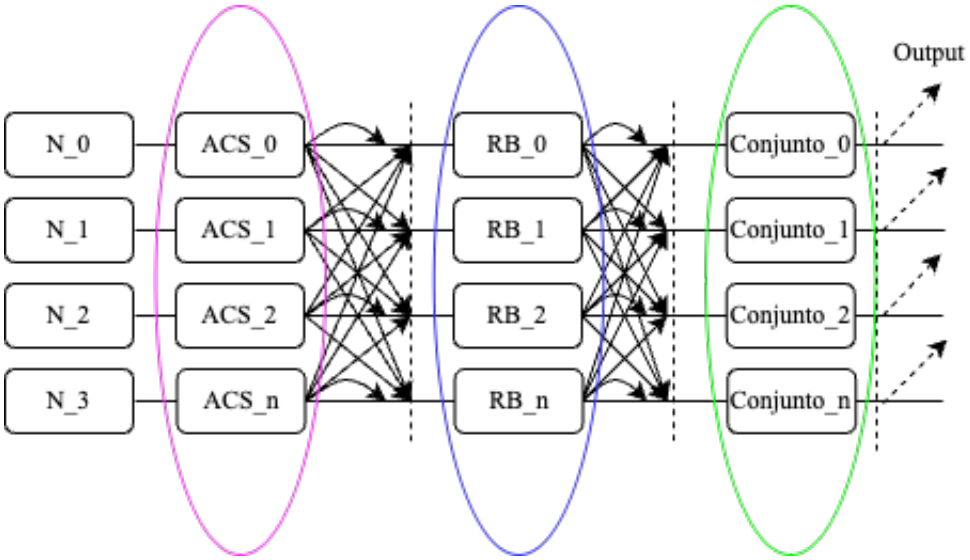


Figura 3.7: Protocolo HoneyBadgerBFT.

Relevante abordar que primeiramente os autores do HoneyBadger [6] referiram que o protocolo consistia numa redução do ACS para MVBA, mas em seguida notam que se torna uma dificuldade e que se obtém melhor eficiência com um ACS alternativo juntamente com o RBC, excluindo assim o MVBA. Na próxima secção pode-se verificar que a situação do MVBA é diferente do que foi proposto aqui e que se é possivel tirar benefício da utilização do protocolo.

3.3 Dumbo

Neste Capítulo é explicado o protocolo Dumbo, o protocolo testado nesta dissertação, que foi construido com base no HoneyBadgerBFT [6] do Miller et al. mas com uma nova proposta de ACS e com a utilização do MVBA que no artigo de Miller et al. não era viável.
A arquitetura do protocolo está demonstrada na figura 3.8, onde cada camada utiliza as camadas de baixo sendo que no topo encontramos o ***Atomic Broadcast*** que utiliza o ACS que por sua vez utiliza o MVBA com o RBC e o ABA que estão por cima de criptografia *threshold* e de *reliable channels*.

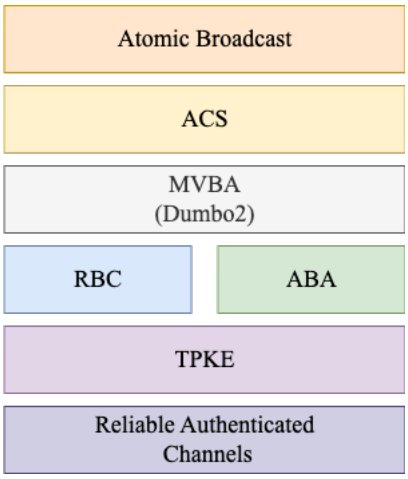


Figura 3.8: Arquitetura do protocolo Dumbo.

Depois do primeiro protocolo BFT *atomic broadcast* assíncrono prático que demonstrou bons resultados, o HoneyBadgerBFT, segue-se o artigo de Bingyong Guo et al. que propõe 2 novos protocolos **atomic broadcast**, o Dumbo1 e o Dumbo2, ambos protocolos BFT assíncronos. Duas coisas importantes deste trabalho foi reduzir as instâncias do ABA e também utilizar o MVBA (com zelo) que no trabalho anterior não era viável mas aqui consegue obter um ACS mais eficiente visto que é ele juntamente com o PRBC que constroem o ACS.
O **Dumbo1** consegue então reduzir o número das instâncias do *asynchronous binary agreement* (ABA) correndo apenas um pequeno conjunto e o **Dumbo2** vai mais longe reduzindo este número a constante. Bingyong Guo et al. realizou as mesmas experiências que foram feitas com o HoneyBadger para perceber que os resultados do Dumbo são ligeiramente melhores no que toca à latência e taxa de transferência (*throughput*).
Tal como o HoneyBadger, o Dumbo é um protocolo que apresenta uma rede de comunicação puramente assíncrona com canais confiáveis e autenticados entre os nós, assume ainda que pode haver até f nós faltosos num conjunto de n nós onde n é maior ou igual a 3f + 1, esses nós são certificados por uma infraestrutura de chave pública (PKI).

Os protocolos *atomic broadcast*, ACS e RBC são utilizados pelo **Dumbo** e têm as mesmas propriedades que já foram utilizadas no **HoneyBadger** com a mesma proposta que foi abordada nos capítulos em cima.

3.3.1 *Asynchronous Binary Agreement*

O protocolo ABA é um protocolo que permite que um grupo de n nós chegue a um acordo sobre um único valor (bit), que pode ser 0 ou 1.
Cada nó tem seu próprio valor de input e o objetivo é que todos os nós concordem sobre o mesmo valor. É também projetado para garantir que, se qualquer nó honesto decidir um valor de bit específico, então todos os nós honestos também decidem esse valor (***Agreement***); se todos os nós

honestos receberem input, então todos os nós decidem um valor, sendo ele 0 ou 1 (**Terminação**); e se qualquer nó honesto decidir um valor específico, então pelo menos um nó honesto deve ter recebido esse valor como input (**Validação**).
No protocolo ABA, é possível que alguns nós continuem à espera de mensagens mesmo depois de terem chegado a uma decisão, desde que todos os nós honestos tenham decidido um valor [8].

3.3.2 Esquema de Encriptação *Threshold*

Um esquema *(t, n)-non interactive threshold signature* (t de 0 até n) é um sistema que envolve n nós e permite que até t-1 deles possam ser corruptos. O esquema de assinatura é projetado para ser resistente a falsificações e garante que todas as partes honestas eventualmente obtenham uma assinatura correta para uma mensagem dada que pode ser verificada por outros [8].

3.3.3 *Multi-valued Byzantine Agreement*

O protocolo MVBA é um método para permitir que um grupo de nós chegue a um acordo sobre um valor que não é obrigatório ser 0 ou 1. Todos os nós estão cientes de um predicado computável global e polinomial em tempo chamado Q, que é específico para o protocolo utilizado e pode ser usado para determinar a ação adequada em alguma situação dada. Cada nó oferece um valor único de *input* que inclui informações de validação específicas, e o protocolo assegura que o valor final da decisão foi proposto por pelo menos um nó. O protocolo MVBA pode satisfazer quatro propriedades: **terminação**,que significa que, se todos os nós honestos tiverem input de valores válidos de fora, todos os nós terão output de um valor; **validade externa**, se um nó honesto tiver output de um valor, então o predicado Q(v) será verdadeiro; ***agreement***, todos os nós honestos que terminam têm o mesmo valor de output; e **integridade**, se todos os nós forem honestos e alguns nós derem output de um valor, então alguns nós devem ter proposto esse valor [8].

3.3.4 $(1,K,\epsilon) - CommitteeElection(CE)$

O protocolo CE tem pelo menos f + 1 participantes honestos num conjunto de n, o processo termina com um grupo de nós honestos a produzir um conjunto de k nós chamado *committe set* C, que inclui pelo menos um nó honesto.
Apresenta 4 propriedades: **Terminação** - sempre que for iniciado por (f + 1) nós não faltosos e todas as mensagens entre os nós honestos forem recebidas, os participantes honestos vão gerar o conjunto C; ***agreement*** - qualquer dois nós honestos geram o mesmo conjunto; **validação** - se qualquer nó honesto produzir o conjunto C, ele terá k elementos, terá probabilidade igual para cada nó de ser selecionado e incluirá pelo menos um nó honesto; ***unpredictability*** - antes de ser chamado por um participante honesto, a chance de um participante corrupto acertar o *committee* resultante é, no máximo 1/(n k) [8].

3.3.5 *Consistent Broadcast*

O protocolo CBC proposto por Reiter [16] é utilizado apenas no Dumbo2 para a entrega confiável das transações e é semelhante ao protocolo RBC mas sem a propriedade da totalidade que se pode verificar no Capítulo 3.2.2.2.

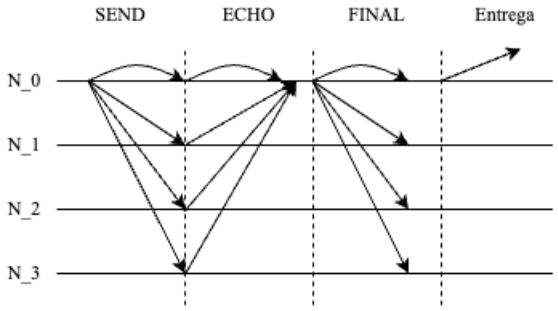


Figura 3.9: Protocolo *Consistent Broadcast* CBC.

Como se pode ver na figura 3.9 o remetente envia uma mensagem que tem informações importantes para todas as partes envolvidas no protocolo (**SEND**), quando um participante recebe a mensagem, ele cria uma *share* de uma assinatura *threshold* e envia de volta para o remetente (**ECHO**). Se um determinado número de *shares* da assinatura *threshold* for obtido, tendo ele que ser 2f+1, o remetente recebe então a assinatura na mensagem e envia para todos os participantes (**FINAL**). Por último, quando recebem a mensagem com a assinatura *threshold*, entregam a mensagem original (**Entrega**). Estas etapas são usadas no Dumbo2 para garantir a segurança da transmissão da mensagem.

3.3.6 Dumbo1-ACS

Como já foi referido no inicio deste Capítulo, o protocolo Dumbo1 consegue reduzir o número de instâncias ABA que são necessárias na execução do ACS. Em suma este protocolo apresenta 2 fases principais que se enquadram no protocolo RBC, sendo elas **data-RBC** e **index-RBC** (que se encontram, respetivamente, nos dois sítios da figura 3.10 que têm o RBC).
Na primeira fase (data-RBC), os inputs de cada participante são transmitidos e também se selecionam k líderes para a segunda fase (index-RBC). No index-RBC, espera-se por n-f transações e depois transmite-se os índices dos participantes que as receberam. Por último, há então a inserção do valor 1 para o ABA se tiver sido recebido as n-f transações e os seus respetivos valores que vêm da primeira fase [8].

A construção do Dumbo1 foca-se então em 5 passos principais:

- Cada participante envia uma mensagem para todos os outros participantes com o valor proposto;
- Depois de transmitidas as mensagens, participam todos no protocolo CE, selecionando o

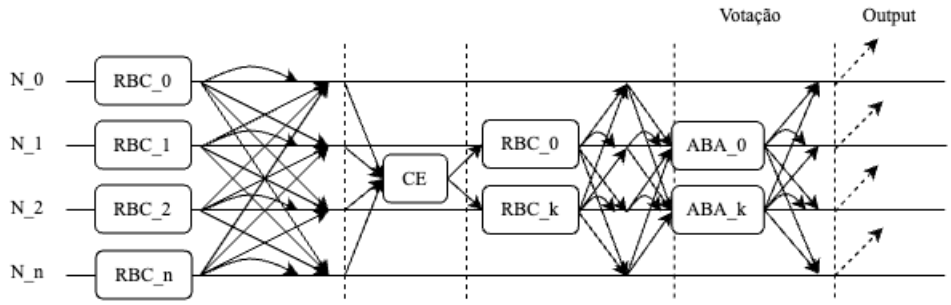


Figura 3.10: *Asynchronous Common Subset* do Dumbo1.

committee C e guardando os identificadores dos membros dele;

- Neste ponto é quando se chega à segunda fase, o index-RBC, os membros de C esperam por n-f mensagens que contem o valor proposto inicial de cada participante e transmitem depois disso uma mensagem que contem os índices de cada através do protocolo RBC;
- Chega-se então aqui à fase da votação que se pode ver na figura 3.10, onde cada participante não corrupto depois de receber a mensagem com os índices que foi enviada antes e todos os valores iniciais que correspondem a esses índices insere o valor 1 para o protocolo ABA. Se um participante honesto receber um *output* de alguma instância ABA que pertence a C e ao conjunto de identificadores, insere então 0 nas outras instâncias ABA que ainda não receberam *input*;
- Depois de todas as instâncias do ABA (k) serem executadas, cada participante vai esperar pelas mensagens com os índices correspondentes que tinham valor 1 e depois disso espera pelos valores iniciais propostos correspondentes a esses índices. O *output* dado é o conjunto de valores [8].

3.3.7 Dumbo2-ACS

O Dumbo2 já leva o ACS a outro patamar conseguindo reduzir o número de instâncias do ABA a constante com a ajuda do RBC e do MVBA.
Bingyong Guo et al. propõe o PRBC (*provable reliable broadcast*), um tipo de RBC, porque o MVBA pode ser alterado por um participante corrupto e produzir conjuntos desonestos, o PRBC vai emitir uma prova garantindo a chegada do valor correspondente a cada participante não corrupto.

Provable Reliable Broadcast

O protocolo PRBC é usado apenas no Dumbo2 e gera uma prova da autenticidade dos participantes, verificando-os com assinatura *threshold* utilizando os identificadores do RBC, isto faz-se com o algoritmo *Verify* e o *id* (identificador) [8].
Bingyong Guo et al. aborda o protocolo PRBC com as seguintes propriedades [8]:

- *Agreement* - Quando dois participantes honestos dão output de dois valores, sendo eles v e v' então v é igual a v';
- Totalidade - Se qualquer participante emitir (id,σ) e a verificação for bem sucedida, ou seja (Verify(id, σ) = 1),então todos os participantes não corruptos fazem output do valor e de (id,σ);
- Validação - Se o participante que envia os valores não for corrupto e fizer input de um valor então todos os outros participantes não corruptos fazem output desse valor e de uma string válida (id,σ);
- *Succinctness* - O tamanho de um σ que seja válido não depende do tamanho do valor que é emitido.

O processo do PRBC passa por na primeira fase enviar através do remetente um valor que foi recebido como input para o RBC, de seguida, depois dos participantes receberem a mensagem com esse valor enviam uma mensagem *Done* para todos os outros participantes que contem o identificador do RBC e a *share* da assinatura *threshold*. Por último, os participantes esperam por f+1 mensagens *Done* válidas, juntam as *shares* respetivas e fazem output [8].

Podemos encontrar na figura 3.11 todas as fases do Dumbo2 e como se pode ver começa com o PRBC que envia o conjunto resultante para o MVBA.

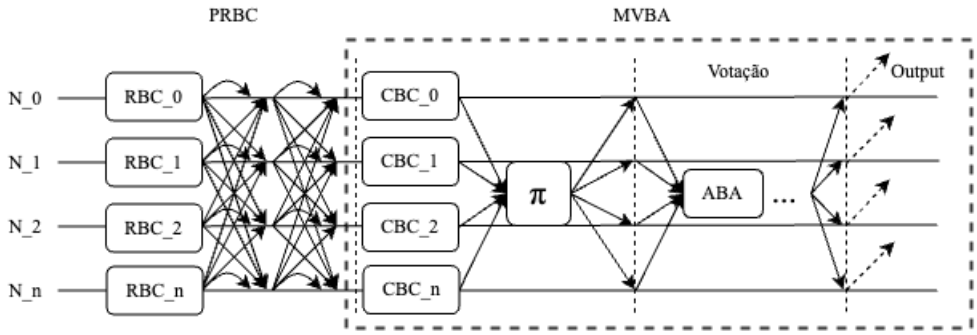


Figura 3.11: *Asynchronous Common Subset* do Dumbo2.

O Dumbo2 está então representado na figura 3.11 e como referido em cima, o valor de input é enviado pelos participantes para o protocolo PRBC, de seguida, os participantes ao receberem uma mensagem válida *Finish* do PRBC, adicionam o índice e *shares* respetivos ao conjunto. Depois de receberem n-f mensagens *Finish* diferentes do PRBC chamam o MVBA com os conjuntos criados. Depois de ser invocado, acontece todo o processo do MVBA, descrito em 3.3.3 e por último, quando um participante recebe então o resultado do MVBA que contem os índices ele espera pelas mensagens que têm os valores que correspondem aos índices e retorna [8].

3.3.8 *Atomic Broadcast*

O protocolo *atomic broadcast* utilizado por Bingyong Guo et al. é semelhante ao do Andrew Miller et al. [6] que já foi explicado em cima no Capítulo 3.2, encontra-se no topo da *stack* como se pode ver na figura 3.8 e utiliza todos os protocolos que foram abordados anteriormente neste Capítulo.

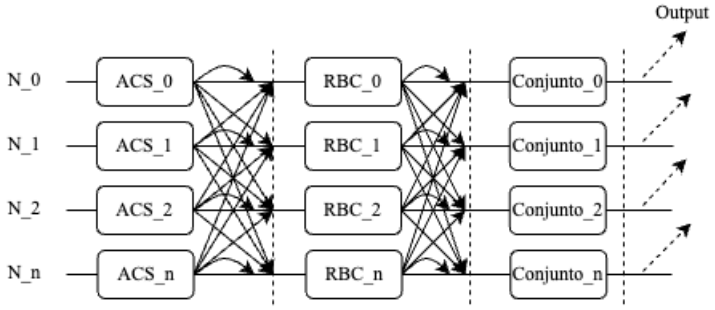


Figura 3.12: *Atomic Broadcast do Dumbo.*

Na figura 3.12 podemos ver a representação do *atomic broadcast* que é igual à que foi apresentada no HoneyBadger na figura 3.7 e que já foi bem explicada. Em resumo, temos a representação do Dumbo em si que é um protocolo de consenso distribuído que permite que um grupo de participantes chegue a acordo sobre um conjunto de transações. Começa pelo ACS (podendo ser o Dumbo1 ou Dumbo2 dependendo da implementação), em seguida o conjunto que adveio do ACS é então entregue pelo RBC que produziu *shares* da desencriptação dos valores recebidos. Por último, há a combinação das *shares* recebidas sendo possível decifrar o *plaintext*, que são as transações que vão ser executadas e que são enviadas como resultado do Dumbo.
No próximo Capítulo, o Dumbo é apresentado por fases, sendo o *atomic broadcast* dividido nos seus sub-protocolos e por aí adiante, conseguindo perceber tanto as funcionalidades em especifico dos sub-protocolos como quais poderão ser as vulnerabilidades e como podem ser exploradas.

Capítulo 4

Análise do Problema

Neste capítulo serão apresentados as diferentes possibilidades de vulnerabilidades pensadas dentro dos vários subprotocolos do Dumbo. Em cada etapa está descrita uma análise detalhada das faltas que poderão ser impactantes em cada um dos protocolos e quais os sítios específicos das alterações. Depois de identificadas quais e onde vão ser feitas as injeções de faltas, vai se conseguir perceber no Capítulo 5, qual o real impacto de cada uma delas.

O objetivo desta dissertação era, primeiramente analisar e avaliar as vulnerabilidades do protocolo HoneyBadger, depois de ter sido escolhida a implementação em python que se encontra no GitHub [17] verificou-se que já não era atualizada e alterada entre 4 e 6 anos e depois de muitas tentativas de instanciar o protocolo não se conseguiu obter sucesso (as versões dos sistemas utilizados não são compatíveis).
Quando não se obteve êxito nas várias tentativas decidiu-se optar então pelo Dumbo, um protocolo que é baseado no HoneyBadger mas que tentou superá-lo na performance e que a implementação é muito mais recente, tendo sido atualizada em 2022, como se pode ver no GitHub [13].

4.1 Desenho da Análise

Para elaborar a avaliação do protocolo Dumbo é necessária uma estratégia bem definida. Antes da análise detalhada das modificações e testes realizados no código, é essencial explicar o contexto e metodologia que orientaram as alterações. Nesta subsecção são apontados os pontos do protocolo que vão ser testados/analisados e quais os testes que vão ser efetuados.

4.1.1 Atomic Broadcast

O Dumbo, como se pode verificar na arquitetura representada na figura 3.8 tem como protocolo principal o *Atomic Broadcast* que está posicionado no topo da pilha. Isto significa que o *Atomic Broadcast* engloba todos os subprotocolos identificados no Dumbo.

Dado isto, começou-se por pensar que alterações são possíveis fazer a este protocolo de maneira a identificar possíveis vulnerabilidades. O *Atomic Broadcast* está dividido no ACS, RBC e por fim na entrega do conjunto final como se verifica na figura 4.1. Importante notar que o ACS é o protocolo central do Dumbo uma vez que permite o consenso em ambientes desafiantes onde atrasos em mensagens e nós corruptos podem perturbar os métodos tradicionais de consenso. Ao concentrar-se num subconjunto de nós que podem alcançar o consenso apesar destes desafios consoante o número limite de participantes faltosos, pode ser assegurada a credibilidade e integridade do processo de consenso.
Por estas razões, dentro do *Atomic Broadcast* foi decidido testar e modificar o ACS.

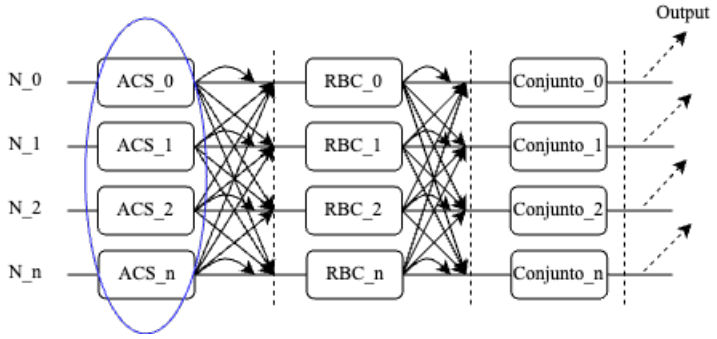


Figura 4.1: *Atomic Broadcast.*

4.1.2 ACS

O ACS representa vários outros protocolos como se pode verificar na figura 3.11, sendo eles divididos maioritariamente pelo PRBC e o MVBA. Dividindo então esses pontos estratégicos, o ACS vai ser testado sendo que vão ser injetadas faltas nas várias etapas do PRBC e também nas diferentes fases do MVBA, sendo elas o CBC, a instância *Coin*, o ABA e por último a entrega das transações.

Visto que alguns protocolos do Dumbo estão divididos em diferentes ficheiros de código dentro da pasta principal, na próxima secção os subcapítulos estão divididos pelo nome do protocolo principal identificado nesse ficheiro mas podem incluir vários outros protocolos.
Esclarecendo, a secção 4.2.1 é o *Atomic Broadcast* e representa as modificações feitas na fase inicial do ACS, sendo ela o RBC e PRBC, a secção 4.2.2 é o MVBA que também faz parte do ACS, apenas não está incluído na secção anterior porque é um ficheiro à parte e tem muitas fases estratégicas que têm que ser especificamente representadas. O mesmo acontece com o acontece com a secção 4.2.3 que é o CBC, este protocolo também se encontra dentro do MVBA mas está especificado num ficheiro de código à parte dos outros, por essa razão cada umas das suas etapas sofreram modificações que são essenciais na performance do Dumbo.

4.1.3 Modificações

Depois de identificados os pontos principais onde vão ser realizadas as modificações também foram pensados quais os testes que deverão ser efetuados. Dentro do contexto e também dos requisitos teóricos do protocolo as faltas pensadas foram:

- Alterar as mensagens não corruptas por mensagens corruptas nos nós faltosos
- Trocar as mensagens não corruptas por nenhuma mensagem nos nós faltosos
- Os nós faltosos não esperem/recebam o output decidido
- Alterar a ordem da permutação da eleição sequencial dos lideres para um n fixo menor nos nós faltosos
- Alterar o valor da votação dos nós faltosos para um valor diferente do possível (0 ou 1)
- Trocar as etiquetas dos protocolos que são enviadas na transação para uma *string* diferente apenas nos nós corruptos
- Alterar a *share* que os nós faltosos vão enviar
- Alterar a chave privada do ECDSA (contém as chaves públicas de todos) nos nós corruptos

Seguidamente, na secção 4.2 vai ser feita a análise do código e as modificações efetuadas nos sítios estratégicos definidos nesta secção de cada protocolo.

4.2 Análise do Código

Todos os protocolos abordados no Capítulo 3.3 encontram-se na implementação escolhida [13], exceto o Dumbo1-ACS visto que nesta implementação é utilizado o Dumbo2-ACS, por essa razão neste capítulo aborda-se apenas esse protocolo.
Dentro da pasta do Dumbo e depois de analisar a fundo todos os ficheiros de código percebeu-se que os diferentes protocolos estariam divididos, o *atomic broadcast* encontra-se no ficheiro **dumbo.py**, o Dumbo2-ACS no ficheiro **dumbocommonsubest.py**, o PRBC no **provablereliablebroadcast.py**, o MVBA no **validatedagreement.py** e por último, o CBC no **consistentbroadcast.py**.
Nos próximos Subcapítulos está feita a análise detalhada de cada protocolo, foi crucial para a perceção das possíveis vulnerabilidades caso um ou mais nós fosse corrupto. São identificadas então as possíveis vulnerabilidades, os sítios no código de cada protocolo que foram necessários alterar para as obter e como seria de esperar que eles se comportassem.

4.2.1 Atomic Broadcast

No ficheiro dumbo.py, encontra-se o protocolo ***Atomic Broadcast***, como podemos ver na figura 3.12 e como foi explicado no Capítulo 3.3.8 o ACS envia o conjunto resultante do protocolo para ser transmitido de forma confiável pelo RBC e por último combina-se as *shares* para se obter o conjunto de transações final. Posto isto, este é o protocolo principal e engloba todas as fases necessárias, sendo possível verificar as vulnerabilidades prováveis através da alteração do código em sítios específicos e estratégicos.

4.2.1.1 ACS

É percetível que o Dumbo, tal como o HoneyBadger, gira em torno do ACS, (sendo neste caso o Dumbo2-ACS). Por essa razão, no ficheiro dumbo.py foi alterado o código que faz parte do ACS dentro do *atomic broadcast* para se conseguir perceber se haveria algum impacto. De notar que houveram também alterações aos subprotocolos dentro do ACS que serão explicadas nas próximas Secções.

Em primeiro lugar, foi considerado o envio das transações selecionadas de forma aleatória recebidas como input no sitio especificado na figura 4.2, como são entregues com o RBC e o Dumbo tem n >= 3f+1, sabe-se que, com por exemplo 4 nós, consegue-se tolerar apenas 1 nó faltoso. Por essa razão foram pensadas 2 possíveis alterações:

- Com n nós, o protocolo tolera f nós faltosos, sendo n >= 3f+1. Pode-se então alterar o número de nós que é suposto o protocolo tolerar, por exemplo, com n = 4, altera-se um nó qualquer para em vez de enviar o valor suposto para os outros nós, não enviar nada;
- A segunda possibilidade pode ser também alterar o nó escolhido, para em vez de enviar o valor suposto aos outros nós, enviar um valor qualquer aleatório (quanto maior o n mais nós faltosos o protocolo tolera (n >= 3f+1) e portanto os testes podem ser adequados ao n e ao f conforme for necessário);

Com estas alterações é possível perceber se o protocolo se comporta como esperado. Está-se à espera que a performance do Dumbo se mantenha e não haja nenhuma queda na taxa de transferência, na latência e que todos entreguem as mesmas transações no final. Os resultados a estes testes encontram-se no Capítulo 5 - Testes e Validação.

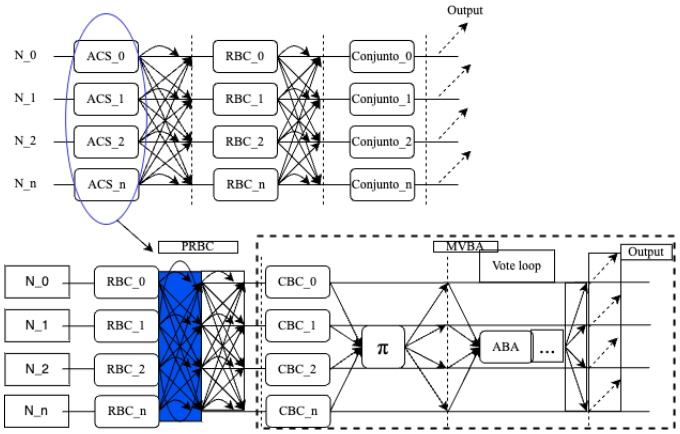


Figura 4.2: Modificações no protocolo PRBC

De seguida, foi considerada a segunda parte no processo do PRBC onde há a entrega/espera de n-f mensagens *finish* que contem os índices e as *shares* do conjunto recebido, o sitio encontra-se representado na figura 4.3 a azul. Como já foi abordado, o Dumbo tem n >= 3f+1 e, como exemplo, sabe-se que com 4 nós consegue tolerar apenas 1 nó faltoso. Sendo que espera por n-f mensagens válidas, 4-1 = 3 (neste exemplo), a espera é de pelo menos 3 mensagens válidas. Tal como antes, as alterações possíveis são então, f nós não enviarem nada ou enviarem valores aleatórios em vez do valor suposto e espera-se que mais uma vez, de acordo com as especificações do protocolo, não haja alterações na performance nem no *output*/transações finais entregues.

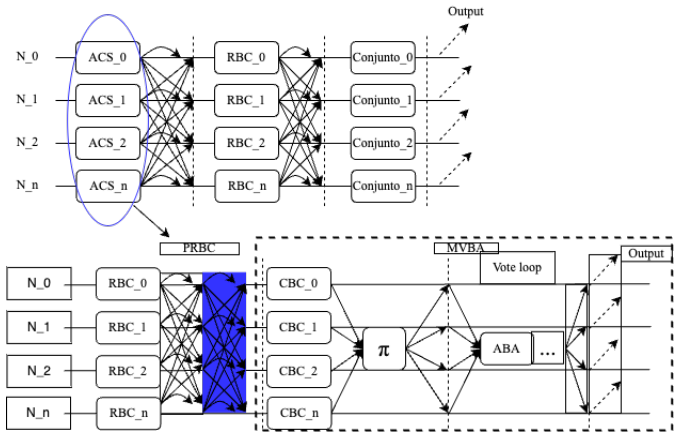


Figura 4.3: Modificações no protocolo PRBC(2)

Por último, ainda no ficheiro dumbo.py, encontra-se função que espera pelo output do PRBC, no sitio em azul da figura 4.4. Os participantes esperam por f + 1 mensagens válidas, juntam as shares e fazem então *output* para o MVBA (invocando-o). Com n = 4 como no exemplo anterior e com f = 1, o protocolo espera então por 2 mensagens válidas para invocar o MVBA, neste caso, pode-se fazer com que 2 nós (no caso deste exemplo) não esperem/recebam o *output* decidido. Mais uma vez, de acordo com as especificações do protocolo, espera-se que as alterações (supostas vulnerabilidades) não causem impacto, no Capítulo 5 verifica-se que nem sempre é assim.

De notar que o ficheiro dumbocommonsubset.py apenas chama e define as funções criadas nos outros ficheiros (com o output) que têm os subprotocolos do ACS para começar a correr o protocolo e por isso, este ficheiro não é alterado.

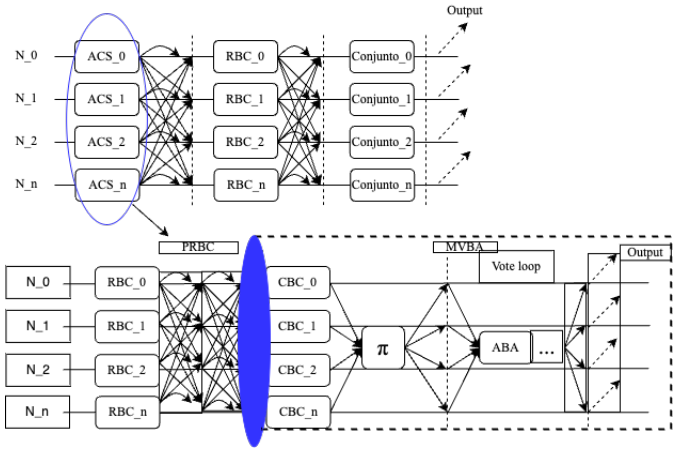


Figura 4.4: Modificações no protocolo PRBC(3)

4.2.2 MVBA

O MVBA que se encontra no Dumbo2-ACS permite que os participantes cheguem então a um acordo sobre os valores que recebe do PRBC (não sendo necessário serem 0 ou 1) e está no ficheiro validateagreement.py. Resumidamente, é aqui que finalmente se escreve o valor final no canal escolhido.
Como já foi referido e se pode verificar na figura 3.11, este protocolo tem os subprotocolos CBC, ABA e também a operação de *multicast* com a *common coin*. Apesar destes subprotocolos estarem definidos noutros ficheiros como por exemplo o consistenbroadcast.py que apresenta todas as fases do CBC, como se pode ver na subsecção seguinte, o validatedagreement.py envia os valores que vêm do CBC e do ABA e por essa razão consegue-se também alterá-los para, mais uma vez, verificar as possíveis vulnerabilidades.

Posto isto, começa-se pelo primeiro passo do MVBA como se verifica na figura 4.5, o CBC (alterado de forma mais detalhada na próxima secção). A primeira função é o cbc_send(k, o), os parâmetros o e k são o valor a ser enviado e o nó para o qual se vai enviar, respetivamente. Nesta situação o *output* será o nó e o valor, pensou-se então em alterar o parâmetro 'o' (um tuplo com os valores que vieram de trás) para None,None em f nós faltosos (f representando aqui os nós faltosos que o protocolo tolera), sendo que no exemplo de n = 4 e f = 1, apenas um participante envia valores corruptos em vez de enviar o que seria suposto. Com isto, é possível verificar se existe uma possível vulnerabilidade, mais uma vez na teoria não deveria haver alterações nos resultados do protocolo e tudo se deveria manter igual ao normal.
Outra opção foi fazer com que f nós faltosos de acordo com a tolerância dos n >= 3f+1, enviassem valores aleatórios ou strings corruptas e verificar se teria algum impacto no protocolo. Cumprindo

todas as especificações teóricas do protocolo, não poderia haver impacto prático nos resultados e performance, esses valores e efeitos práticos estão descritos no próximo capitulo. De realçar ainda que, teoricamente se não se cumprir a tolerância dos nós corruptos que o protocolo consegue suportar, ou seja, se houver mais do que aquilo que ele tolera o protocolo não termina.

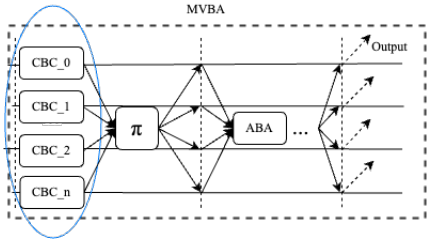


Figura 4.5: Modificações no protocolo MVBA - *Consistent Broadcast*

No passo seguinte, como especificado na figura 4.6, quando um nó recebe pelo menos 2f+1 propostas rápidas entra num *loop*, em cada ronda do *loop*, um candidato é escolhido na ordem dada por uma permutação pi de 1;..;n. É aqui que uma instância *Coin* permuta os IDs dos nós para eleger sequencialmente os líderes. De maneira a verificar vulnerabilidades, altera-se a ordem da permutação para um n fixo menor em f nós (nós faltosos) avaliando assim se há algum impacto.

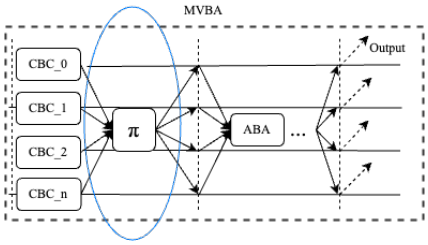


Figura 4.6: Modificações no protocolo MVBA(2)

Avançando para a primeira parte do protocolo ABA da figura 4.7, executa-se repetidamente instâncias ABA até que o output 1 seja gerado. Para cada candidato, se o nó aceitou o ***consistent broadcast*** contendo a proposta do candidato, envia uma mensagem "*yes-vote*", caso contrário envia "*no-vote*". Somente após a obtenção de 2f+1 votos "*yes*", o próximo passo, será iniciado. Como os votos podem ser apenas 0 ou 1, alterando-se apenas a votação de f nós para -1 pretende-se que, cumprindo as especificações do protocolo não haja impacto.

Quando se obtém 2f+1 votos positivos, patilha-se o valor escolhido (função def aba_coin_bcast(o)) com todos os nós envolvidos e testa-se novamente as vulnerabilidades como já foi feito anteriormente, alterando-se o valor real por um valor faltoso em f nós.

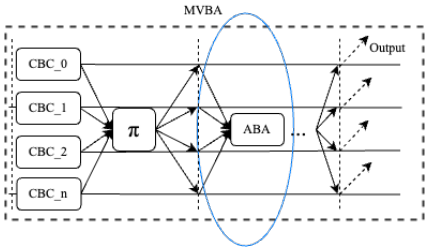


Figura 4.7: Modificações no protocolo MVBA(3) - *Asynchronous Binary Agreement*

Finalmente, olhando para a figura 4.8, há uma cópia profunda da função *send* encapsulada do aba, onde temos o valor escolhido a ser enviado e também o nó que vai receber o valor, conseguindo-se obter por último o output do protocolo MVBA. Nesta fase do protocolo MVBA altera-se uma vez mais o valor escolhido para um valor faltoso em apenas f nós, o que não deverá impactar o output.

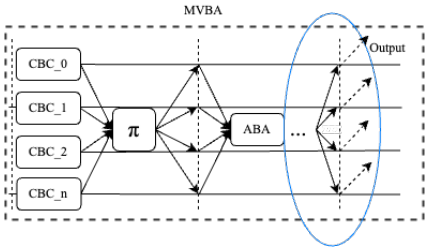


Figura 4.8: Modificações no protocolo MVBA(4)

4.2.3 CBC

O protocolo CBC encontra-se inserido no MVBA do Dumbo2-ACS e permite a entrega confiável das transações. É semelhante ao protocolo RBC mas sem a propriedade da totalidade (conceito abordado no Capítulo 3.3.5) e garante a segurança da transmissão da mensagem.
O ficheiro de código consistentbroadcast.py representa especificamente e com todas as suas fases o CBC e tem 4 etapas principais como se pode ver na figura 3.9.
Neste caso, pensou-se em dividir as etapas do CBC e perceber se em cada uma delas conseguimos encontrar vulnerabilidades cumprindo as especificações teóricas do protocolo. A primeira etapa, como se pode ver na figura 4.9 é o SEND, depois de receber o output do PRBC que é dado como *input* ao CBC, o líder (neste caso todos são lideres, ou seja, todos enviam o seu input para os outros participantes) envia o *input* para cada um dos nós com o CBC-SEND. Por essa razão, as possíveis vulnerabilidades são mais uma vez fazer com que f nós (sendo f o número de nós faltosos que o protocolo tolera) não enviem nada ou enviem valores aleatórios em vez do que era suposto pelo protocolo.
Nestes dois casos, não deveria mais uma vez ser causado impacto no protocolo visto que segue todas os requisitos que eles impõe.

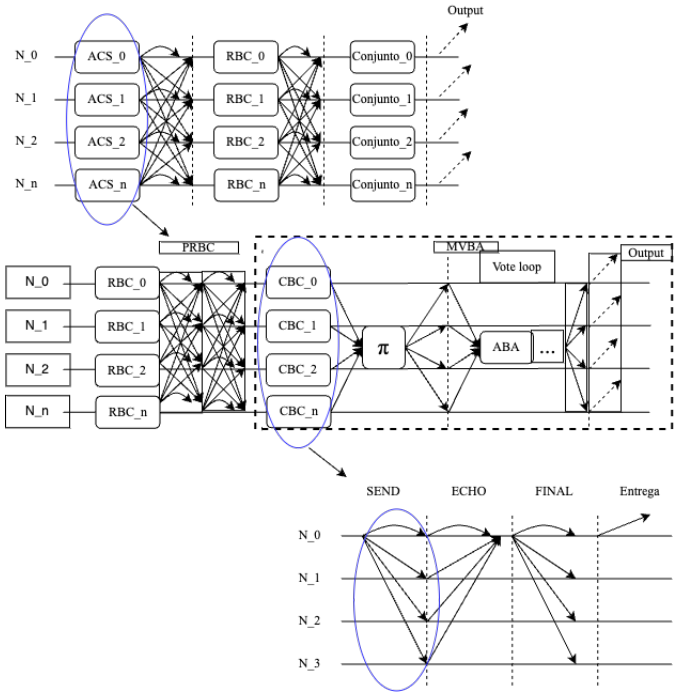


Figura 4.9: Modificações no protocolo CBC(1)

A seguir, como se pode ver na figura 4.10 é o ECHO, quando um nó recebe a mensagem payload gera uma *share* de uma *threshold signature* vinculando o *payload* à instância de *broadcast* específica, e envia de volta ao remetente, em resumo nesta fase são enviadas as *shares* de assinatura. Neste caso pode-se testar 3 ações diferentes, alterar a *share* que f nós vão enviar, a chave privada do ECDSA (contém as chaves públicas de todos) e por último a "etiqueta"'CBC_ECHO'.

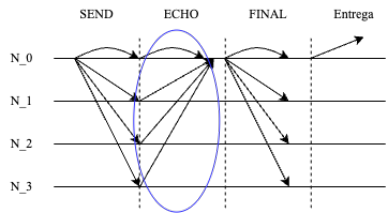


Figura 4.10: Modificações no protocolo CBC(2)

A terceira fase é a FINAL (última antes da entrega da mensagem), da figura 4.11. Nesta fase, o protocolo espera por n-f mensagens CBC-ECHO (com as *shares* da assinatura) para poder finalmente enviar o CBC-FINAL com a assinatura *threshold* para todos os nós.
Depois de verificadas as possíveis vulnerabilidades, visto que tem que esperar por pelo menos n-f mensagens, se tivermos o exemplo de n = 4 e f = 1, sabe-se que o protocolo espera por 3 mensagens para enviar o CBC-FINAL. Por essa razão, com este exemplo, testa-se o caso de 1 nó não enviar nada ou enviar valores aleatórios e espera-se que não haja qualquer impacto no protocolo.

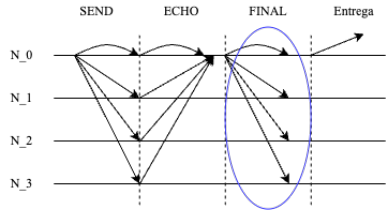


Figura 4.11: Modificações no protocolo CBC(3)

Por último é feita a entrega, na figura 4.12, onde a mensagem original é entregue depois de ser recebida a assinatura do CBC-FINAL, retornando um valor diferente da mensagem real em apenas f nós permite verificar se vai haver algum impacto não suposto. No Capitulo 5, vai-se perceber que este protocolo é fulcral para a performance do protocolo Dumbo.

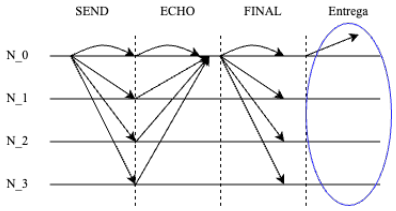


Figura 4.12: Modificações no protocolo CBC(4)

Os testes obtidos das possíveis vulnerabilidades apontadas neste capitulo vão ser analisados no Capítulo 5 - Testes e Validação.

Capítulo 5

Testes e Validação

Neste capítulo serão apresentados e discutidos os testes efetuados com a injeção de faltas nas diferentes etapas do Dumbo(protocolo principal) e dos subprotocolos. O objetivo destes testes é, de acordo com as especificações teóricas do protocolo, cumprindo sempre com os requisitos verificar se poderá haver algum impacto tanto na performance quando nas transações que cada nó entrega. A injeção de faltas foi feita em sítios estratégicos de maneira a ser fácil de visualizar onde o impacto é maior.

Decidiu-se testar o protocolo com as faltas utilizando o Docker e uma única máquina. Em duas pastas diferentes tem-se o código modificado e o código sem qualquer alteração. Dito isto, foi utilizado uma máquina MacOS Monterey com 16GB de RAM, mas uma vez que a execução do código estava a tornar-se um problema criou-se a imagem base Docker utilizando o Ubuntu 18.04 como base, criada foi com "ROM –platform=linux/amd64 ubuntu:18.04 AS base_config", onde –platform=linux/amd64 é um modificador que define a plataforma alvo da imagem Docker construída com 64 bits num sistema Linux.
Com apenas uma imagem Docker consegue-se fazer a execução adequada do protocolo, isto porque foi criado o docker-compose onde o endereço ipv4, porta, pasta (com e sem alterações), entre outros parâmetros foram especificados em cada um dos nós (dependendo do id), conseguindo assim identificar quais os nós são ou não faltosos.

5.1 *Throughput e Latência*

Antes da análise dos testes efetuados, começa-se por perceber as métricas de *performance* para avaliação do protocolo, denominadas *throughput* e latência.

Throughput refere-se à taxa com que as transações/dados são processados ou transferidos dentro de um sistema. Mede a quantidade de trabalho que pode ser realizado num certo período de tempo. Com um alto *throughput* o sistema lida eficientemente com um grande volume de dados. Neste caso é medido em transações por segundo(tps), definindo a quantidade de transações média que o protocolo envia por segundo.

Latência é o tempo que demora uma transação a ir do ponto de origem até ao destino e uma resposta ser recebida. Representa o tempo de espera do sistema. Quando há baixa latência as transações foram processadas ou entregues rapidamente, com um atraso mínimo. Normalmente é medida em ms, mas para representação deste sistema mediu-se em segundos por transação(spt).

As duas métricas são importantes para avaliação do desempenho geral e a capacidade de resposta do sistema. Com um *throughput* alto e baixa latência consegue-se alcançar um sistema equilibrado.

5.2 Injeção de faltas

5.2.1 *Cluster* com 4 nós

Inicialmente testou-se o protocolo Dumbo utilizando 4 nós. Como já foi visto anteriormente, pode haver até f nós faltosos num conjunto de n nós onde n é maior ou igual a 3f + 1, por essa razão com n = 4 o protocolo consegue suportar no máximo 1 nó faltoso.

Utilizando o **docker-compose** como referido anteriormente, definiu-se o volume para a replica dumbo0 que representa o Nó 0, definido como o faltoso com /Users/utilizador/Desktop/Tese/-dumbo_master:/app1 sendo que a pasta dumbo_master é utilizada com injeção de faltas e para os Nós 1,2 e 3 das réplicas dumbo1, dumbo2 e dumbo3 com /Users/utilizador/Desktop/Tese/-dumbo_master1:/app1 onde a pasta dumbo_master1 representa o protocolo sem faltas. Em cada uma das réplicas e para cada nó utilizou-se no docker-compose o comando ["python3", -u", "/app1/run_socket_node.py", --sid", "sidA", --id", "0", --N", "4", --f", "1", --B", "1000", --K", "20", --S", "50", --T", "1", --F", "1000000", --P", "dumbo"] definindo os 4 nós (–N) com 1 nó faltoso (–f), 20 rondas executadas (–K), o tamanho do *batch* 1000 (–B) sendo que isto significa que cada nó está a entregar 15000 transações.

5.2.1.1 Sem injeção de faltas

Focando nos testes obtidos, como representado na tabela 5.1, o *throughput* médio do protocolo sem injeção de faltas é 228,53 transações por segundos enquanto que a latência é 0,0044 segundos por transação ou 4,4 milisegundos. Sendo que o cada nó entrega 15000 transações e que todos os nós dão o mesmo *output* que representa as transações entregues por cada um (verifica-se que entregam todos as mesmas transações).

Throughput (tps)	Latência (spt)
228,53	0,0044

Tabela 5.1: *Throughput* e latência sem injeção de faltas

5.2.1.2 Provable Reliable Broadcast

Na tabela 5.2 e evidenciando uma vez mais que se está a cumprir todas as especificações do protocolo (f=1), apenas alterando-se o nó 0 que não está a envia nada (tuplo None,None) ou a enviar valores faltosos (*string aleatória*). O *throughput* médio do protocolo sofre um ligeiro decréscimo passando de 228,53 para 192,64 transações enviadas por segundo e a latência aumenta para 0,0052 segundos por cada transação ou 5,2 milisegundos. Pode-se considerar que 0,8 ms de aumento de latência por transação algo significativo se forem testados sistemas de alta escala com milhões de transações, neste caso cada nó entrega 15000 transações e por essa razão considera-se tolerável esta diferença. De notar que seria esperado que o *throughput* quase não diminuísse e a latência aumentasse pouco.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	192,64	0,0052

Tabela 5.2: *Throughput* e latência com injeção de faltas - PRBC

Na segunda parte do protocolo PRBC como se verificou na figura 4.3 a azul, há a entrega/espera das n-f mensagens *finish* que representam os índices e as shares do conjunto recebido. Novamente e tal como na primeira parte do PRBC altera-se o conteúdo que vai ser entregue ao CBC, para None,None ou uma string aleatória. Na tabela 5.3 o *throughput* e latência médios do protocolo mantêm-se semelhantes aos valores anteriores da tabela 5.2, o que seria de esperar visto que são dois passos intermédios com os mesmos valores e do mesmo protocolo, o PRBC.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	189,52	0,0053

Tabela 5.3: *Throughput* e latência com injeção de faltas - PRBC(2)

A seguir ao protocolo PRBC, há a espera pelo *output* do mesmo. Resumidamente, os participantes esperam por f + 1 mensagens válidas, juntam as shares e invocam o MVBA. Neste caso, com f = 1, e cumprindo as especificações teóricas do protocolo poderia haver 2 nós a não receber/esperar pelo output do PRBC que seria esperado não haver impacto na performance do protocolo e que tudo ocorresse normalmente. Verificou-se que com apenas apenas o nó 0 a não receber nenhum valor do PRBC não se consegue obter **não se consegue obter o *throughput* e a latência**, isto porque há um erro no *output* e nenhuma transação é entregue no final, sendo impossível obter o tempo real de execução de cada nó.

5.2.1.3 Multi-valued Validated Agreement - CBC

Começando pela primeira fase do MVBA, o protocolo CBC divide-se em 4 passos intermédios e todos eles foram representados e testados devidamente. Começando pela primeira etapa, definida no Capítulo 4.1.3 como SEND, onde são enviados por cada um dos participantes os valores recebidos do PRBC, foi alterado o que o nó 0 envia, sendo que passou a não enviar nada ou valores aleatórios (as duas possibilidades dão o mesmo resultado). Na tabela 5.4 é demonstrado que o *throughput* médio do protocolo sofreu um decréscimo grande passando de 228,53 para 130,48 transações enviadas por segundo e a latência aumentou para 0,0077 segundos por cada transação ou 7,7 milisegundos, isto porque o tempo de execução do protocolo duplicou. Estando a cumprir todos os requisitos teóricos do protocolo, nada previa que isto fosse acontecer, verificando-se então uma falha no protocolo principal, o **Dumbo**, quando se injeta faltas no CBC.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	130,48	0,0077

Tabela 5.4: *Throughput* e latência com injeção de faltas - CBC(send)

De seguida, na segunda etapa do CBC definida como ECHO, onde são enviadas as *shares* de assinatura, foram feitas diferentes alterações faltosas, sendo elas:

- Alterar a *share* que nó 0 envia para NONE - sendo que neste cenário há um *print "Signature share fail"* refletido no *output*. Concentrando nos testes obtidos como se verifica na tabela 5.5 não há um decréscimo significativo, tal como nas duas primeiras etapas do PRBC. O *throughput* médio do protocolo sofre um ligeiro decréscimo passando de 228,53 para 190,86 transações enviadas por segundo e a latência aumenta para 0,0052 segundos por cada transação ou 5,2 milisegundos;
- Alterar a chave privada do ECDSA (contem todas as chaves públicas) apenas no nó 0 - neste caso, não há entrega de transações nem se consegue obter o tempo de execução do protocolo, o nó 0 não termina;
- Alterar a *flag* CBC_ECHO para NONE ou para uma *string* aleatória acontece exatamente o mesmo que o primeiro ponto com o mesmo *throughput* e latência da tabela 5.5.

Verifica-se mais uma vez, que o CBC é um protocolo que quando afetado em certas etapas, apresenta não só um impacto na performance como também pode fazer com que o **Dumbo** nem sequer termine.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	190,86	0,0052

Tabela 5.5: *Throughput* e latência com injeção de faltas - CBC(echo)

Nas seguintes etapas do CBC (FINAL e entrega), como se verifica nas tabelas 5.6 e 5.7, o *throughput* médio do protocolo diminuiu de forma evidente e a latência aumentou. Isto acontece porque houve a modificação da mensagem final que o nó 0 envia para os outros nós e consequentemente o nó 0 não retorna o valor suposto que irá ser recebido e definido no ABA. Ou seja, nos testes a mensagem foi alterada para outra mensagem aleatória e também para nenhuma mensagem (um valor vazio). Concluindo, há um impacto grande na performance do protocolo Dumbo quando se injeta faltas no protocolo CBC.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	127,90	0,0078

Tabela 5.6: *Throughput* e latência com injeção de faltas - CBC(Final)

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	140,15	0,0071

Tabela 5.7: *Throughput* e latência com injeção de faltas - CBC(Entrega)

5.2.1.4 Multi-valued Validated Agreement - Coin

Seguindo para a segunda fase do protocolo MVBA como se verificou anteriormente na figura 4.6, entra-se num *loop* onde a instância *Coin* permuta os IDs dos participantes elegendo de forma sequencial os líderes. Salientando os testes obtidos como representados na tabela 5.8 onde se alterou a ordem da permutação pi de 1;..;n para 1;..10; apenas no nó 0 (o nó faltoso) verifica-se que o *throughput* médio do protocolo sofreu um decréscimo passando de 228,53 para 190,54 transações enviadas por segundo e a latência aumentou para 0,0052 segundos por cada transação ou 5,2 milisegundos. Resultados muito parecidos a todas as etapas que foram testadas anteriormente nos vários protocolos, excluindo o CBC que representa um impacto maior.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	190,54	0,0052

Tabela 5.8: *Throughput* e latência com injeção de faltas - Coin

5.2.1.5 Multi-valued Validated Agreement - ABA

Por último, na fase final do protocolo MVBA encontra-se o ABA onde cada candidato, se aceitar a proposta do *CBC* envia uma mensagem *yes-vote*, caso contrário envia *no-vote*. Somente depois de receber 2f+1 votos "yes"o próximo passo vai ser iniciado. Visto que os votos são 0 ou 1, alterou-se o voto do nó 0 para -1. Focando nos testes obtidos, como representado na tabela 5.9,

o *throughput* médio do protocolo diminui de 228,53 para 197,75 transações enviadas por segundo e a latência aumenta para 0,0051 segundos por cada transação ou 5,1 milisegundos.

Finalmente, depois de obter 2f+1 votos positivos, o valor escolhido é partilhado como todos os nós e depois há uma cópia da função *send* do ABA sendo por fim dado o *output* do protocolo MVBA. Alterou-se o valor a ser partilhado e depois dado pelo MVBA apenas no nó 0 (tanto para *none* como um valor aleatório diferente do escolhido) que resultou com o *throughput* médio a diminuir de 228,53 para 196,28 e a latência a aumentar para 0,0051 por transação como representado na tabela 5.10.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	197,75	0,0051

Tabela 5.9: *Throughput* e latência com injeção de faltas - ABA(1)

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	228,53	0,0044
Com injeção de faltas	196,28	0,0051

Tabela 5.10: *Throughput* e latência com injeção de faltas - ABA(2)

Analisando o impacto da injeção de faltas como representado na figura 5.1, as modificações efetuadas não são tão significativas no PRBC e ABA como quando há um nó faltoso nas etapas send, final e entrega do CBC, neste último caso o tempo de execução quase duplica (diminuindo o *throughput* para metade e aumentando a latência por consequência) e a performance do protocolo Dumbo diminui o que não deveria acontecer quando se cumprem todas as especificações teóricas do protocolo. De notar ainda que na última parte do PRBC mais precisamente quando os participantes esperam por f + 1 mensagens válidas como se pode verificar na figura 4.4 há um impacto negativo no Dumbo. Quando se modifica esta fase do protocolo PRBC não se consegue obter o *throughput* e a latência porque não há entrega de transações no final.

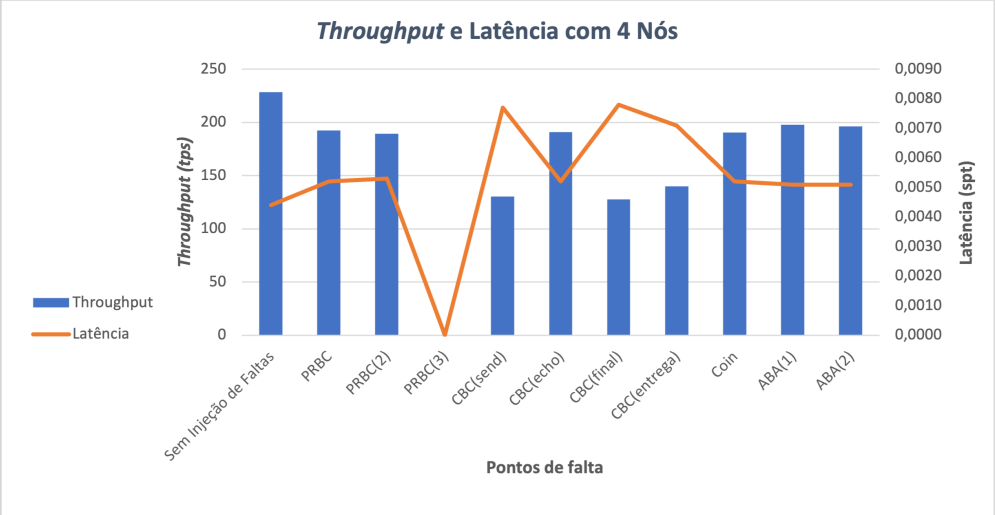


Figura 5.1: Resultados das modificações nos diferentes subprotocolos do Dumbo

5.2.2 *Cluster* com 8 nós

Os testes ao protocolo seguiram-se utilizando 8 nós. Mais uma vez, pode haver até f nós faltosos num conjunto de n nós onde n é maior ou igual a 3f + 1, por essa razão com n = 8 o protocolo consegue suportar no máximo 2 nós faltosos. Alterou-se os campos no docker-compose para os valores dos testes com n = 8, continuando cada nó a entregar 15000 transações.

5.2.2.1 Sem injeção de faltas

Para o caso dos 8 nós, como se pode ver na tabela 5.11, o *throughput* médio do protocolo sem injeção de faltas é 53,33 transações por segundos enquanto que a latência é 0,019 segundos por transação ou 19 milisegundos. Todos os nós dão o mesmo output que representam as transações entregues por cada um.

Throughput (tps)	Latência (spt)
55,33	0,019

Tabela 5.11: *Throughput* e latência sem injeção de faltas

5.2.2.2 Provable Reliable Broadcast

Diferente do caso a cima mencionado com n = 4, na primeira parte do PRBC utilizando exatamente os mesmos valores faltosos que no exemplo anterior mas agora adicionando 2 nós faltosos, sendo eles o nó 0 e o nó 1, o consenso não é realizado e nenhuma transação é entregue. Há um erro no *output* de (*not enough values to unpack (expected 3, got 2)*) e mesmo com apenas

um nó faltoso este erro é imprimido e o protocolo não termina. Verifica-se então, com n = 8, ao injetar faltas nesta primeira fase do PRBC o protocolo Dumbo não funciona adequadamente, o que não seria de esperar.

Já na segunda parte do PRBC, utilizando a mesma lógica do n = 4 com os mesmos valores faltosos, apenas adicionando mais um nó corrupto (nó 0 e nó 1), verifica-se na tabela 5.12 que o *throughput* médio do protocolo diminui de 55,33 para 48,37 transações enviadas por segundo e a latência aumenta de 0,019 para 0,021 segundos por transação ou 21 ms. Verifica-se o que foi concluído em cima, 2 ms de diferença poderá ser bastante significativo num sistema de grande escala e ter um impacto grande, neste caso a diferença é ainda tolerável.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	55,33	0,019
Com injeção de faltas	48,37	0,021

Tabela 5.12: *Throughput* e latência com injeção de faltas - PRBC(2)

Na espera pelo *output* do PRBC, o exemplo dado no Capítulo 5.2.1 é semelhante aos resultados com n = 8 e f = 2 descritos neste capítulo. Por essa razão, com o nó 0 e o nó 1 a não receberem nenhum valor do PRBC não se consegue obter o *throughput* e a latência. Nenhuma transação é entregue no final, sendo impossível obter o tempo real de execução de cada nó.

5.2.2.3 Multi-valued Validated Agreement - CBC

Começando pela primeira etapa do CBC, o SEND, onde são enviados por cada recebidos do PRBC , foi alterado o que o nó 0 e o nó 1 enviam, sendo que passam a não enviar nada ou valores aleatórios. Na tabela 5.13 verifica-se que o *throughput* médio e a latência neste passo não são possíveis de ser calculados porque o consenso não existe e não há entrega de transações. Não era de prever que não houvesse consenso visto que com n = 4 apesar de ter havido um impacto na performance do protocolo, todas as transações foram entregues.

Na segunda etapa, olhando agora para o ECHO do CBC representado na tabela 5.13 os testes efetuados são iguais aos pontos representados no Capítulo 5.2.1 onde alterando a *share* que os nós 0 e 1 enviam para um valor nulo ou a *flag* CBC_ECHO para NONE o decréscimo no *throughput* médio não é demasiado significativo (de 55,33 para 48,82 transações por segundo) e a latência aumenta em 1ms. Já no caso de se alterar a chave privada, mais uma vez há um grande impacto no Dumbo, e as transações não são entregues.

Na fase denominada por FINAL do CBC, com os mesmos valores faltosos do Capítulo 5.2.1, utilizando 2 nós faltosos (nó 0 e nó 1) o protocolo não entrega nenhuma transação e por essa razão não se consegue calcular o *throughput* e a latência.

A última fase de entrega, a mensagem que o nó 0 e o nó 1 enviam foi alterada para outra

mensagem aleatória e também para nenhuma mensagem, o impacto na performance do protocolo Dumbo verifica-se com o decréscimo do *throughput* de 53,33 para 48,94 transações por segundo e do aumento da latência para 0,020 segundos.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	55,33	0,019
Com injeção de faltas (**SEND**)	-	-
Com injeção de faltas (**ECHO**)	48,82	0,020
Com injeção de faltas (**FINAL**)	-	-
Com injeção de faltas (**Entrega**)	48,94	0,020

Tabela 5.13: *Throughput* e latência com injeção de faltas - CBC

5.2.2.4 Multi-valued Validated Agreement - Coin

Na segunda fase do protocolo MVBA como se verificou anteriormente, a instância *Coin* permuta os IDs dos participantes elegendo de forma sequencial os líderes. Altera-se como no Capítulo 5.2.1 a ordem da permutação pi de 1;..;n para 1;..10; apenas nos nós 0 e 1 (os nós faltosos) verifica-se, através dos testes obtidos como representado na tabela 5.14, que o *throughput* médio do protocolo sofre um decréscimo passando de 53,33 para 48,72 transações enviadas por segundo e a latência aumenta para 0,021 segundos por cada transação ou 21 milisegundos.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	55,33	0,019
Com injeção de faltas	48,72	0,021

Tabela 5.14: *Throughput* e latência com injeção de faltas - COIN

5.2.2.5 Multi-valued Validated Agreement - ABA

Na fase final do protocolo MVBA, chega-se ao ABA onde cada candidato vota *yes* ou *no* de acordo com o que foi referido em cima. Neste caso, alterou-se os votos dos nós 0 e 1 para -1 e verificou-se na tabela 5.15 que o *throughput* médio do protocolo diminui de 53,33 para 47,59 transações enviadas por segundo e a latência aumenta para 0,021 segundos por transação.

Finalmente, depois de obter 2f+1 votos positivos, o valor é enviado para todos os participantes sendo por fim dado o output do protocolo MVBA. Alterou-se o valor nos nós 0 e 1 (tanto para *none* como um valor aleatório diferente do escolhido) que resultou (tabela 5.16) com o *throughput* médio a diminuir ligeiramente 53,33 para 50,13 e a latência a aumentar para 0,020 por transação.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	55,33	0,019
Com injeção de faltas	47,59	0,021

Tabela 5.15: *Throughput* e latência com injeção de faltas - ABA(1)

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	55,33	0,019
Com injeção de faltas	50,13	0,020

Tabela 5.16: *Throughput* e latência com injeção de faltas - ABA(2)

Avaliando os testes da injeção de faltas como representado na figura 5.2, as modificações efetuadas na primeira e última parte do PRBC e também nas etapas send e final do CBC impactam significativamente o Dumbo. Ao modificar estas fases dos protocolos PRBC e CBC não se consegue obter o *throughput* e a latência porque não há entrega de transações.
Passando então de n=4 e f=1 para n=8 e f=2, os pontos de falha e o impacto aumentaram. Por exemplo, com n=4 o CBC apresentava uma diminuição significativa do *throughput* mas com n=8 se forem injetadas faltas nas duas etapas do CBC (send e final) a entrega das transações é totalmente comprometida (nenhuma transação é entregue).

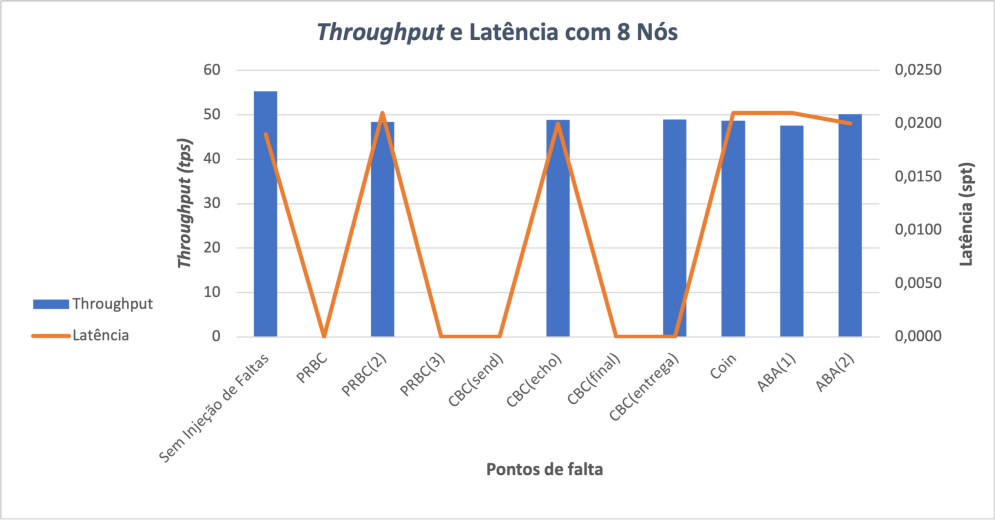


Figura 5.2: Resultados das modificações nos diferentes subprotocolos do Dumbo

5.2.3 *Cluster* com 16 nós

Os testes ao protocolo neste subcapítulo tiveram como base 16 nós. Com n = 16 o protocolo consegue suportar no máximo 5 nós faltosos. Alterou-se os campos no docker-compose para os valores dos testes com n = 16 e f = 5, continuando cada nó a entregar 15000 transações.

5.2.3.1 Sem injeção de faltas

Depois das modificações como representado na tabela 5.17, o *throughput* médio do protocolo sem injeção de faltas é 9,95 transações por segundos enquanto que a latência é 0,101 segundos por transação ou 101 milisegundos. Todos os nós dão o mesmo output ou seja, todos entregam as mesmas transações, o que já seria de esperar.

Throughput (tps)	Latência (spt)
9,95	0,101

Tabela 5.17: *Throughput* e latência sem injeção de faltas

5.2.3.2 Provable Reliable Broadcast

Tal como no caso a cima mencionado com n = 8, na primeira parte do PRBC utilizando exatamente os mesmos valores faltosos mas agora adicionando 5 nós faltosos sendo que n = 16, o consenso não é realizado e nenhuma transação é entregue. Dá exatamente o mesmo um erro no *output* de (*not enough values to unpack (expected 3, got 2)*), mesmo com os requisitos teóricos a primeira fase do PRBC o protocolo Dumbo não funciona adequadamente.

Já na segunda parte do PRBC, utilizando mais uma vez a lógica do n = 4 e n = 8 com os mesmos valores faltosos, agora com 5 nós corruptos, verifica-se como mostra a tabela **??** que o *throughput* médio do protocolo diminui de 9,95 para 9,70 transações enviadas por segundo e a latência aumenta de 0,101 para 0,103 segundos por transação ou 103 ms. Verifica-se que há uma diferença de 2 ms, o que poderá não ser significativo.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	9,95	0,101
Com injeção de faltas	9,70	0,103

Tabela 5.18: *Throughput* e latência com injeção de faltas - PRBC(2)

Seguidamente, na espera pelo *output* do PRBC, os resultados do teste com n = 16 são iguais aos dois testes realizados em cima. Por essa razão, com os nós 0,1,2,3 e 4 a não receberem nenhum valor do PRBC não se consegue obter o *throughput* e a latência sendo que nenhuma transação é entregue.

5.2.3.3 Multi-valued Validated Agreement - CBC

Na primeira etapa do CBC, o SEND, foram alterados 5 nós faltosos para enviarem valores aleatórios e não enviarem nada. O *throughput* médio e a latência neste passo não podem ser calculados porque o consenso não existe e não há entrega de transações. Com n = 4 apesar de

ter havido um impacto na performance do protocolo, todas as transações foram entregues ao contrário do que acontece com n = 8 e n = 16 o que demonstra que à medida que se aumenta o n e os participantes corruptos o Dumbo não consegue garantir os seus requisitos teóricos.

Na fase ECHO como demonstra a tabela 5.19, alterando-se mais uma vez a *share* que os 5 nós faltosos enviam para um valor nulo ou a *flag* CBC_ECHO para NONE o decréscimo no *throughput* médio não é significativo (de 9,95 para 9,28 transações por segundo) e a latência aumenta em 7 ms. Já quando se alterar a chave privada, verifica-se o impacto causado no Dumbo sendo que nenhuma transação é entregue.

Na etapa FINAL do CBC, com 5 nós faltosos não há entrega de transações.

Na fase de entrega, testou-se para os 5 nós faltosos enviarem uma mensagem corrupta e também nenhuma mensagem, verificou-se o decréscimo do *throughput* de 9,95 para 9,64 transações por segundo e do aumento da latência de 0,101 para 0,104 segundos.

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	9,95	0,101
Com injeção de faltas (**SEND**)	-	-
Com injeção de faltas (**ECHO**)	9,28	0,108
Com injeção de faltas (**FINAL**)	-	-
Com injeção de faltas (**Entrega**)	9,64	0,104

Tabela 5.19: *Throughput* e latência com injeção de faltas - CBC

5.2.3.4 Multi-valued Validated Agreement - Coin

Na segunda etapa do MVBA, na instância *Coin* alterou-se mais uma vez a ordem da permutação tal como no Capítulo 5.2.2 sendo que agora os testes são feitos com 5 nós faltosos. Verifica-se que neste caso de 16 nós sendo que 5 são nós corruptos, ao contrário do que acontecia com n = 4 e n = 8, não há entrega de transações nesta fase do protocolo e por essa razão não é possível calcular o *throughput* médio e a latência, havendo assim um impacto negativo no Dumbo.

5.2.3.5 Multi-valued Validated Agreement - ABA

Na fase final do MVBA, chega-se ao ABA onde há a votação de *yes* ou *no* conforme o que foi dito anteriormente. Alterou-se os votos dos 5 nós faltosos para -1 e verificou-se, como representado na tabela 5.20 que o *throughput* médio do protocolo diminui de 9,95 para 9,53 transações enviadas por segundo e a latência aumenta para 0,105 segundos, sendo que não são valores significativos.

Por último, depois de 2f+1 votos positivos serem recebidos, o valor é enviado para todos os participantes chegando-se assim ao *output* do MVBA. Testou-se esta fase do protocolo com 5 nós faltosos a enviarem tanto o valor *none* como um valor faltoso aleatório que resultou no *throughput*

médio a diminuir ligeiramente 9,95 para 9,49 e a latência a aumentar para 0,105 segundos como se verifica na tabela 5.21).

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	9,95	0,101
Com injeção de faltas	9,53	0,105

Tabela 5.20: *Throughput* e latência com injeção de faltas - ABA(1)

	Throughput (tps)	Latência (spt)
Sem injeção de faltas	9,95	0,101
Com injeção de faltas	9,49	0,105

Tabela 5.21: *Throughput* e latência com injeção de faltas - ABA(2)

Os testes da injeção de faltas com 16 nós em que 5 podem ser nós faltosos estão representados na figura 5.3. Como se consegue observar, mais uma vez as modificações efetuadas na primeira e última parte do PRBC e também nas etapas send e final do CBC impactam significativamente o Dumbo, não se conseguindo obter o *throughput* e a latência visto que nenhuma transação é entregue.
Para além destes dois subprotocolos junta-se agora a fase *Coin* que quando afetada compromete também a entrega das transações.
Um ponto importante nesta analise é que, passando de n=8 e f=2 para n=16 e f=5 os pontos de falha e o impacto aumentaram, comprovando-se como referido anteriormente que quando o n aumenta aumentam também os pontos de falha no Dumbo.

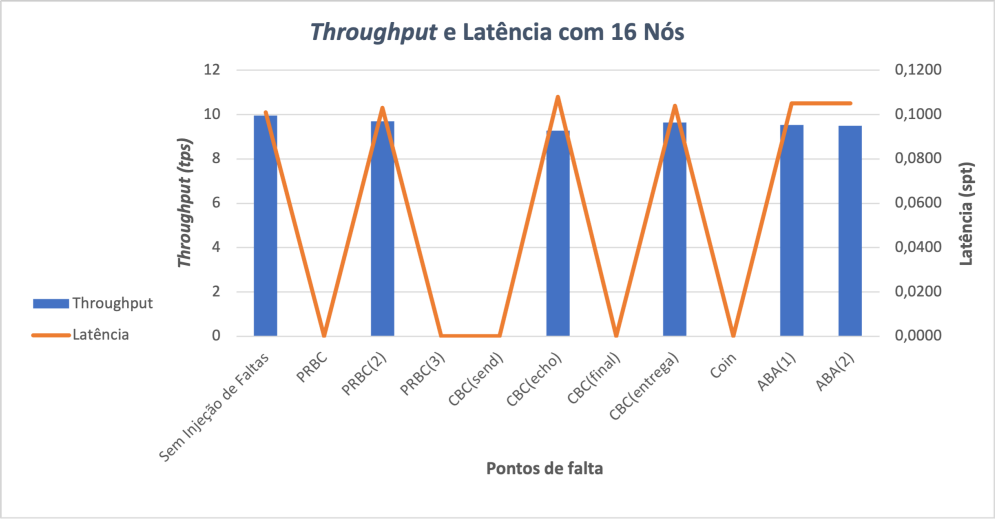


Figura 5.3: Resultados das modificações nos diferentes subprotocolos do Dumbo

5.2.4 Análise

Depois de efetuados os testes ao protocolo Dumbo com diferentes número de nós não corruptos e corruptos verifica-se que à medida que o N vai aumentando o protocolo vai sofrendo cada vez mais impacto na performance, sendo que os requisitos teóricos do protocolo foram sempre cumpridos.

É notório que com n = 4 e f = 1 o protocolo com mais impacto na performance do Dumbo quando um dos nós envia valores faltosos ou tem comportamento faltoso é o CBC como representado na tabela 5.22. Dentro das 4 etapas (SEND, ECHO, FINAL e Entrega), 3 enviam aproximadamente menos 100 transações por segundo do que quando não havia nenhum comportamento corrupto.

Cluster	Modificações	*Throughput (tps)*	Latência (spt)
4 nós	Sem injeção de faltas	228,53	0,0044
4 nós	PRBC	192,64	0,0052
4 nós	PRBC(2)	189,52	0,0053
4 nós	**PRBC(3)**	**-**	**-**
4 nós	**CBC(send)**	**130,48**	**0,0077**
4 nós	CBC(echo)	190,86	0,0052
4 nós	**CBC(Final)**	**127,90**	**0,0078**
4 nós	**CBC(Entrega)**	**140,15**	**0,0071**
4 nós	*Coin*	190,54	0,0052
4 nós	ABA(1)	197,75	0,0051
4 nós	ABA(2)	196,28	0,0051

Tabela 5.22: Modificações com 4 nós

No caso de n = 8 e f = 2, observa-se na tabela 5.23 que para além do CBC, também a primeira fase do PRBC tem um grande impacto no Dumbo. Neste caso, é um impacto ainda maior visto que não há entrega de transações e o consenso não é realizado. Acontece exatamente a mesma coisa para as fases SEND e FINAL do CBC que com 2 nós faltosos não conseguem entregar nenhuma transação. Todo este comportamento do protocolo não seria de esperar, analisando os testes obtidos depois das modificações, quando o n aumenta os comportamentos faltosos dos nós afetam ainda mais o protocolo principal.

Cluster	Modificações	*Throughput (tps)*	Latência (spt)
8 nós	Sem injeção de faltas	55,33	0,019
8 nós	**PRBC**	**-**	**-**
8 nós	PRBC(2)	48,37	0,021
8 nós	**PRBC(3)**	**-**	**-**
8 nós	**CBC(send)**	**-**	**-**
8 nós	CBC(echo)	48,82	0,020
8 nós	**CBC(Final)**	**-**	**-**
8 nós	CBC(Entrega)	48,94	0,020
8 nós	*Coin*	48,72	0,021
8 nós	ABA(1)	47,59	0,021
8 nós	ABA(2)	50,13	0,020

Tabela 5.23: Modificações com 8 nós

Com n = 16 e f = 5, para além do PRBC e CBC impactarem de forma negativa o Dumbo há o acréscimo da instância onde é feita a permutação dos IDs dos participantes para a eleição sequencial dos líderes ter também um papel fundamental na performance do protocolo como se verifica na tabela 5.24. Alterando-se a ordem da permutação pi que antes seria 1;..;n para 1;..10; apenas nos 5 nós faltosos o Dumbo não consegue suportar e não entrega nenhuma transação. Como nos testes anteriores com n = 4 e n = 8 este comportamento não existia quando havia modificações nesta etapa do protocolo, consegue-se comprovar mais uma vez que quando o n cresce o impacto das faltas é maior.

Cluster	Modificações	*Throughput (tps)*	Latência (spt)
16 nós	Sem injeção de faltas	9,95	0,101
16 nós	**PRBC**	**-**	**-**
16 nós	PRBC(2)	9,70	0,103
16 nós	**PRBC(3)**	**-**	**-**
16 nós	**CBC(send)**	**-**	**-**
16 nós	CBC(echo)	9,28	0,108
16 nós	**CBC(Final)**	**-**	**-**
16 nós	CBC(Entrega)	9,64	0,104
16 nós	***Coin***	**-**	**-**
16 nós	ABA(1)	9,53	0,105
16 nós	ABA(2)	9,49	0,105

Tabela 5.24: Modificações com 16 nós

Concluindo, depois dos testes efetuados consegue-se avaliar as vulnerabilidades do protocolo principal. De uma forma geral, o protocolo com mais impacto no **Dumbo** é o CBC desde o inicio dos testes com n=4 e f =1. Também a terceira fase do PRBC, onde há espera por f + 1

mensagens válidas, quando é comprometida afeta o Dumbo de uma maneira muito acentuada desde os primeiros testes, visto que não são entregues transações. Não seria de esperar que estes subprotocolos afetassem tanto a performance.

De seguida, verifica-se que quando os testes são efetuados com um maior número de dados, consequência do aumento dos parâmetros n e f, há subprotocolos do protocolo principal que começam a não enviar transações mesmo com todas as especificações teóricas cumpridas, como por exemplo a primeira fase do PRBC e também o *Coin* que não são afetados quando n = 4 e n = 8 respetivamente.

Capítulo 6

Conclusões

Esta dissertação é focada nos protocolos probabilísticos resistentes a falhas bizantinas que foram estudados/analisados nos primeiros capítulos. Decidiu-se testar e aprofundar o **Dumbo** [8] que propôs um novo protocolo ACS (Asynchronous Common Subset) e supostamente utiliza o protocolo MVBA que iria trazer mais eficiência ao ACS.

Analisou-se exaustivamente o Dumbo e a sua implementação de maneira a identificar possíveis vulnerabilidades caso um ou mais nós faltosos não respeitassem o protocolo. No Capítulo 5 testou-se o resultado das vulnerabilidades identificadas tendo em conta as especificações teóricas do protocolo.

Verificou-se no subprotocolo CBC, assumindo que no máximo f nós são faltosos num conjunto de n >= 3f+1 nós, há na mesma um impacto grande na performance que não era suposto acontecer. Como o CBC faz parte do MVBA que foi apontado como o subprotocolo que iria trazer mais eficiência ao ACS, ao contrário do HoneyBadger que não utiliza este protocolo, podemos assumir que esta afirmação não é tão real como na teoria.

Para além disso, sabe-se agora que à medida que o n aumenta também o protocolo perde eficiência quando existem f nós faltosos que respeitam os requisitos teóricos do protocolo.

Algumas especificações feitas pelos autores do **Dumbo** [8] na prática não são exequíveis e por essa razão, as vulnerabilidades mostram que o protocolo consegue suportar menos nós corruptos do que o suposto, a performance do Dumbo é afetada de forma ativa quando há injeção de faltas em certas etapas do protocolo principal.

Concluindo, dependendo do ponto onde a falta ocorre, o impacto da mesma tem consequências diferentes. Por exemplo, com 4 nós a alteração da mensagem original afeta mais quando se modifica o CBC(Final) (*throughput* = 127,90 tps) do que quando se modifica a primeira etapa do PRBC (*throughput* = 192,64 tps). Para além disso, à medida que o número de nós aumenta, aumentam também os pontos de falha do protocolo, com n = 4 há um impacto quando se modifica o CBC e a última etapa do PRBC, com n = 8 há impacto negativo através dos protocolos PRBC (primeira e última etapa) e CBC e por último com n = 16 o maior impacto acontece quando há

injeção de faltas nos protocolos PRBC, CBC e *Coin*. Todos estes pontos de quebra de protocolo abordados nesta dissertação podem ser um problema da implementação utilizada.

6.1 Trabalho Futuro

Como trabalho futuro, seria útil testar também o Honey Badger [6] fazendo uma comparação direta entre este protocolo e o **Dumbo** [8]. Isto porque, os dois são comparados na teoria mas não existe nenhuma comparação prática de qual será o mais eficiente quando são injetadas faltas em pontos estratégicos, podendo-se focar apenas no ACS e com isto, concluir-se entre os dois qual o protocolo probabilístico mais resistente a faltas bizantinas.
Por último, seria também útil conseguir perceber se os resultados obtidos e a queda da performance em alguns pontos de falta são devido à implementação utilizada ou se é um problema do desenho do protocolo.

Bibliografia

[1] Ailijiang, Charapko, and Demirbas. Consensus in the Cloud: Paxos Systems Demystified. Technical report, University at Buffalo, SUNY, August 2016.

[2] Baek and Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. 2003.

[3] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. December 2013.

[4] G. Bracha. Asynchronous byzantine agreement protocols. 1987.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. 1999.

[6] Andrew Miller et al. The honey badger of bft protocols. February 2016.

[7] Ben-Or et al. Asynchronous secure computations with optimal resilience. 1994.

[8] Bingyong Guo et al. Dumbo: Faster Asynchronous BFT Protocols. Technical report, Institute of Software, Chinese Academy of Sciences, Department of Computer Science, New Jersey Institute of Technology, State Key Laboratory of Cryptology, School of Computer Science and Technology, University of Chinese Academy of Sciences and JDD-NJIT-ISCAS Joint Blockchain Lab, October 2020.

[9] Müller S. et al. Fast probabilistic consensus with weighted votes. 2021.

[10] Y. Xiao et al. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22:1432–1465, January 2020. doi:10.1109/COMST.2020.2969706.

[11] Yang Xiao et al. Distributed consensus protocols and algorithms. *Blockchain for Distributed Systems Security*, chapter 2.

[12] Leslie Lamport. Paxos made simple. November 2001.

[13] Maxilbert. Implementação do dumbo em python do github. url: https://github.com/yylluu/dumbo, 2022.

[14] Miguel Correia Nuno Ferreira Neves and Paulo Veríssimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. Technical report, Faculdade de Ciências da Universidade de Lisboa.

[15] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. June 2014.

[16] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in rampart. pages 68–80, 1994.

[17] Brian Schroeder Sylvain Bellemare, Andrew Miller and dantengsky. Implementação do honeybadger em python do github. url: https://github.com/initc3/HoneyBadgerBFT-Python/, 2016.

[18] André Schiper Xavier Défago and Peter Urbán. Total Order Broadcast and Multicast Algorithms:Taxonomy and Survey. Technical report, December 2004.