Algoritmo de Difusão Atômica Rápido a Despeito de Colisões Tolerante a Falhas Bizantinas

Rodrigo Q. Saramago[1], Eduardo A. P. Alchieri[2], Tuanir F. Rezende[1], Lasaro Camargos[1]

[1] Faculdade de Ciência da Computação – Universidade Federal de Uberlândia (UFU)

[2]Departamento de Ciência da Computação – Universidade de Brasília (UNB) *

Abstract. *The inefficiency of Consensus-based Atomic Broadcast protocols in the presence of collisions (concurrent proposals) harms their adoption in the implementation of State Machine Replication. Proposals that are not decided in some instance of Consensus (commands not delivered) must be reproposed in a new instance, delaying their execution. The CFABCast algorithm (Collision-Fast Atomic Broadcast) uses M-Consensus, a Consensus variant, to decide and deliver multiple values in the same instance. However, CFABCast is not Byzantine fault tolerant, a requirement for many systems. Our first contribution is a variation CFABCast that handles Byzantine failures. Unfortunately, the resulting protocol is not collison-fast due to the possibility of Byzantine failures. In fact, our second contribution is the conjecture that there are no Byzantine collision-fast algorithm in the asynchronous model. Finally, our third contribution is a Byzantine collision-fast algorithm that bypasses our impossibility conjecture by using the USIG (Unique Sequential Identifier Generator) trusted component.*

Resumo. *O uso de protocolos de Difusão Atômica baseados em Consenso na implementação de Máquinas de Estados Replicadas esbarra na ineficiência destes protocolos na presença de colisões, isto é, propostas simultâneas. Isso porquê propostas não decididas no Consenso (comandos não entregues) devem ser repropostas em novas instâncias, aumentando seu tempo de entrega e atrasando sua execução. O algoritmo CFABCast (Collision-Fast Atomic Broadcast) utiliza uma variante do Consenso, o M-Consenso, para decidir e entregar múltiplos valores por instância. CFABCast, contudo, não tolera falhas bizantinas, requisito importante em diversos cenários. Como primeira contribuição deste artigo, propomos uma versão modificada do CFABCast que tolera falhas bizantinas. Apesar de ser baseado em um algoritmo rápido à despeito de falhas, nosso algoritmo não é rápido ante a possibilidade de falhas bizantinas. De fato, nossa segunda contribuição é a conjectura de que nenhum algoritmo possa sê-lo no modelo assíncrono. Finalmente, nossa terceira contribuição é a proposta de um algoritmo que é rápido à despeito de falhas bizantinas, contornando a impossibilidade pela extensão do modelo computacional com o componente seguro USIG (Unique Sequential Identifier Generator).*

*Este trabalho foi apoiado pela CAPES no contexto do projeto PVE CAPES 88881.062190/2014-01, pelo CNPq e pela Fapemig.

1. Introdução

Replicação de Máquinas de Estados, ou SMR (*state machine replication*), é uma técnica para obtenção de serviços tolerantes a falhas [Lamport 1978, Lampson 1996]. SMR pode ser implementada por meio de primitivas de Difusão Atômica, ou ABCast (*Atomic Broadcast*), que provê a entrega confiável e ordenada de mensagens a todos os destinatários não faltosos, que executam os comandos contidos nestas mensagens e, consistentemente, modificam seu estado. ABCast, por sua vez, é redutível ao problema de Consenso Distribuído da seguinte forma: sejam infinitas instâncias de Consenso, identificadas univocamente por um inteiro positivo; mensagens a serem difundidas são propostas na primeira instância ainda não decidida e a decisão da i-ésima instância de consenso é a i-ésima mensagem entregue atomicamente. Um problema dessa redução de SMR para Consenso, via ABCast, é que propostas não decididas (comandos não entregues) devem ser repropostas em novas instâncias, aumentando seu tempo de entrega e atrasando sua execução.

Algoritmos que evitam tais reproposições são denominados **rápidos à despeito de colisões** (*collision-fast*) e apresentam latência ótima de dois passos de comunicação. Todos os algoritmos conhecidos desta classe, contudo, possuem certas limitações. Por exemplo, [Mao et al. 2008] não é rápido entre a falha de algum processo e o reinício do algoritmo e [Du et al. 2014] usa relógios sincronizados. Outra importante limitação é que nenhum algoritmo tolera falhas bizantinas, sendo este trabalho o primeiro esforço no sentido de superar tal limitação.

De uma forma geral, este trabalho apresenta três contribuições principais. Primeiro, propomos uma variante do algoritmo CFABCast (*Collision-Fast Atomic Broadcast*) [Schmidt et al. 2014]. Diferentemente do CFABCast, o proposto aqui tolera falhas bizantinas, apesar de em menor número ($f < n/5$ em vez de $f < n/2$), e não é rápido ante a possibilidade de falhas. De fato, nossa segunda contribuição é a conjectura de que nenhum algoritmo possa sê-lo no modelo assíncrono. Finalmente, nossa terceira contribuição é um algoritmo que considera um modelo estendido pelo componente seguro USIG (*Unique Sequential Identifier Generator*) [Veronese et al. 2013], chamado USIG-BCFABCast (*USIG based Byzantine Collision-Fast Atomic Broadcast*). O algoritmo resultante, além de ser rápido à despeito de falhas, tolera $f < n/2$ falhas, mesmo limiar para falhas do tipo *crash* [Schmidt et al. 2014].

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta a fundamentação conceitual para os protocolos propostos, enquanto que os trabalhos relacionados são discutidos na Seção 3. A Seção 4 apresenta o primeiro algoritmo e discute a impossibilidade de um protocolo *collison-fast* no modelo assíncrono. A Seção 5 mostra o protocolo USIG-BCFABCast, que usa o componente seguro USIG para implementar um protocolo *collison-fast* mesmo com a possibilidade de falhas bizantinas. Na Seção 6 são apresentadas nossas considerações finais.

2. Conceitos Fundamentais

2.1. Modelo Computacional

Modelamos problemas de acordo em termos dos papéis desempenhados por agentes nos protocolos, como em [Lamport 1998]. Agentes são entidades que executam alguma tarefa computacional, e.g. processos, *threads*, atores, etc. Por exemplo, no Consenso Distribuído estão presentes os seguintes tipos de agentes:

- **Proposers** (P) propõe valores;
- **Acceptors** (A) cooperam na escolha de um valor como decisão; e
- **Learners** (L) aprendem o valor decidido.

Agentes se comunicam por troca de mensagens em um modelo computacional assíncrono, i.e., no qual não há limite no tempo de transmissão das mensagens ou ações executadas pelos agentes. Um agente é correto se não é falho, e é falho se sua execução foge da especificação do protocolo, podendo inclusive executar ações arbitrárias e maliciosas (bizantinas). Mensagens podem ser perdidas ou duplicadas mas não indetectavelmente corrompidas, e se repetidamente reenviadas de um agente correto para outro, são entregues em algum momento.

Consideramos também que cada agente possui um par de chaves (pública e privada) usadas para gerar e verificar assinaturas digitais; todo agente conhece apenas sua própria chave privada, mas todas as chaves públicas. Somente mensagens corretamente assinadas são processadas nos receptores. Representamos uma mensagem m assinada por uma agente x como $\langle m \rangle_{\sigma_x}$.

2.2. M-Consensus

O algoritmo Collision-Fast Atomic Broadcast (CFABcast) usa instâncias do algoritmo Collision-Fast Paxos, que resolve o problema M-Consenso. Nesta variante do Consenso Distribuído, agentes devem concordar em um mapeamento de *proposers* para valores propostos ou um valor especial *Nil*[Schmidt et al. 2014]. Esse mapeamento é capturado pela estrutura de dados *v-mapping*, i.e. mapa de valores, ou v-map, que contêm um conjunto, potencialmente vazio, de funções que mapeiam *proposers* a valores propostos ou *Nil*.

Os mapas possuem uma relação de precedência, de forma que se pode estabelecer uma ordem total entre eles (e.g. usando um identificador único para cada *proposer*). Um v-map com apenas um elemento em seu Domínio (i.e. conjunto de *proposers*) é denominado simples ou um *single-mapping*. Um *v-mapping* é dito trivial se todos os elementos de seu domínio mapeiam para o valor *Nil*. Finalmente, dois *v-mapping* são compatíveis se os elementos na intersecção de seus domínios mapeiam para os mesmos valores. Em outras palavras, estes mapas podem ser estendidos até que se tornem iguais, pois não há discordância entre eles. Assim, em um algoritmo de M-Consenso, *proposers* propõem valores e *learners* aprendem v-map, possivelmente diferentes mas sempre compatíveis, e que são estendidos por mapas triviais até que sejam iguais.

Formalmente, o M-Consenso é definido pelas seguintes propriedades, onde $learned[l]$ é o valor aprendido pelo $learner\ l \in L$, inicialmente $\perp$ (vazio), e $s \sqsubseteq r$ significa que s é prefixo de r e que r é uma extensão de s:

- *Não trivialidade:* $\forall l \in L, learned[l] = \perp$, $learned[l]$ é sempre um v-map proposto e não-trivial.
- *Estabilidade:* $\forall l \in L$, se $learned[l] = s$ em algum momento, então $s \sqsubseteq learned[l]$ em todos momentos posteriores.
- *Consistência:* O conjunto de v-map aprendidos é sempre compatível e não-trivial.

O objetivo do M-Consenso é fazer com que todos os *learners* aprendam, em algum momento, um *v-map* completo, isto é, um *v-map* em que todos os agentes que propuseram valores, são mapeados para os valores propostos ou para Nil. Isso só é possível se existir

um quorum de *acceptors* funcional durante a execução do algoritmo, sendo que um subconjunto dos *acceptors* forma um quorum se tem interseção não vazia com qualquer outro quorum. Formalmente, o progresso no M-Consenso é dado pela seguinte propriedade:

- *Progresso:* Para qualquer $p \in P$ e $l \in L$, se p, l e um quorum de *acceptors* não são falhos e p propõe um valor, então $learned[l]$ se tornará completo em algum momento.

2.3. Difusão Atômica

Difusão Atômica é definida pelas seguintes propriedades, onde $delivered[l]$ é a sequência de mensagens entregues pelo *learner* $l \in L$, inicialmente vazia e $\sqsubseteq$ é o operador de prefixo entre sequências:

- *Não trivialidade:* $\forall l \in L, delivered[l]$ contém apenas mensagens difundidas e não duplicadas.
- *Estabilidade:* $\forall l \in L$, se $delivered[l] = s$ em algum momento, então $s \sqsubseteq delivered[l]$ em todos momentos posteriores.
- *Consistência:* $\forall l_1, l_2 \in L$, ou $delivered[l_1] \sqsubseteq delivered[l_2]$ ou $delivered[l_2] \sqsubseteq delivered[l_1]$.

Como mostrado em [Schmidt et al. 2014], Difusão Atômica pode ser reduzido ao problema de M-Consensus de forma semelhante à redução ao Consenso, com a vantagem de entregar múltiplas mensagens por instância de M-Consensus em vez de apenas uma.

3. Trabalhos Relacionados

Esta seção descreve os trabalhos relacionados com os protocolos aqui propostos. Primeiramente, os algoritmos do Paxos e algumas variantes são discutidos e as principais diferenças entre estes protocolos são destacadas. Por fim, o algoritmo *Collision-fast Atomic Broadcast* (CFABCast) é apresentado, pois o mesmo deve ser instanciado com os protocolos propostos nas seções seguintes para a concretização das variantes bizantinas: BCFABCast e USIG-BCFABCast.

3.1. Paxos e Variantes

Paxos é um protocolo de consenso [Lamport 1998] notório por seu uso na indústria, e.g., *Chubby Lock Service* [Burrows 2006]. No *Paxos*, as propostas (comandos) são enviadas pelos clientes para um *proposer* eleito *coordenador* para que as repasse aos *acceptors* para serem decididas e entregues. Desde que não haja mais de um processo que se julgue coordenador e que exista um quorum de *acceptors* corretos e alcançáveis, a instância decidirá.

O algoritmo é executado em rodadas, e cada rodada possui duas fases, com dois passos de comunicação cada; a fase 1 é de preparação do protocolo e a fase 2 é que efetivamente leva a uma decisão. Em um cenário em que múltiplas instâncias são necessárias, a fase 1 pode ser executada em paralelo para todas as instâncias, amortizando seu custo, e reduzindo a decisão à execução da fase 2. Assim, se o coordenador faz uma proposta, o valor pode ser decidido em dois passos de comunicação. Para os demais *proposers*, 3 passos são necessários.

Fast Paxos [Lamport 2006a] é um algoritmo que "contorna" o coordenador para reduzir o tempo de decisão para dois passos para múltiplos *proposers*; algoritmos similares são conhecidos em geral como rápidos (*fast*). Contudo, contornar o coordenador pode levar o algoritmo a não progredir na presença de colisões, isto é, propostas concorrentes para a mesma instância. Algoritmos *Collision-Fast* [Lamport 2006b] decidem mesmo na presença de colisões mas, como todo algoritmo de Consenso, decidem por apenas um valor ao final de cada instância. O mesmo é válido para versões bizantinas destes protocolos, como Bizantine Paxos (BFT Paxos)[Castro and Liskov 2002] e Bizantine Fast Paxos (FAB)[Martin and Alvisi 2006]. Assim, na redução clássica de Difusão Atômica para Consenso Distribuído, mensagens propostas mas não decididas em uma instância precisam ser repropostas, aumentando a sua latência. CFABcast [Schmidt et al. 2014], discutido a seguir, contorna este problema permitindo que múltiplos valores sejam decididos por instância. Até agora, contudo, este protocolo não tinha uma versão que tolerasse falhas bizantinas.

3.2. Collision-fast Atomic Broadcast

Em [Schmidt et al. 2014] os autores demonstram a redução da Difusão Atômica ao problema de M-Consensus. Em específico, apresentam o algoritmo *Collision-fast Atomic Broadcast* (CFABCast), reproduzido aqui como Algorithm 1. O CFAbcast utiliza infinitas instâncias de *Collision-fast Paxos* (CFPaxos), que resolve M-Consensus, para entregar diversas mensagens em paralelo. As ações do CFABCast são diretamente mapeadas em ações do CFPaxos e, de forma simplificada, consistem em escolher a instância CFPaxos com menor identificador e que ainda não tenha decidido ou proposto nesta instância a mensagem a ser entregue. Para mais detalhes, recomendamos a leitura de [Schmidt et al. 2014], que inclui detalhada prova de corretude. É importante notar que para aumentar as chances de terminação no CFPaxos, somente um subconjunto dos *proposers*, os *collision-fast proposers* ou CF-proposers, pode propor em determinada rodada. Este subconjunto é determinado pelo coordenador da rodada, seguindo algum oráculo que monitora o sistema [Saramago 2016].

Algorithm 1 *Collision-fast Atomic Broadcast [Schmidt et al. 2014]*

I: the set of all Collision-fast Paxos instances used
$CFP(i)!A$: the action or variable A of Collision-fast Paxos instance i

1: $Propose(p, V) \triangleq$
2: $\quad \forall i \in I, CFP(i)!Propose(p, V)$
3: $NewPhase1a(i, c, r) \triangleq$
4: $\quad$ **pre-conditions:**
5: $\quad\quad c = C(r)$
6: $\quad\quad crnd[c] < r$
7: $\quad\quad c$ believes itself to be the leader
8: $\quad\quad c$ heard of a round $r > j > crnd[c]$ for some instance or $CF(crnd[c]) \notin active[c]$
9: $\quad$ **actions:**
10: $\quad\quad CFP(i)!Phase1a(c, r)$
11: $Phase1a(c, r) \triangleq$
12: $\quad \forall i \in I, CFP(i)!NewPhase1a(i, c, r)$
13: $Phase1b(a, r) \triangleq$
14: $\quad \forall i \in I, CFP(i)!Phase1b(a, r)$
15: $Phase2Start(c, r) \triangleq$
16: $\quad \forall i \in I, CFP(i)!Phase2Start(c, r)$
17: $Phase2Prepare(p, r) \triangleq$
18: $\quad \forall i \in I, CFP(i)!Phase2Prepare(p, r)$
19: $Phase2a(p, r, V) \triangleq$
20: $\quad$ **pre-condition:**
21: $\quad\quad p$ has not yet proposed V
22: $\quad$ **action:**
23: $\quad\quad$ LET $i = Min([j : CFP(j)!pval[p] = none])$
24: $\quad\quad CFP(i)!Phase2a(p, r, V)$
25: $Phase2b(i, a, r) \triangleq$
26: $\quad CFP(i)!Phase2b(a, r)$
27: $Learn(i, l) \triangleq$
28: $\quad CFP(i)!Learn(l)$

Enquanto [Schmidt et al. 2014] consideravam apenas falhas do tipo *crash*, neste trabalho estamos interessados em falhas bizantinas. Assim, apresentamos duas variantes

do CFPaxos que toleram tais falhas e que podem ser usadas sem adaptações pelo CFAB-Cast. A primeira variante é denominada BCFABCast e é o primeiro algoritmo bizantino para o M-Consensus; apesar de ser baseada no CFPaxos, não é *collision-fast* e é apresentada como um passo intermediário para o nosso algoritmo final, USIG-BCFABCast.

3.3. Resumo

Para facilitar a comparação dos algoritmos discutidos aqui, a Tabela 1 apresenta várias métricas dos mesmos, sendo que as duas últimas colunas referem-se aos protocolos que estamos propondo.

	Paxos	Fast Paxos	CFABCast	BFT Paxos	BCFABCast	USIG-BCFABCast
#passos	3	2	2	4	3	2
#acceptors	2f + 1	3f + 1	2f + 1	3f + 1	5f + 1	2f + 1
#quorum	f + 1	2f + 1	f + 1	2f + 1	4f + 1	f + 1
Falha	Crash	Crash	Crash	Bizantinas	Bizantinas	Bizantinas
Componente	–	–	–	–	–	USIG

Tabela 1. Para cada protocolo, considerando até f falhas: número de passos de comunicação para se alcançar acordo; número de réplicas desempenhando o papel de *acceptors* necessárias para garantir corretude; tamanho dos quóruns de *acceptors*; tipo de falha suportada; e necessidade de um componente seguro.

4. *Byzantine* Collision-Fast Paxos

No Algorithm 2 introduzimos o Byzantine Collision-fast Paxos, que resolve M-Consenso na presença de falhas bizantinas. Quando usado em conjunto com o Algoritmo 1, resolve o problema de Difusão Atômica na presença de falhas bizantinas. Contudo, apesar do nome e de ser derivado do *Collision-Fast Paxos*, o protocolo não é rápido à despeito de colisões. Na Seção 5 sanaremos esta deficiência.

4.1. Visão Geral

Como outras variantes do Paxos, cada instância do nosso algoritmo é executada em rodadas divididas em duas fases. Na fase 1 o coordenador questiona os *acceptors* (ação $Phase1a$) por valores aceitos em rodadas anteriores. Baseado nas respostas dos *acceptors* (emitidas na ação $Phase1b$), o coordenador estabelece se algum v-map já foi possivelmente decidido (ação $Phase2Start$).

Observe que o protocolo garante que se algum mapeamento tiver sido decidido então pelo menos $2f + 1$ mensagens $1b$ conterão tal mapeamento. Seja S o conjunto de todos os *single-maps* aceitos por $2f + 1$ acceptors. Se S não é vazio, o coordenador informa os CF-proposers, para que se abstenham de propor nesta rodada, e os *acceptors*, para que aceitem os mapeamentos em S e informem os *learners* (ação $Phase2b$). Assim, o coordenador tenta terminar a instância e progredir na computação.

Caso contrário, o coordenador informa os *CF-proposers* da rodada sendo iniciada, para que proponham seus valores para os *acceptors* (ação $Phase2Prepare$). Autorizado pelo coordenador, um *CF-proposer* pode repassar o valor em mensagem $\langle$“`propose`”, _$\rangle$ recebida de um *proposer* qualquer, incluindo si mesmo (ação $Phase2a$). *CF-proposers* repassam a mensagem $2S$ recebida do coordenador para os *acceptors* para provarem que tem permissão de propor.

Algorithm 2 *Byzantine Collision-fast Paxos*

Pr, A, L: proposers, acceptors and learners sets;
$CF(i)$: round i's collision-fast proposers set;
$C(i)$: round i's coordinator.
$prnd[p]$, $crnd[c]$, $rnd[a]$: current round of proposer p, coordinator c, and acceptor a, respectively, initially 0.
$pval[p]$: value p has fast-proposed at $prnd[p]$ or $none$ if p has not fast-proposed at $prnd[p]$, initially $none$.
$cval[c]$: initial v-mapping for $crnd[c]$, if c has queried an acceptor quorum or $none$ otherwise; initially $\perp$ for coordinator of round 0 and $none$ for others.
$vrnd[a]$: round at which a has accepted its latest value.
$vval[a]$: v-mapping a has accepted at $vrnd[a]$ or $none$ if no value accepted at $vrnd[a]$; initially $none$.
$learned[l]$: v-mapping currently learned by learner l; initially $\perp$.
$m \Leftarrow s$: received message m from source s
$m \Rightarrow d$: send message m to destination d

1: $Propose(p, V) \triangleq$
2: **pre-condition:**
3: $p \in Pr$
4: **action:**
5: $\langle$"propose", $V\rangle_{\sigma_p} \Rightarrow cf \in CF(prnd[p])$
6: $Phase1a(c, r) \triangleq$
7: **pre-conditions:**
8: $c = C(r)$
9: $crnd[c] < r$
10: **actions:**
11: $crnd[c] \leftarrow r$
12: $cval[c] \leftarrow none$
13: $\langle$"1a", $r\rangle_{\sigma_c} \Rightarrow A$
14: $Phase1b(a, r) \triangleq$
15: **pre-conditions:**
16: $a \in A$
17: $rnd[a] < r$
18: $\langle$"1a", $r\rangle_{\sigma_c} \Leftarrow C(r)$
19: **actions:**
20: $rnd[a] \leftarrow r$
21: $\langle$"1b", $r, vrnd[a], vval[a]\rangle_{\sigma_a} \Rightarrow C(r)$
22: $Phase2Start(c, r) \triangleq$
23: **pre-conditions:**
24: $c = C(r)$
25: $crnd[c] = r$
26: $cval[c] = none$
27: $\exists Q \subseteq A$:
28: Q is a quorum
29: $\forall a \in Q, \langle$"1b", $r, vrnd, vval\rangle_{\sigma_a} \Leftarrow a$
30: **actions:**
31: LET $1bs = [\ m = \langle$"1b", _, _, _$\rangle_{\sigma_a} : m \Leftarrow a \in Q\ ]$
32: LET $k = Max([vrnd : \langle$"1b", _, $vrnd$, _$\rangle_{\sigma_a} \in 1bs\ ])$
33: LET $A_{\langle p,v\rangle} = [a : \langle$"1b", $r, k, vval\rangle_{\sigma_a} \in 1bs \wedge vval[p] = v]$
34: LET $S = \left[\langle p, v\rangle : A_{\langle p,v\rangle} \geq 2f + 1\right]$
35: IF $S = \emptyset$ THEN
36: $cval[c] \leftarrow \perp$
37: $\langle$"2S", $r, cval[c], msgs\rangle_{\sigma_c} \Rightarrow CF(r)$
38: ELSE
39: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in CF(r)]$
40: $\langle$"2S", $r, cval[c], msgs\rangle_{\sigma_c} \Rightarrow CF(r) \cup A$
41: $Phase2Prepare(p, r) \triangleq$
42: **pre-conditions:**
43: $p \in CF(r)$
44: $prnd[p] < r$
45: $\langle$"2S", $r, v, proofs\rangle_{\sigma_{C(r)}} \Leftarrow C(r)$
46: goodRoundValue(r,v,proofs)
47: **actions:**
48: $prnd[p] \leftarrow r$
49: $proof[p] \leftarrow proofs$
50: IF $v = \perp$ THEN $pval[p] \leftarrow none$
51: ELSE $pval[p] \leftarrow v(p)$
52: $Phase2a(p, r, V) \triangleq$
53: **pre-conditions:**
54: $p \in CF(r)$
55: $prnd[p] = r$
56: $pval[p] = none$
57: **either** $(V \neq Nil \wedge \langle$"propose", $V\rangle_{\sigma_p} \Leftarrow p \in Pr)$
58: **or** $(V = Nil \wedge \langle$"2a", $r, \langle q, W\rangle\rangle_{\sigma_q} \Leftarrow q \in CF(r) \wedge W \neq Nil)$
59: **actions:**
60: $pval[p] \leftarrow V$
61: IF $V \neq Nil$ THEN
62: $\langle$"2a", $r, \langle p, V\rangle, proof[p]\rangle_{\sigma_p} \Rightarrow A \cup CF(r)$
63: ELSE
64: $\langle$"2a", $r, \langle p, V\rangle, proof[p]\rangle_{\sigma_p} \Rightarrow A$
65: $Phase2b(a, r) \triangleq$
66: LET $Cond1 =$
67: $vval[a] = none \vee$
68: $(\langle$"2S", $r, v, proofs\rangle_{\sigma_c} \Leftarrow C(r) \wedge$
69: goodRoundValue(r, v, proofs)
70: $v \neq \perp \wedge vrnd[a] < r)$
71: LET $Cond2 =$
72: $\langle$"2a", $r, \langle p, V\rangle, proofs\rangle_{\sigma_p} \Leftarrow p \in CF(r) \wedge V \neq Nil$
73: goodRoundValue(r, V, proofs)
74: **pre-conditions:**
75: $a \in A$
76: $rnd[a] \leq r$
77: $Cond1 \vee Cond2$
78: **actions:**
79: IF $Cond1$
80: THEN $vval[a] \leftarrow v$
81: ELSE
82: IF $Cond2 \wedge (vrnd[a] < r \vee vval[a] = none)$
83: THEN $vval[a] \leftarrow \perp \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
84: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
85: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
86: $\langle$"2b", $r, vval[a]\rangle_{\sigma_a} \Rightarrow L$
87: $Learn(l) \triangleq$
88: **pre-conditions:**
89: $l \in learners$
90: $\exists Q \subseteq A$:
91: Q is a quorum
92: $\forall a \in Q, \langle$"2b", $r, \_\rangle_{\sigma_a} \Leftarrow a$
93: **actions:**
94: $Q2bVals = \{v : \langle$"2b", $r, v\rangle_{\sigma_a} \Leftarrow a \in Q\}$
95: $w = \sqcap Q2bVals$
96: $learned[l] = learned[l] \sqcup w$

Para que um coordenador bizantino não possa ignorar as propostas já aceitas, propondo qualquer valor na nova rodada e violando a propriedade de acordo do consenso, o coordenador prova que executou a fase 1 corretamente. Isto é feito enviando-se o conjunto das mensagens recebidas de um quórum de *acceptors* e usadas para computar S (linhas

31-35) nas mensagens $2S$. Os agentes receptores fazem o mesmo cálculo para verificar se o valor proposto realmente é válido para a rodada em questão (função $goodRoundValue$).

Ao receberem propostas e verificarem sua validade (ação $Phase2b$), *acceptors* estendem os v-maps que já tenham aceito e repassam para os *learners*. *Learners*, por sua vez, coletam, dos v-maps recebidos dos *acceptors*, os *single-maps* aceitos por um quorum (linha 93), e os agregam ao v-map aprendido (ação $Learn$).

4.2. Corretude

Em alto nível, a corretude do BCFABCast é herdada do CFABCast, pois a execução do primeiro imita a execução do segundo, com alguns passos extra para coibir agentes de agir maliciosamente. Estas diferenças se concentram nos *acceptors* e *CF-proposers*. Isto porquê *learners* bizantinos não podem atrapalhar outros agentes, uma vez que não enviam mensagens. Contudo, podem aprender qualquer valor, mas, até onde sabemos, este problema é incontornável. Quanto a *proposers*, estes são normalmente clientes do sistema e é difícil impedir que sejam comprometidos. Contudo, mesmo um *proposer* malicioso não pode deixar de seguir o protocolo, pois a única ação que executa é propor valores. Neste trabalho desconsideramos ataques de negação de serviço, que podem ser tratados usando técnicas específicas descritas na literatura.

De um total de $5f$ acceptors, $4f + 1$ devem ser corretos. Assim, de um quórum de $4f+1$, garantidamente $3f+1$ serão corretos. *learners* esperam por $4f+1$ confirmações do mesmo mapeamento antes de aprendê-lo, aprendendo apenas valores aceitos por um quorum e também detectar caso um agente se comporte de forma inconsistente. Na pior das hipóteses, acceptors e *CF-Proposer* bizantinos podem entrar em conluio para aceitarem diferentes valores, repassados a *learners* diferentes. Embora este ataque possa impedir uma rodada de terminar, ela não pode levar a decisões conflitantes.

Um coordenador correto que inicie uma nova rodada também pode, em virtude dos quóruns de $4f + 1$ *acceptors*, identificar qualquer mapeamento possivelmente decidido, uma vez que o mesmo deve ter sido aceito por pelo menos $2f + 1$ dos *acceptors* que responderem com mensagens $1b$, pois no máximo f *acceptors* bizantinos e f *acceptors* corretos podem reportar outros valores.

Se o coordenador for bizantino, este poderia ignorar as mensagens recebidas dos *acceptors* durante a fase 1 na tentativa de violar a propriedade de acordo do protocolo. Por isso o protocolo exige que o coordenador prove que executou a fase 1 corretamente. A prova é constituída das mensagens $\langle$"1b", $a\rangle$ssinadas, previamente recebidas de um quorum, que o coordenador não conseguiria forjar. Logo, um coordenador não consegue manipular o que já foi aprendido sem ser detectado, embora possa forçar reconfigurações corretas indefinidamente, comprometendo o progresso do protocolo. Este último problema pode ser resolvido utilizando-se, por exemplo, o mecanismo de Multi-Coordenação proposto em [Camargos et al. 2007] ou o rodízio do papel do coordenador, limitando seu ataque.

4.3. Impossibilidade de *Byzantine Collision-fast Atomic Broadcast*

No CFABCast [Schmidt et al. 2014], se um *CF-proposer* i não tem um valor a propor e recebe a proposta de outro *CF-proposer* j, i envia uma mensagem diretamente aos *learners*, avisando que se absterá de propor nesta rodada. *learner*, ao receber a mensagem de

i, aprende que i não poderá ser mapeado a outro valor, não precisando do aceite dos *acceptors*. Este cenário é apresentado na Figura 1, à esquerda, em que há dois *Collision-fast proposer* CFP_0 e CFP_1, e nenhuma falha, bizantina ou não, acontece. Assim, o protocolo consegue terminar em dois passos de comunicação sempre que todas as propostas forem feitas concorrentemente.

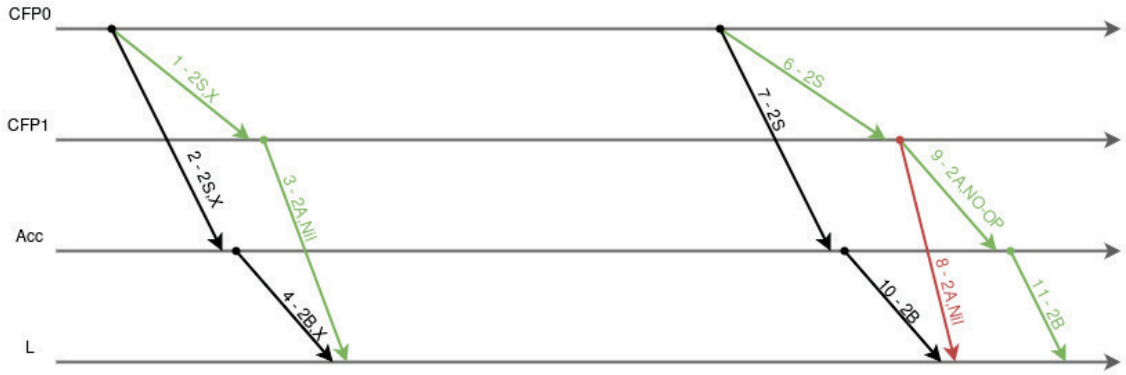

Figura 1. Quantidade de passos de comunicação necessários para se decidir um *v-mapping* completo no CFPaxos (esquerda) e BCFPaxos (direita).

Se a mesma abordagem fosse possível no BCFABCast, então seria possível que um *CF-Proposer* bizantino dissesse que está se abstendo ao mesmo tempo que envia uma proposta aos *acceptors*. Este cenário é apresentado na Figura 1, do lado direito, onde o CFP_0 propôs algum valor e CFP_1 propôs *Nil* direto para o *learner*, Para evitar esse caso, no BCFABCast não há contato direto entre *CF-Proposer* e *learner* e mesmo as abstenções devem ser confirmadas por um quorum antes de se tornarem parte do v-map aprendido.

Para que um *CF-proposer* não possa propor *Nil* continuamente, impedindo o progresso do protocolo, exigimos que o mesmo só proponha caso tenha recebido mensagem "$propose''$ ou "$2a$". Assim, caso um *CF-Proposer* não tenha nada a propor, serão necessários até três passos de comunicação para terminar o protocolo: 1 passo para receber a mensagem "$2a$" e mais 2 passos para enviar a abstenção aos *learners* via os acceptors.

Nos parece que tal restrição seja válida a qualquer protocolo e , assim, conjecturamos que nenhum protocolo bizantino possa ser *collision fast*, ou seja, decidir em dois passos de comunicação na possível presença de agentes bizantinos. Fazê-lo implicaria em confiar nas mensagens recebidas diretamente de um *CP-proposer*, talvez bizantino.

5. Estendendo o modelo de sistema com o uso de um componente seguro

Nesta seção mostramos que adicionando um componente seguro no sistema é possível reduzir o número tanto de réplicas (de $5f + 1$ para $2f + 1$ acceptors) quanto de passos de comunicação (de 3 para 2) necessários para resolver o BCFABCast. Particularmente interessante, mostramos que é possível resolver o BCFABCast com apenas 2 passos de comunicação, mesmo número necessário para resolver o CFABCast [Schmidt et al. 2014].

Os componentes seguros propostos para auxiliar no desenvolvimento de protocolos que tolerem falhas maliciosas possuem diferenças consideráveis tanto em aspectos de implementação quanto de desempenho. Quanto à implementação, estes componentes podem ser: i) implementados de forma distribuída, como o *Trusted Timely Computing Base*

(TTCB) [Correia et al. 2004]; ii) locais baseados em memória, como o *Attested Append-Only Memory* (A2M) [Chun et al. 2007]; e iii) locais baseados em um simples contador, como o *Unique Sequential Identifier Generator* (USIG) [Veronese et al. 2013].

Basicamente, um atacante não consegue acessar estes componentes, mesmo que consiga invadir um servidor. Com isso, é possível projetar protocolos que restringem as ações que um processo malicioso (invadido) pode executar sem ser descoberto. Neste trabalho escolhemos o componente seguro USIG [Veronese et al. 2013] pela sua simplicidade, além da fácil implementação e utilização no sistema.

5.1. USIG: Gerador de identificadores sequenciais únicos

Este componente é local em cada processo (proposer, acceptor ou learner) e será usado para atribuir um identificador único para uma mensagem, além de assiná-la. Para cada processo, os identificadores atribuídos às mensagens são únicos (um mesmo identificador nunca é atribuído a duas ou mais mensagens), monotônicos (o identificar atribuído a uma mensagem nunca é menor do que o anterior) e sequenciais (o identificador atribuído a uma mensagem sempre é o sucessor do anterior).

A interface definida para acessar este serviço é a seguinte [Veronese et al. 2013]:

- `createUI(m)`: retorna um certificado que contém um identificador único `UI` e a prova de que este identificador foi criado por este componente e atribuído a `m`. O identificador é basicamente um contador monotonicamente crescente que é incrementado a cada chamada desta função. Já a prova envolve criptografia e pode ser baseada em um *hash* (*Hash-based Message Authentication Code* – HMAC) ou em criptografia assimétrica (assinaturas digitais).
- `verifyUI(PK,UI,m)`: verifica se o identificador único `UI` é válido para `m`.

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

Este componente pode ser implementado de duas formas [Veronese et al. 2013]: é possível usar apenas *hashes* ou empregar assinaturas digitais. Neste trabalho adotaremos a solução que emprega assinaturas digitais, de forma que a verificação pode ser realizada fora do componente seguro, bastando a chave pública correspondente. Além disso, diferentes níveis de isolação podem ser empregados, usando máquinas virtuais ou até mesmo módulos seguros de hardware (*Trusted Platform Module* – TPM).

5.2. USIG-BCFABCast: USIG *based Byzantine Collision-fast Atomic Broadcast*

Esta seção apresenta o protocolo para resolver o BCFABCast usando o componente seguro USIG, chamado de USIG-BCFABCast (*USIG based Byzantine Collision-fast Atomic Broadcast*). Mais especificamente, o Algoritmo 3 substitui o Collision-Fast Paxos na resolução do M-Consensus e deve ser usado em conjunto com o Algoritmo 1. Devido ao uso deste componente seguro, o protocolo proposto necessita de apenas $2f + 1$ *acceptors* e 2 passos de comunicação, sendo rápido à despeito de colisões mesmo na presença de processos maliciosos.

A ideia geral é que através dos identificadores únicos adicionados às mensagens, os *collision-fast proposers* não precisam enviar suas mensagens através dos *acceptors* caso não tenham nada a propor, i.e., os *learners* podem confiar nas propostas, com valores nulos, recebidas diretamente dos *proposers* uma vez que eles não conseguem enviar propostas diferentes com o mesmo identificador. Note que as propostas com valores continuam seguindo o caminho normal através dos *acceptors*, visto que caso o coordenador seja trocado (devido a falhas ou assincronismos) estes valores estão armazenados nos *acceptors* e são usados para configuração do novo *round* como no algoritmo anterior (Seção 4).

5.2.1. Visão Geral

O protocolo proposto é apresentado no Algoritmo 3. As fases iniciais de configuração de um novo *round* pelo novo coordenador não precisam de grandes alterações, pois caso um coordenador envie diferentes valores para o *round r* nas mensagens $1a$, o mesmo não vai conseguir uma prova para reconfigurar o sistema com as mensagens $2S$. De fato, a primeira alteração ocorre nas mensagens $2S$, que carrega um identificador único gerado pelo USIG do coordenador e impossibilita que mensagens diferentes sejam enviadas a agentes (processos) diferentes. Consequentemente, todos os agentes recebem a mesma configuração inicial para o *round r*.

Cada *CF-proposer* também adiciona um identificador único às suas propostas nas mensagens $2a$, impossibilitando que propostas diferentes sejam enviadas para agentes diferentes. Apesar destas mensagens serem para todos os agentes ($A \cup CF(r) \cup L$), apenas os *learners* processam as propostas com Nil (linhas 46,60 e 86). Isso é necessário para que os *acceptors* e os outros *CF-Proposer* possam incrementar seus contadores para as propostas esperadas mesmo quando a proposta é Nil. Por outro lado, os *learners* incrementam seus contadores para propostas diferentes de Nil quando recebem estas propostas dos *acceptors* (linha 84). Note também que os identificadores únicos sempre acompanham suas respectivas propostas, de forma que quando uma nova rodada precisa ser iniciada, os mesmos são enviados pelos *acceptors* para o novo coordenador, que os encaminha nas mensagens $2S$ para então serem repassados pelos *acceptors*, aos *learners* nas mensagens $2b$ (linhas 69 e 76).

Finalmente, vale destacar que os *acceptors* também adicionam um identificador único às mensagens $2b$ enviadas aos *learners* (linhas 75–76). Consequentemente, um *acceptor* malicioso é incapaz de enviar diferentes mensagens para diferentes *learners* sem ser descoberto. De fato, os *learners* apenas processam a mensagem com o contador imediatamente seguinte na sequência (linha 84).

5.2.2. Corretude

Através do uso de identificadores sequenciais únicos, o protocolo proposto no Algoritmo 3 impede que um agente malicioso (coordenador, *acceptor* ou *proposer*) envie mensagens diferentes para diferentes agentes em um mesmo passo do algoritmo. Isto torna o funcionamento do protocolo semelhante ao [Schmidt et al. 2014], que tolera apenas falhas por *crash*: ou um agente envia um mesmo conteúdo (que pode ser forjado) para todos

Algorithm 3 USIG *based Byzantine Collision-fast Paxos*

All variables and sets from Algorithm 2
$USIG^C[c]$, $cnt[c]$: USIG used by coordinator c and expected counter value for msgs received from c (initially, 0)
$USIG^A[a]$, $cnt[a]$: USIG used by acceptor a and expected counter value for msgs received from a (initially, 0)
$USIG^P[p]$, $cnt[p]$: USIG used by CF proposer p and expected counter value for msgs received from p (initially, 0)
verifyCnt(UI_x,$cnt[x]$) $\equiv$ **if**($UI_x.cnt = cnt[x]$)**then**$cnt[x]$++; *return true*; **else** *return false*;

1: $Propose(p, V) \triangleq$
2: lines 2 – 5 of Algorithm 2

3: $Phase1a(c, r) \triangleq$
4: lines 7 – 13 of Algorithm 2

5: $Phase1b(a, r) \triangleq$
6: lines 15 – 21 of Algorithm 2

7: $Phase2Start(c, r) \triangleq$
8: **pre-conditions:**
9: $c = C(r)$
10: $crnd[c] = r$
11: $cval[c] = none$
12: $\exists Q \subseteq A$:
13: Q is a quorum
14: $\forall a \in Q, \langle$"1b", $r, vrnd, vval\rangle_{\sigma_a} \Leftarrow a$
15: **actions:**
16: LET $msgs = [\ m = \langle$"1b", $r, vrnd, vval\rangle_{\sigma_a} : m \Leftarrow a \in Q\ ]$
17: LET $k = Max([vrnd : \langle$"1b", $r, vrnd, vval\rangle_{\sigma_a} \in msgs\ ])$
18: LET $S = [vval : \langle$"1b", $r, k, vval\rangle_{\sigma_a} \in msgs, vval \neq none]$
19: IF $S = \emptyset$ THEN
20: $cval[c] \leftarrow \bot$
21: ELSE
22: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil, Nil\rangle : p \in CF(r)]$
23: $UI_c \leftarrow USIG^C[c]$.createUI($\langle$"2S", $r, cval[c], msgs\rangle$)
24: $\langle$"2S", $r, cval[c], msgs, UI_c\rangle \Rightarrow CF(r) \cup A$

25: $Phase2Prepare(p, r) \triangleq$
26: **pre-conditions:**
27: $p \in CF(r)$
28: $prnd[p] < r$
29: $\langle$"2S", $r, v, proof, UI_{C(r)}\rangle \Leftarrow C(r)$
30: goodRoundValue(r,v,proofs)
31: verifyUI($PK_{C(r)}$,$UI_{C(r)}$,$\langle$"2S", $r, v, proofs\rangle$)
32: verifyCnt($UI_{C(r)}$,$cnt[C(r)]$)
33: **actions:**
34: $prnd[p] \leftarrow r$
35: IF $v = \bot$ THEN $pval[p] \leftarrow none$
36: ELSE $pval[p] \leftarrow v(p)$

37: $Phase2a(p, r, V) \triangleq$
38: **pre-conditions:**
39: $p \in CF(r)$
40: $prnd[p] = r$
41: $pval[p] = none$
42: **either** ($V \neq Nil \wedge \langle$"propose", $V\rangle_{\sigma_p} \Leftarrow p \in Pr$)
43: **or** ($V = Nil \wedge$
44: $\langle$"2a", $r, \langle q, W\rangle, UI_q\rangle \Leftarrow q \in CF(r) \wedge$
45: verifyUI(PK_q,UI_q,$\langle q, W\rangle$) $\wedge$
46: verifyCnt(UI_q,$cnt[q]$) $\wedge W \neq Nil$)
47: **actions:**
48: $pval[p] \leftarrow V$
49: $UI_p \leftarrow USIG^P[p]$.createUI($\langle p, V\rangle$)
50: $\langle$"2a", $r, \langle p, V\rangle, UI_p\rangle \Rightarrow A \cup CF(r) \cup L$

51: $Phase2b(a, r) \triangleq$
52: LET $Cond1 =$
53: $vval[a] = none \vee$
54: ($\langle$"2S", $r, v, proofs, UI_{C(r)}\rangle \Leftarrow C(r) \wedge$
55: verifyUI($PK_{C(r)}$,$UI_{C(r)}$,$\langle$"2S", $r, v, proofs\rangle$) $\wedge$
56: verifyCnt($UI_{C(r)}$,$cnt[C(r)]$) $\wedge$
57: goodRoundValue(r,v,proofs) $\wedge$
58: $\wedge v \neq \bot \wedge vrnd[a] < r$)
59: LET $Cond2 =$
60: $\langle$"2a", $r, \langle p, V\rangle, UI_p\rangle \Leftarrow p \in CF(r) \wedge V \neq Nil \wedge$
61: verifyUI(PK_p,UI_p,$\langle p, V\rangle$) $\wedge$
62: verifyCnt(UI_p,$cnt[p]$)
63: **pre-conditions:**
64: $a \in A$
65: $rnd[a] \leq r$
66: $Cond1 \vee Cond2$
67: **actions:**
68: IF $Cond1$
69: THEN $vval[a] \leftarrow v$
70: ELSE
71: IF $Cond2 \wedge (vrnd[a] < r \vee vval[a] = none)$
72: THEN $vval[a] \leftarrow \bot \bullet \langle p, V, UI_p\rangle \bullet [\langle p, Nil, Nil\rangle : p \in Pr \setminus CF(r)]$
73: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V, UI_p\rangle$
74: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
75: $UI_a \leftarrow USIG^A[a]$.createUI($\langle$"2b", $r, vval[a]\rangle$)
76: $\langle$"2b", $r, vval[a], UI_a\rangle \Rightarrow L$

77: $Learn(l) \triangleq$
78: **pre-conditions:**
79: $l \in learners$
80: $\exists Q \subseteq A$:
81: Q is a quorum
82: $\forall a \in Q, \langle$"2b", $r, \_, UI_a\rangle \Leftarrow a \wedge$
83: verifyUI(PK_a,UI_a,$\langle$"2b", $r, \_\rangle$) $\wedge$
84: verifyCnt(UI_a,$cnt[a]$)
85: **actions:**
86: LET $P \subset CF(r) : \forall p \in P, \langle$"2a", $r, \langle p, Nil\rangle, UI_p\rangle \Leftarrow p \wedge$ verifyUI(PK_p,UI_p,$\langle p, Nil\rangle$) $\wedge$ verifyCnt(UI_p,$cnt[p]$)
87: $Q2bVals = [v : \langle$"2b", $r, v, UI_a\rangle \Leftarrow a \in Q \wedge$
88: $\forall \langle q, W, UI_q\rangle \in v$: verifyUI($PK_q$,$UI_q$,$\langle p, W\rangle$) $\wedge$
89: verifyCnt(UI_q,$cnt[q]$]
90: $w = \sqcap Q2bVals \bullet [\langle u, Nil, UI_u\rangle : u \in P]$
91: $learned[l] = learned[l] \sqcup w$

ou não envia nada.

Sendo assim, cada *collision-fast proposer* pode fazer uma única proposta para uma determinada *rodada* r. Caso a proposta seja Nil, então essa pode ser enviada diretamente para os *learners*, caso contrário, deve ser enviada para os *acceptors* para que fique armazenada caso seja necessário um novo *round* para finalizar aquela instância do consenso.

Desta forma, o algoritmo necessita de apenas 2 passos de comunicação para um valor proposto ser decido, sendo rápido à despeito de colisões mesmo na presença de agentes maliciosos.

Além disso, apenas $2f + 1$ *acceptors* são necessários, pois os mesmos não conseguem modificar uma proposta feita por um *CF-proposer*, pois o identificador único gerado pelo *proposer* sempre acompanha a proposta. Tampouco conseguem enviar diferentes mensagens para diferentes *learners*.

6. Conclusão

De forma geral as principais contribuições feitas neste trabalho são três: i) proposta de uma versão modificada do protocolo *Collision Fast Atomic Broadcast* que tolera falhas bizantinas mas que não é rápido a despeito de falhas bizantinas; ii) conjectura da impossibilidade de um protocolo de Consenso Bizantino rápido à despeito de falhas bizantinas em sistema parcialmente síncrono; e, iii) proposta de um algoritmo rápido à despeito de falhas bizantinas, que contorna a impossibilidade apresentada com o uso do componente seguro USIG (*Unique Sequential Identifier Generator*).

Em trabalhos futuros pretendemos primeiramente provar formalmente a conjectura apresentada aqui. Em segundo lugar, pretendemos avaliar experimentalmente os protocolos propostos, comparando-os com outros na literatura.

Referências

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems.

Camargos, L. J., Schmidt, R. M., and Pedone, F. (2007). Multicoordinated paxos. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 316–317, New York, NY, USA. ACM.

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

Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2007). Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 189–204, New York, NY, USA. ACM.

Correia, M., Neves, N. F., and Verissimo, P. (2004). How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183, Washington, DC, USA. IEEE Computer Society.

Du, J., Sciascia, D., Elnikety, S., Zwaenepoel, W., and Pedone, F. (2014). Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 343–354.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Lamport, L. (1998). The part-time parliament. *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, 16(2):133–169.

Lamport, L. (2006a). Fast paxos. *Distributed Computing*, 19(2):79–103.

Lamport, L. (2006b). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.

Lampson, B. (1996). How to build a highly available system using consensus. In Babaoglu, O. and Marzullo, K., editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg.

Mao, Y., Junqueira, F. P., and Marzullo, K. (2008). Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA. USENIX Association.

Martin, J.-P. and Alvisi, L. (2006). Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215.

Saramago, R. Q. (2016). Implementing and evaluating the collision-fast atomic broadcast. Master's thesis, Faculty of Computing, Federal University of Uberlândia, Brazil.

Schmidt, R., Camargos, L., and Pedone, F. (2014). Collision-fast atomic broadcast. In *Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, AINA '14, pages 1065–1072, Washington, DC, USA. IEEE Computer Society.

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