Usando Criptografia de Limiar para Tolerar Clientes Maliciosos em Memória Compartilhada Dinâmica*

Eduardo Adilio Pelinson Alchieri[1] **Alysson Neves Bessani**[2] **Joni da Silva Fraga**[3]

[1]CIC, Universidade de Brasília, Brasília - Brasil
[2]LaSIGE, Universidade de Lisboa, Lisboa - Portugal
[3]DAS, Universidade Federal de Santa Catarina, Florianópolis - Brasil

Resumo. *Sistemas de quóruns Bizantinos são ferramentas usadas na implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas arbitrárias. Vários protocolos para implementação destes sistemas foram propostos para ambientes estáticos e, mais recentemente, também surgiram propostas de protocolos para ambientes dinâmicos. Um dos desafios na implementação desses sistemas em ambientes dinâmicos está na reconfiguração do conjunto de servidores devido a entradas e saídas arbitrárias de processos. Este trabalho vai além e apresenta protocolos que toleram a presença de clientes maliciosos em sistemas de quóruns Bizantinos dinâmicos, através do emprego de um mecanismo de criptografia de limiar que fornece a flexibilidade suficiente para operação nestes ambientes. Este mecanismo é utilizado para controlar as ações que clientes maliciosos podem executar contra o sistema. Além disso, todos os protocolos apresentados são para sistemas assíncronos, não necessitando de nenhuma premissa temporal sobre o comportamento do sistema.*

Abstract. *Byzantine quorum systems are an useful tool to implement consistent and available data storage systems in the presence of arbitrary faults. Several protocols were proposed to implement these systems in static environments and, more recent, proposals have also appeared for dynamic environments. One of the challenges to implement these systems in dynamic environments is the reconfiguration of the servers set due to joins and leaves. In this work we go one step further and propose protocols that tolerate malicious clients in dynamic Byzantine quorum systems, by using a threshold cryptography scheme that provides flexibility necessary to operate in these environments. This scheme is used to control actions that malicious clients can execute against the system. Moreover, all proposed protocols are for asynchronous systems.*

1. Introdução

Sistemas de quóruns [Gifford 1979] são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. Além de serem blocos básicos de construção para protocolos de sincronização (ex.: consenso), o grande atrativo destes sistemas está relacionado com seu poder de escalabilidade e balanceamento de carga, uma vez que as operações não precisam ser executadas por todos os servidores do sistema, mas apenas por um quórum dos mesmos. A consistência de um sistema de quóruns é assegurada pela propriedade de intersecção dos subconjuntos de servidores (quóruns).

Sistemas de quóruns foram inicialmente estudados em ambientes estáticos, onde não é permitida a entrada e saída de servidores durante a execução do sistema [Bazzi and Ding 2004, Malkhi and Reiter 1998]. Esta abordagem não é adequada para sistemas que permanecerão em execução por um longo tempo, uma vez que, dispondo de um quantidade suficiente de tempo,

*Este trabalho recebeu apoio do DPP/UnB através do Edital 10/2012.

um adversário pode comprometer um número maior de servidores do que o tolerado e então quebrar as propriedades do sistema. Outra limitação é que estes protocolos não permitem que um administrador, em tempo de execução, adicione máquinas no sistema (para suportar um aumento na carga de processamento) ou troque máquinas antigas. Estes protocolos também não são adequados para sistemas distribuídos modernos, desenvolvidos para redes móveis e auto-organizáveis (e.g., MANETs, P2P), onde, pela sua própria natureza, o conjunto de processos que compõem o sistema sofre modificações durante a execução.

A dinamicidade é de fato um grande desafio no projeto dos sistemas de quóruns e, mais especificamente, no projeto de protocolos para reconfiguração do conjunto de servidores que mantêm a memória compartilhada. Devido a entradas, saídas e falhas de servidores e de clientes, é preciso manter a consistência dos dados e garantir a sua disponibilidade. Esse processo de reconfiguração torna-se ainda mais complexo quando considera-se a possibilidade de componentes maliciosos estarem presentes na computação [Lamport et al. 1982].

Alguns trabalhos foram propostos com o intuito de prover uma memória compartilhada em ambientes dinâmicos e assim implementar um sistema de quóruns dinâmicos capaz de tolerar tanto o *crash* de servidores [Lynch and Shvartsman 2002, Aguilera et al. 2011] quanto o comportamento malicioso dos mesmos [Martin and Alvisi 2004, Rodrigues and Liskov 2004, Alchieri et al. 2012]. Porém, uma característica comum a todas estas propostas é que nenhuma destas fornece um protocolo capaz de tolerar a presença de clientes maliciosos no sistema. De fato, um dos grandes desafios em sistemas de quóruns é desenvolver protocolos eficientes para tolerar comportamento malicioso de clientes, pois os mesmos podem executar uma série de ações maliciosas com o objetivo de ferir as propriedades do sistema, como por exemplo realizar uma escrita incompleta, atualizando apenas alguns servidores. Esta característica é importante na medida em que os sistemas de quóruns foram desenvolvidos para operar em ambientes de larga escala, como a Internet, onde sua natureza de sistema aberto aumenta significativamente a possibilidade de clientes maliciosos estarem presentes no sistema. Se considerarmos um ambiente dinâmico e todas suas incertezas, esta possibilidade é ainda maior.

Neste artigo, usamos como base os protocolos de reconfiguração do QUINCUNX [Alchieri et al. 2012] e propomos protocolos para um sistema de quóruns dinâmico que tolera tanto servidores quanto clientes maliciosos. Para isso, um esquema de criptografia de limiar é utilizado, de forma que os servidores corretos conseguem tanto controlar as ações de clientes maliciosos quanto mascarar o comportamento malicioso de outros servidores. Os protocolos do QUINCUNX foram escolhidos por permitirem a reconfiguração do sistema sem a necessidade de premissas temporais, o que os tornam adequados para sistemas assíncronos[1]. Os procedimentos de reconfiguração envolvem a evolução do sistema de uma configuração (visão) antiga para uma nova configuração (visão) atualizada, onde servidores entram e/ou saem do sistema. Além disso, os protocolos de reconfiguração do QUINCUNX são independentes dos protocolos de leitura e escrita (R/W) usados para acessar o sistema, o que facilita a adequação de outros protocolos de R/W para ambientes dinâmicos, além de aumentar o desempenho destas operações quando estão concorrendo com reconfigurações.

A solução proposta neste trabalho, chamada QUINCUNX-BC (QUINCUNX *for byzantine clients*) preserva estas características, i.e., os protocolos do QUINCUNX-BC não necessitam de nenhuma premissa temporal e mantém o desacoplamento entre os protocolos de reconfiguração e os protocolos de R/W. Os protocolos de R/W do QUINCUNX-BC são uma adaptação para ambientes dinâmicos dos protocolos do PBFT-BC [Alchieri et al. 2009], que utilizam cripto-

[1]Como sabemos, não é necessário premissas de sincronização (tempo) para a manutenção da consistência de uma memória compartilhada em um conjunto estático de servidores [Malkhi and Reiter 1998].

grafia de limiar para tolerar processos maliciosos. Neste sentido, um exigência fundamental no QUINCUNX-BC é o emprego de um protocolo de redistribuição de chaves parciais (usadas pelo mecanismo de criptografia de limiar), onde as chaves parciais relacionadas a uma visão mais antiga são redistribuídas entre os servidores de uma visão mais atual. Neste sentido, este trabalho apresenta pelo menos duas contribuições principais:

- Novos protocolos de R/W para sistema de quóruns Bizantinos dinâmicos, que utilizam criptografia de limiar para tolerar tanto clientes quanto servidores maliciosos;
- Um protocolo para redistribuição de chaves parciais entre as visões instaladas no sistema, que não requer primitivas de sincronização.

No QUINCUNX-BC, estes protocolos são utilizados em conjunto com os protocolos de reconfiguração do QUINCUNX resultando em um sistema de quóruns Bizantino dinâmico que tolera tanto clientes quanto servidores maliciosos. Esta característica torna o QUINCUNX-BC interessante para sistemas dinâmicos que, devido às suas características, são mais susceptíveis à presença de componentes maliciosos.

O resto deste artigo é organizado da seguinte forma. A Seção 2 apresenta algumas definições preliminares. Os protocolos para tolerar clientes maliciosos em sistemas de quóruns Bizantinos Dinâmicos são apresentados na Seção 3. A Seção 4 apresenta algumas discussões sobre os protocolos propostos e as conclusões do trabalho são apresentadas na Seção 5.

2. Definições Preliminares

Esta seção apresenta os principais conceitos e forma a base teórica para o QUINCUNX-BC.

2.1. Modelo de Sistema

Consideramos um sistema distribuído assíncrono completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois subconjuntos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. Cada processo do sistema (cliente ou servidor) possui um identificador único e está sujeito a faltas Bizantinas [Lamport et al. 1982]. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. A chegada dos processos no sistema segue o modelo de chegadas infinita com concorrência desconhecida mas finita [Aguilera 2004]. As comunicações entre processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados.

Nossos protocolos utilizam criptografia de limiar para controlar as ações dos clientes e garantir a integridade dos dados armazenados. Deste modo, consideramos que na inicialização do sistema cada servidor presente na visão inicial recebe a sua chave parcial, usada na elaboração de assinaturas parciais. Servidores corretos nunca revelam estas chaves, que são geradas e distribuídas por um administrador correto que somente é necessário na inicialização do sistema. Após isso, para instalar visões atualizadas no sistema, um protocolo para redistribuição de chaves parciais é utilizado. A chave pública do serviço, usada para verificar as assinaturas geradas por este mecanismo, é armazenada pelos servidores e fica disponível para qualquer processo do sistema. Vale ressaltar que esta chave nunca é modificada, i.e., mesmo que ocorram redistribuições das chaves parciais, a chave pública do serviço não é alterada.

Cada servidor possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Cada chave privada é conhecida apenas pelo seu próprio dono, por outro lado todos os processos conhecem todas as chaves públicas. Denotamos uma mensagem m assinada pelo processo i como m_i e consideramos que apenas mensagens corretamente assinadas são processadas pelos processos corretos.

Consideramos a existência de uma função criptográfica de resumo (*hash*) resistente a colisões h, de tal modo que qualquer processo é capaz de calcular o resumo $h(v)$ do valor v, sendo computacionalmente inviável obter dois valores distintos v e v' tal que $h(v) = h(v')$. Por fim, para evitar ataques de *replay*, *nonces* são adicionados em certas mensagens. Consideramos que os clientes corretos não escolhem *nonces* repetidos, i.e., já utilizados.

2.2. Criptografia de Limiar

A principal ferramenta utilizada no QUINCUNX-BC é um esquema de assinatura de limiar (*threshold signature scheme* - TSS) [Desmedt and Frankel 1990, Shoup 2000] através do qual é possível controlar as ações dos clientes e garantir a integridade dos dados armazenados pelos servidores. Em um esquema (n, k)-TSS, um distribuidor confiável inicialmente gera n chaves parciais $(SK_1, ..., SK_n)$, n chaves de verificação $(VK_1, ..., VK_n)$, a chave de verificação de grupo VK e a chave pública PK usada para validar assinaturas. O distribuidor envia estas chaves para n partes diferentes, chamados portadores, de modo que cada portador i recebe sua chave parcial SK_i e sua chave de verificação VK_i. A chave pública e as chaves de verificação são disponibilizadas para qualquer parte que compõe o sistema.

Após esta configuração inicial, o sistema está apto à gerar assinaturas. Na obtenção de uma assinatura do serviço A para o dado $data$, primeiramente cada portador i gera a sua assinatura parcial a_i para $data$. Posteriormente, um combinador obtém pelo menos k assinaturas parciais válidas $(a_1, ..., a_k)$ e constrói a assinatura A através da combinação destas k assinaturas parciais. Uma propriedade fundamental deste esquema é a impossibilidade de gerar assinaturas válidas com menos de k assinaturas parciais. Este esquema é baseado nas seguintes primitivas:

- $Th\_Sign(SK_i, VK_i, VK, data)$: função usada pelo portador i para gerar sua assinatura parcial a_i sobre $data$ e as provas v_i de validade desta assinatura, i.e., $\langle a_i, v_i \rangle$.
- $Th\_VerifyS(data, \langle a_i, v_i \rangle, PK, VK, VK_i)$: função usada para verificar se a assinatura parcial a_i, apresentada pelo portador i, é válida.
- $Th\_CombineS(a_1, ..., a_k, PK)$: função usada para combinar k assinaturas parciais válidas e obter a assinatura A.
- $verify(data, A, PK)$: função usada para verificar a validade da assinatura A sobre o dado $data$ (verificação normal de assinatura).

O QUINCUNX-BC utiliza o protocolo proposto em [Shoup 2000], onde é provado que tal esquema é seguro no modelo dos oráculos aleatórios [Bellare and Rogaway 1993], não sendo possível forjar assinaturas. Este protocolo representa um esquema de assinaturas de limiar baseado no algoritmo RSA [Rivest et al. 1978], i.e., a combinação das assinaturas parciais gera uma assinatura RSA. Neste modelo a geração e verificação de assinaturas parciais é completamente não interativa, não sendo necessárias trocas de mensagens para executar estas operações.

Para redistribuição de chaves parciais, seguimos as definições apresentadas em [Wong et al. 2002] onde a transição de um esquema (n, k)-TSS para um esquema (n', k')-TSS funciona da seguinte forma: primeiro, cada um dos n portadores de (n, k)-TSS gera n' *shares* de sua chave parcial (um *share* para cada portador do novo esquema) e envia estes *shares* para os n' portadores do novo esquema (n', k')-TSS, os quais obtém pelo menos k *shares* válidos e combinam suas novas chaves parciais. Um requisito fundamental para este protocolo funcionar é que, para as novas chaves parciais serem compatíveis, é necessário que os k *shares* combinados por cada portador do esquema (n', k')-TSS sejam gerados pelo mesmo conjunto de k portadores de (n, k)-TSS [Wong et al. 2002]. Note que as novas chaves parciais derivam das chaves parciais anteriores e são compatíveis com a chave pública PK do serviço. Para redistribuição de chaves parciais entre estes dois esquemas, as seguintes funções são necessárias:

- $Th\_share(SK_i, j, n', k')$: função usada pelo portador i para gerar o *share* $\widehat{k}_{ij}$ de sua chave parcial SK_i. Este *share* é usado por j na obtenção de sua nova chave parcial.
- $Th\_generateV(i, n', k')$: função usada pelo portador i para gerar o conjunto de verificadores e_i, os quais atestam que os *shares* de sua chave parcial são válidos.
- $Th\_verifySK(\widehat{k}_{ji}, e_j)$: função usada pelo portador i para verificar a validade do *share* $\widehat{k}_{ji}$ gerado por j. Nesta função, e_j representa os verificadores gerados por j.
- $Th\_combineSK(\widehat{k}_{1i}, ..., \widehat{k}_{ki}, red\_set)$: função usada pelo portador i para obter sua nova chave parcial SK_i (e verificador VK_i) a partir da combinação de k *shares* válidos obtidos das chaves parciais da configuração antiga. Os identificadores dos portadores, a partir dos quais os *shares* devem ser usados nesta combinação, estão em red_set.

2.3. Sistemas de Quóruns Bizantinos

Sistemas de quóruns Bizantinos [Malkhi and Reiter 1998], doravante denominados apenas como sistemas de quóruns, implementam sistemas replicados de armazenamento de dados distribuídos com garantias de consistência e disponibilidade mesmo com a ocorrência de faltas Bizantinas em algumas de suas réplicas. Algoritmos para sistemas de quóruns são reconhecidos por seus bons desempenho e escalabilidade, já que os clientes desse sistema faze m acesso, de fato, a somente um conjunto particular de servidores ao invés de todos os servidores.

Servidores em um sistema de quóruns organizam-se em subconjuntos denominados quóruns. Cada dois quóruns de um sistema mantêm um número suficiente de servidores corretos em comum (garantia de consistência), sendo que existe pelo menos um quórum no sistema formado somente por servidores corretos (garantia de disponibilidade) [Malkhi and Reiter 1998]. Os clientes realizam operações de leitura e escrita em registradores replicados por estes quóruns. Cada registrador detém um par $\langle v, t \rangle$ com um valor v do dado armazenado e uma estampilha de tempo (*timestamp*) t associada. Este *timestamp* é definido pelo cliente quando da sua escrita, sendo que cada cliente c utiliza conjuntos disjuntos de *timestamps*.

Dentre as várias propostas de protocolos de leitura e escrita (R/W) para sistemas de quóruns [Malkhi and Reiter 1998, Liskov and Rodrigues 2006, Alchieri et al. 2009], este trabalho utiliza como base o PBFT-BC (*Proactive Byzantine fault-tolerance for Byzantine clients*) [Alchieri et al. 2009], adicionando mecanismos para suportar a reconfiguração no conjunto de servidores que mantêm o sistema e, com isso, tolerar a presença de clientes maliciosos em ambientes dinâmicos. Os protocolos de R/W do PBFT-BC foram escolhidos por apresentarem resiliência ótima ($n \geq 3f + 1$ servidores, utilizando quóruns de $\lceil \frac{n+f+1}{2} \rceil$ servidores para tolerar até f faltas maliciosas em servidores), tolerarem clientes maliciosos e implementarem um registrador com semântica atômica [Lamport 1978]. Além disso, o PBFT-BC utiliza criptografia de limiar para controlar as ações dos clientes, onde através de um esquema $(n, \lceil \frac{n+f+1}{2} \rceil)$-TSS é necessário que pelo menos um quórum de servidores aprovem determinada ação de um cliente. Como veremos adiante, este mecanismo fornece a flexibilidade necessária para adaptação do protocolo às mudanças que ocorrem na composição do grupo de servidores em um ambiente dinâmico. Por exemplo, os clientes apenas precisam conhecer uma única chave pública do serviço (Seção 2.2), ao invés de todas as chaves públicas dos servidores.

Implementar um sistema de quóruns replicado em apenas $3f + 1$ servidores requer que os dados armazenados sejam auto-verificáveis, pois a intersecção entre os quóruns poderá conter apenas um servidor correto. Assim, os clientes obtêm corretamente os dados armazenados, a partir deste servidor, apenas se tais dados forem auto-verificáveis, pois deste modo é possível verificar a integridade dos mesmos. Neste sentido, a grande diferença do PBFT-BC em relação a outros sistemas que armazenam dados auto-verificáveis está justamente na forma de garan-

tir a integridade destes dados, onde o PBFT-BC utiliza um assinatura gerada pelos servidores através do emprego de mecanismos de criptografia de limiar e outros protocolos, como o *f-dissemination quorum system* [Malkhi and Reiter 1998], utilizam assinaturas de clientes que, portanto, não podem ser maliciosos[2]. O BFT-BC [Liskov and Rodrigues 2006] utiliza assinaturas assimétricas de servidores para garantir a integridade dos dados armazenados e portanto, apesar de tolerar clientes maliciosos, não apresenta a flexibilidade necessária para utilização em sistemas distribuídos dinâmicos. A Seção 3.1 apresenta os algoritmos de leitura e escrita do PBFT-BC adaptados para os ambientes dinâmicos.

2.4. Dinamicidade do Sistema

Esta seção descreve brevemente os principais aspectos do QUINCUNX [Alchieri et al. 2012], um conjunto de protocolos para reconfiguração de sistemas de quóruns Bizantinos, que são utilizados como base para reconfiguração de nosso sistema. A grande vantagem do QUINCUNX é que seus protocolos de reconfiguração são completamente desacoplados dos protocolos de leitura e escrita, utilizados por clientes para acessar o registrador implementado pelo sistema. Desta forma, qualquer protocolo tradicional de sistemas de quóruns Bizantinos para ambientes estáticos é facilmente adaptado para ambientes dinâmicos através da utilização dos protocolos de reconfiguração do QUINCUNX.

Os protocolos de reconfiguração do QUINCUNX são responsáveis pela atualização da *visão* (*membership*) do sistema. No QUINCUNX, cada visão v é uma tupla $\langle ov, entries, \mathtt{P}\rangle$, onde: ov é a visão cuja atualização gerou v; $entries$ é um conjunto de *updates* e define o *membership* de v (o *update* $\langle +, i\rangle$ define que $i \in v$ enquanto o *update* $\langle -, i\rangle$ define que $i \notin v$); e P é um certificado contendo as provas que garantem tanto a integridade de v como também que v foi instalada no sistema. Sendo $|v|$ o número de servidores presentes em v, no máximo $v.f = \lfloor \frac{|v|-1}{3} \rfloor$ servidores de v podem falhar e o sistema utiliza quóruns de tamanho $v.q = \lceil \frac{|v|+v.f+1}{2} \rceil$. Duas visões v_1 e v_2 são comparadas através de seus campos $entries$. A notação $v_1 \subset v_2$ e $v_1 = v_2$ é utilizada como abreviação de $v_1.entries \subset v_2.entries$ e $v_1.entries = v_2.entries$, respectivamente. Caso $v_1 \subset v_2$, então v_2 é *mais atual* do que v_1 [Alchieri et al. 2012]. Uma visão é válida caso P possua as provas de que tal visão foi gerada e instalada no sistema (função $isValidV$ utilizada nos algoritmos da Seção 3).

Na inicialização do sistema, cada servidor $i \in v_0$ recebe a visão inicial $v_0 = \langle \perp, u_0, \emptyset\rangle$, onde u_0 representa o *membership* de v_0. Após isso, o sistema executa reconfigurações periódicas gerando e instalando novas visões, de forma que é possível determinar uma sequência de visões instaladas a partir de v_0 como $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow ....$ Para não acessar dados antigos, os protocolos de leitura e escrita executados pelos clientes sempre devem acessar a visão mais atual instalada no sistema. Um grande atrativo do QUINCUNX é que seus protocolos permitem a reconfiguração do sistema sem a necessidade de premissas temporais a respeito do comportamento do mesmo [Alchieri et al. 2012].

3. QUINCUNX-BC: Tolerando Clientes Maliciosos

Esta seção discute como integrar os protocolos do PBFT-BC aos do QUINCUNX, fazendo com que o sistema passe a tolerar também clientes maliciosos. É importante que o sistema apresente esta característica, pois clientes maliciosos podem tentar corromper o sistema através das seguintes ações [Liskov and Rodrigues 2006]: *(i)* escrever valores diferentes associados com o mesmo *timestamp*; *(ii)* executar o protocolo parcialmente, atualizando os valores armazenados

[2]O protocolo *f-masking quorum system* [Malkhi and Reiter 1998] também tolera algumas ações de clientes maliciosos. Porém, por não armazenar dados auto-verificáveis, requer replicação em $4f + 1$ servidores.

por apenas algumas réplicas; *(iii)* escolher um *timestamp* muito grande, causando a exaustão do espaço de *timestamps*; e *(iv)* preparar um grande número de escritas e trabalhar junto com um aliado que executa estas operações mesmo após o cliente faltoso ser removido do sistema.

O desafio fundamental no desenvolvimento deste sistema, chamado QUINCUNX-BC (QUINCUNX *for byzantine clients*), está no projeto de um protocolo para redistribuição de chaves parciais que não deve fazer uso de nenhuma primitiva de sincronização para utilização em sistemas completamente assíncronos, além da adaptação dos algoritmos de leitura e escrita do PBFT-BC para ambientes dinâmicos. Além disso, como novos protocolos serão incorporados, novas informações devem fazer parte das visões do sistema. Desta forma, cada visão v do sistema passa a possuir um campo $v.rd\_info$ que contém informações sobre o procedimento de redistribuição de chaves, como veremos adiante.

Para adaptar os protocolos de leitura e escrita do PBFT-BC aos protocolos de reconfiguração do QUINCUNX, é necessário que cada cliente verifique a visão corrente do sistema na execução de cada fase do protocolo. Neste sentido, caso uma visão desatualizada esteja sendo utilizada, é necessário reiniciar a fase em execução. A Seção 3.1 apresenta estes protocolos.

Seguindo os limiares adotados no PBFT-BC [Alchieri et al. 2009], para cada visão v do QUINCUNX-BC é definido um esquema de criptografia de limiar $(|v.members|, v.q)$-TSS, utilizado por servidores e clientes que executam operações na visão v. Como já comentado, na inicialização do sistema dinâmico cada servidor correto da visão inicial recebe sua chave parcial, que é secreta e nunca será revelada. Estas chaves são geradas e distribuídas por um administrador correto necessário para a inicialização do sistema. Após isso, um protocolo deve ser empregado para redistribuição de chaves parciais entre as visões instaladas no sistema. Este protocolo deve ser executado de modo que os servidores de uma visão anterior na sequência de visões instaladas redistribuam suas chaves parciais para os servidores da visão seguinte e assim por diante. Por exemplo, dada uma sequência de visões instaladas $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ...$, o esquema $(|v_0.members|, v_0.q)$-TSS deve sofrer uma redistribuição para o esquema $(|v_1.members|, v_1.q)$-TSS, que deve ser redistribuído para o esquema $(|v_2.members|, v_2.q)$-TSS, e assim por diante. A Seção 3.2 apresenta este protocolo, que deve ser executado sempre que uma visão mais atual é instalada no sistema.

3.1. Leitura e Escrita

Esta seção apresenta os algoritmos de leitura e escrita (R/W) do QUINCUNX-BC. Como a redistribuição de chaves parciais (Seção 3.2) e as reconfigurações [Alchieri et al. 2012] são desacopladas dos protocolos de R/W, estes protocolos são similares as suas versões estáticas [Alchieri et al. 2009]. A principal diferença é que em nosso sistema cada processo (cliente ou servidor) utiliza uma variável cv para armazenar a visão mais atual conhecida.

Para correto funcionamento, os clientes devem escolher *timestamps* de subconjuntos diferentes. Assim, cada cliente concatena seu identificador único com um número de seqüência, i.e., $ts = \langle seq, id \rangle$. *Timestamps* são comparados verificando-se primeiramente o número de seqüência (seq) e posteriormente o identificador do cliente (id), caso os números de sequência sejam idênticos. *Timestamps* são incrementados através da função $succ(ts, c) = \langle ts.seq + 1, c \rangle$.

Para progredir nas operações, um cliente precisa provar que está agindo corretamente. Isto é realizado através do uso de **certificados**, que contêm os dados indicando a validade da ação que o cliente está tentando executar e uma assinatura de serviço que garante a integridade destes dados. Esta assinatura é gerada a partir de assinaturas parciais de $cv.q$ (um quórum) servidores de cv que aprovaram a ação do cliente. Dois tipos de certificados são utilizados:

Certificado de Preparação: o cliente utiliza este certificado para provar que preparou uma escrita (um quórum de servidores de cv aprovam a escrita) e os servidores o usam para provar a integridade dos valores armazenados. Um certificado de preparação cp possui três campos: $cp.ts$ – *timestamp* da escrita proposta; $cp.hash$ – *hash* do valor v proposto para ser escrito; $cp.A$ – assinatura do serviço, provando que pelo menos um quórum de servidores de cv aprovam a escrita de v com o *timestamp* $cp.ts$. Para um certificado de preparação ser válido (função $isValidPC$ – algoritmos 1 e 2) é necessário que a assinatura $cp.A$, sobre a tupla $\langle cp.ts, cp.hash\rangle$, seja válida, o que é determinado pela operação $verify(\langle cp.ts, cp.hash\rangle, cp.A, PK)$.

Certificado de Escrita: o cliente utiliza este certificado para provar que terminou uma escrita. Um certificado de escrita ce possui dois campos: $ce.ts$ – *timestamp* da escrita realizada; $ce.A$ – assinatura do serviço, provando que o cliente realizou a escrita relacionada com o *timestamp* $ce.ts$ em pelo menos um quórum de servidores de cv. Para um certificado de escrita ser válido (função $isValidWC$ – algoritmos 1 e 2) é necessário que a assinatura $ce.A$, sobre $ce.ts$, seja válida, o que é determinado pela operação $verify(ce.ts, ce.A, PK)$.

Este modelo suporta o armazenamento de múltiplos objetos nos servidores, desde que os mesmos tenham identificadores diferentes. Para simplificar a apresentação do protocolo, consideraremos que os servidores armazenam um único objeto. Deste modo, no QUINCUNX-BC cada servidor i utiliza (armazena) as seguintes variáveis: **(1)** $data$ – valor do objeto; **(2)** P_{cert} – certificado de preparação válido para $data$; **(3)** P_{list} – conjunto de tuplas $\langle c, ts, hash\rangle$ contendo o identificador c do cliente, o *timestamp* ts e o *hash* do valor do objeto das escritas propostas; **(4)** max_{ts} – *timestamp* relacionado com a última escrita completa conhecida por i; **(5)** SK_i – chave parcial de i para cv, usada pelo mecanismo de assinatura de limiar; **(6)** VK e Vk_j para cada $j \in cv$ – chaves de verificação para cv, usadas para gerar provas de validade de assinaturas parciais; e **(7)** PK – chave pública do serviço usada para validar certificados. Finalmente, cada cliente c utiliza as seguintes variáveis: **(1)** W_{cert} – certificado de escrita referente à última escrita de c; **(2)** PK – chave pública do serviço usada para validar certificados; e **(3)** VK e Vk_j para cada $j \in cv$ – chaves de verificação para cv, usadas para validar assinaturas parciais.

Para acessar o sistema, um cliente deve obter a visão atual do mesmo, o que pode ser realizado de duas formas [Alchieri et al. 2012]: (1) recuperar as visões a partir de um local pré-definido; ou (2) realizar uma inundação no sistema requisitando a visão atual. Os algoritmos 1 e 2 apresentam os protocolos de R/W executados por clientes e servidores, respectivamente.

Escrita. Após obter a visão atual do sistema e atualizar cv, um cliente c que deseja escrever o valor $value$ no registrador primeiramente envia uma mensagem READ_TS para todos os servidores de cv e aguarda por um quórum ($cv.q$) de respostas válidas. Cada servidor i envia uma resposta $\langle$READ_TS_REP$, p, nonceResp, cv_i\rangle$ para c (linha 2, alg. 2), onde p é o P_{cert} armazenado por i. Esta resposta é válida caso (linha 4, alg. 1): (1) p é válido; (2) sua autenticidade é comprovada, i.e., os *nonces* da requisição e da resposta são iguais; e (3) a visão atual do servidor (cv_i) é válida e igual à visão cv do cliente. Dentre os certificados de preparação recebidos, c seleciona o certificado que contém o maior *timestamp*, chamado P_{max} (linha 10, alg. 1) e define o *timestamp* ts de sua escrita (linha 11, alg. 1).

Na segunda fase, c envia uma mensagem PREPARE para os servidores de cv, preparando sua escrita com valor $value$ e *timestamp* ts. Esta mensagem contém as seguintes informações: (1) o *timestamp* ts definido para a escrita e P_{max} para provar que ts é um valor válido e c não está executando um ataque de exaustão de *timestamp* [Bazzi and Ding 2004, Liskov and Rodrigues 2006]); (2) o *hash* de $value$ para obter um certificado de preparação; (3) o certificado W_{cert} da última escrita de c ou $null$ caso seja a primeira escrita de c; e (4) a visão

atual cv de c. Então, c aguarda por um quórum de respostas válidas. Cada servidor i envia uma resposta $\langle$PREPARE_REP, $\langle ts_i, hash_i\rangle, \langle a_i, v_i\rangle, cv_i\rangle$ para c (linha 27, alg. 2) com sua assinatura parcial a_i, utilizada por c na construção do certificado de preparação. Assim, esta resposta é válida caso (linha 15, alg. 1): (1) ts_i e $hash_i$ são os mesmos valores envidados na mensagem PREPARE; (2) a assinatura parcial de i para $\langle ts_i, hash_i\rangle$ é válida; e (3) a visão atual do servidor (cv_i) é válida e igual a visão atual do cliente. O cliente c combina as assinaturas parciais válidas recebidas e obtém uma assinatura de serviço para o par $\langle ts, h(value)\rangle$ (linha 21, alg. 1), construindo o certificado de preparação P_{new} para ts e $h(value)$ (linha 22, alg. 1).

A segunda fase é a mais importante, pois é na mesma que os servidores verificam se: (1) o *timestamp* sendo proposto é correto (linha 15, alg. 2); (2) o cliente esta preparando somente uma escrita (linha 22, alg. 2); (3) o valor proposto é o mesmo de algum (possível) valor anteriormente proposto para o mesmo *timestamp* (line 22, alg. 2); e (4) o cliente completou sua escrita anterior (linhas 16-21, alg. 2). Desta forma, um cliente malicioso m_c não consegue preparar múltiplas escritas que poderão ser executadas por outro processo mesmo após m_c ser removido do sistema, limitando o poder dos clientes maliciosos [Liskov and Rodrigues 2006].

Algoritmo 1 Algoritmo executado pelo cliente *c*

```
procedure write(value)
    {Step 1}
1:  send⟨READ_TS, nonceReq, cv⟩ to each i ∈ cv
2:  repeat
3:      wait reply ⟨READ_TS_REP, p, nonceResp, cv_i⟩ from i ∈ cv
4:      if isValidPC(p) ∧ nonceReq = nonceResp ∧ isValidV(cv_i)∧
                                                  equals(cv, cv_i) then
5:          RTS_i ← p
6:      else if isValidV(cv_i) ∧ updateClientView(cv_i) then
7:          restart step 1
8:      end if
9:  until |RTS| ≥ cv.q
10: P_max ← max(RTS)
11: ts ← succ(P_max.ts, c)
    {Step 2}
12: send⟨PREPARE, P_max, ts, h(value), W_cert, cv⟩ to each i ∈ cv
13: repeat
14:     wait reply ⟨PREPARE_REP, ⟨ts_i, hash_i⟩, ⟨a_i, v_i⟩, cv_i⟩ from i ∈ cv
15:     if (hash_i = h(value)) ∧ isValidV(cv_i) ∧ equals(cv, cv_i)∧
        (ts_i = ts) ∧ Th_verifyS(⟨ts_i, hash_i⟩, ⟨a_i, v_i⟩, PK, VK, VK_i) then
16:         PREPARED_i ← a_i
17:     else if isValidV(cv_i) ∧ updateClientView(cv_i) then
18:         restart step 2
19:     end if
20: until |PREPARED| ≥ cv.q
21: A ← Th_combineS(PREPARED, PK)
22: P_new ← ⟨ts, h(value), A⟩
    {Step 3}
23: send⟨WRITE, value, P_new, cv⟩ to each i ∈ cv
24: repeat
25:     wait reply ⟨WRITE_REP, ts_i, ⟨a_i, v_i⟩, cv_i⟩ from i ∈ cv
26:     if (ts_i = ts) ∧ isValidV(cv_i) ∧ equals(cv, cv_i)∧
                          Th_verifyS(ts_i, ⟨a_i, v_i⟩, PK, VK, VK_i) then
27:         WROTE_i ← a_i
28:     else if isValidV(cv_i) ∧ updateClientView(cv_i) then
29:         restart step 3
30:     end if
31: until |WROTE| ≥ cv.q
32: A ← Th_combineS(WROTE, PK)
33: W_cert ← ⟨ts, A⟩

procedure read()
    {Step 1}
34: send⟨READ, nonceReq, cv⟩ to each i ∈ cv
35: repeat
36:     wait reply ⟨READ_REP, value, p, nonceResp, cv_i⟩ from i ∈ cv
37:     if isValidPC(p) ∧ (p.hash = h(value)) ∧ isValidV(cv_i)∧
                          nonceReq = nonceResp ∧ equals(cv, cv_i) then
38:         READ_i ← ⟨p, value⟩
39:     else if isValidV(cv_i) ∧ updateClientView(cv_i) then
40:         restart step 1
41:     end if
42: until |READ| ≥ cv.q
43: R_max ← maxTS(READ)
    {Step 2 (write-back phase)}
44: if ∃⟨p, ∗⟩ ∈ READ: p.ts ≠ R_max.p.ts then
45:     send⟨WRITE, R_max.value, R_max.p, cv⟩ to each i ∈ cv
46:     repeat
47:         wait reply ⟨WRITE_REP, ts_i, ⟨a_i, v_i⟩, cv_i⟩ from i ∈ cv
48:         if (ts_i = R_max.p.ts) ∧ isValidV(cv_i) ∧ equals(cv, cv_i) then
49:             WROTE_i ← a_i
50:         else if isValidV(cv_i) ∧ updateClientView(cv_i) then
51:             restart step 2
52:         end if
53:     until |WROTE| ≥ cv.q
54: end if
55: return R_max.value
procedure updateClientView(cv_i)
56: if ((cv ⊂ cv_i) ∨ (cv = cv_i ∧ cv.rd_info ≻ cv_i.rd_info)) then
57:     cv ← cv_i
58:     for all j ∈ cv.members do
59:         update VK_j by fetching it from j for cv.rd_info
60:     end for
61:     return true
62: else
63:     return false
64: end if
procedure equals(v_1, v_2)
65: if v_1 = v_2 ∧ v_1.rd_info = v_2.rd_info then
66:     return true
67: else
68:     return false
69: end if
```

Na última fase, c envia uma mensagem WRITE para os servidores em cv, informando o valor a ser escrito juntamente com o certificado de preparação e aguarda por um quórum de respostas válidas. Cada servidor i envia uma resposta $\langle$WRITE_REP, $ts_i, \langle a_i, v_i\rangle, cv_i\rangle$ para c (linha 10, alg. 2) com sua assinatura parcial a_i que será usada por c na construção do certificado de escrita. Assim, esta resposta é válida caso (linha 26, alg. 1): (1) ts_i é igual ao *timestamp* de escrita; (2) a assinatura parcial de i para ts_i é válida; e (3) a visão do servidor cv_i é válida

e igual a visão atual do cliente. Neste caso, c combina as assinaturas parciais válidas recebidas e obtém a assinatura de serviço para ts (linha 32, alg. 1), construindo o certificado de escrita W_{cert} para sua última escrita (linha 33, alg. 1), o qual será usado em sua próxima escrita.

Leitura. Para ler o valor armazenado no sistema, um cliente c envia uma mensagem READ para todos os servidores de cv e aguarda por um quórum ($cv.q$) de respostas válidas. Cada servidor i envia uma resposta $\langle \text{READ\_REP}, data, p, nonceRep, cv_i \rangle$ para c (linha 13, alg. 2), onde p é um certificado de preparação válido para o valor $data$ armazenado em i. Esta resposta é válida caso (linha 37, alg. 1): (1) p é válido para $data$; (2) a sua autenticidade é comprovada, i.e., os *nonces* da requisição e da resposta são iguais; e (3) a visão atual do servidor cv_i é válida e igual a visão atual do cliente. Dentre todas as respostas recebidas, c seleciona a resposta que possui o certificado de preparação com o maior *timestamp* (linha 43, alg. 1) e retornam o valor $value$ associado com esta resposta (linha 55, alg. 1). Caso nem todos os *timestamps* recebidos sejam iguais, c executa uma fase de *write back* no sistema para o maior *timestamp* (linhas 44-54, alg. 1). Esta fase é semelhante à fase 3 do protocolo de escrita.

Algoritmo 2 Algoritmo executado pelo servidor *i* para comunicação com clientes

Upon receipt of $\langle$READ_TS, $nonce$, $cv_c\rangle$ $from\ client\ c$
1: **checkServerView**(cv_c)
2: $send\langle$READ_TS_REP, P_{cert}, $nonce$, $cv\rangle$ $to\ c$
Upon receipt of $\langle$WRITE, $value$, P_{new}, $cv_c\rangle$ $from\ client\ c$
3: **checkServerView**(cv_c)
4: **if** $isValidPC(P_{new}) \wedge P_{new}.hash = h(value)$ **then**
5: **if** $P_{new}.ts > P_{cert}.ts$ **then**
6: $data \leftarrow value$
7: $P_{cert} \leftarrow P_{new}$
8: **end if**
9: $\langle a_i, v_i\rangle \leftarrow Th\_sign(SK_i, VK_i, VK, P_{new}.ts)$
10: $send\langle$WRITE_REP, $P_{new}.ts$, $\langle a_i, v_i\rangle$, $cv\rangle$ $to\ c$
11: **end if**
Upon receipt of $\langle$READ, $nonce$, $cv_c\rangle$ $from\ client\ c$
12: **checkServerView**(cv_c)
13: $send\langle$READ_REP, $data$, P_{cert}, $nonce$, $cv\rangle$ $to\ c$
Upon receipt of $\langle$PREPARE, P_c, ts, $hash$, W_c, $cv_c\rangle$ $from\ client\ c$
14: **checkServerView**(cv_c)
15: **if** $isValidPC(P_c) \wedge isValidWC(W_c) \wedge ts = succ(P_c.ts, c)$ **then**
16: **if** $W_c \neq \perp$ **then**
17: $max_{ts} \leftarrow max(max_{ts}, W_c.ts)$
18: **for all** $e = \langle *, ts, * \rangle \in P_{list}$: $ts \leqslant max_{ts}$ **do**
19: $P_{list} \leftarrow P_{list} \setminus e$
20: **end for**
21: **end if**
22: **if** $\nexists \langle c, t, h\rangle \in P_{list}$: $t \neq ts \vee h \neq hash$ **then**
23: **if** $\langle c, ts, hash\rangle \notin P_{list} \wedge ts > max_{ts}$ **then**
24: $P_{list} \leftarrow P_{list} \cup \langle c, ts, hash\rangle$
25: **end if**
26: $\langle a_i, v_i\rangle \leftarrow Th\_sign(SK_i, VK_i, VK, \langle ts, hash\rangle)$
27: $send\langle$PREPARE_REPLY, $\langle ts, hash\rangle$, $\langle a_i, v_i\rangle$, $cv\rangle$ $to\ c$
28: **end if**
29: **end if**

Lidando com o dinamismo. Para cada resposta recebida dos servidores, o cliente c verifica se a visão do servidor é mais atual do que a sua visão cv. Nestes casos, c atualiza a sua variável cv (função $updateClientView$ – linhas 56-64, alg. 1 – o operador $\succ$ usado para comparar os campos rd_info de duas visões é explicado na Seção 3.2) e reinicia a fase que está executando, garantindo que não irá acessar dados obsoletos. Além disso, c envia sua visão atual cv em todas as mensagens e cada servidor verifica se está atrasado em alguma reconfiguração (função $checkServerView$ usada no Algortimo 2). Nestes casos, o servidor deve aguardar até que os procedimentos de reconfiguração instalem esta visão atualizada e então responder para c. Usando esta técnica, um cliente apenas recebe respostas de processos corretos com visões iguais a cv ou mais atuais, de forma que respostas com visões antigas podem ser descartadas.

3.2. Redistribuição de Chaves Parciais

Esta seção apresenta o protocolo para redistribuição de chaves parciais, onde os servidores de uma visão v ($(|v.members|, v.q)$-TSS) redistribuem suas chaves parciais para os servidores da visão seguinte w ($(|w.members|, w.q)$-TSS) na sequência de visões instaladas. Este protocolo, que pode ser utilizado por qualquer sistema que emprega criptografia de limiar e suporta mudanças no conjunto de processos que compõem o sistema, garante que os servidores da visão w obtém suas chaves parciais a partir de *shares* gerados pelo mesmo conjunto de servidores presentes na visão v, o que é um requisito para que as novas chaves parciais sejam compatíveis e continuem gerando assinaturas válidas [Wong et al. 2002].

A ideia principal deste protocolo consiste em fazer com que os servidores de v gerem os *shares* de suas chaves parciais e enviem cada *share* para o seu respectivo receptor de w.

Após isso, todos os servidores de w combinam os *shares* recebidos de um mesmo conjunto de processos de v e obtêm as suas novas chaves parciais. Como servidores maliciosos de v podem enviar *shares* inválidos, e além disso em um ambiente assíncrono não é possível garantir que as mensagens sejam entregues na mesma ordem nos servidores de w, estes servidores devem trocar mensagens até convergirem para a mesma informação a respeito da redistribuição RD_INFO, que é composta por dois conjuntos: RD_INFO.$good\_set$ - conjunto de $v.q$ servidores de v a partir dos quais os *shares* devem ser usados na geração das novas chaves parciais; RD_INFO.bad_set - conjunto de servidores maliciosos de v que devem ser desconsiderados.

O Algoritmo 3 apresenta este protocolo e deve ficar sempre ativo para atualizar a RD_INFO de uma visão w assim que novas informações forem recebidas pelos servidores de w, causando a obtenção novas chaves parciais. Qualquer alteração na RD_INFO de um servidor é refletida em todos os outros servidores de w (invariante mantido pelo algoritmo), de forma que estes servidores escolhem os *shares* do mesmo conjunto de processos.

Algoritmo 3 Redistribuindo chaves parciais da visão v para a w (servidor *i*)

variables: {Sets and arrays used in the protocol}
RDK, RDV - set of encrypted shares and set of verifiers generated by i
GOOD, BAD - set of encrypted good shares and set of invalid shares collected by i
SHARES - set of good shares collected by i
RD_INFO - information about the redistribution

procedure $Partial\_Keys\_Redist(v, w)$
1: **if** $i \in v$ **then**
2: **for all** $j \in w$ **do**
3: $\widehat{k}_{ij} \leftarrow Th\_shareK(SK_i, j, |w.members|, w.q)$; $\texttt{RDK}^w \leftarrow \texttt{RDK}^w \cup \langle j, encrypt(\widehat{k}_{ij})\rangle$
4: **end for**
5: $\texttt{RDV}^w \leftarrow Th\_generateV(i, |w.members|, w.q)$;
6: $\forall j \in w, send\langle$REDIST, $\langle \texttt{RDK}^w, \texttt{RDV}^w, v, w\rangle_i\rangle$ to j
7: **end if**

Upon receipt of $\langle$REDIST, $\langle rdk, rdv, v, w\rangle_j\rangle$ from $j \in v \wedge i \in w$
8: **if** $(\texttt{GOOD}_j^w = \perp) \wedge (\texttt{BAD}_j^w = \perp)$ **then**
9: $extractSK(\langle rdk, rdv, v, w\rangle_j)$
10: $updateSK(v, w)$
11: **end if**

procedure $extractSK(\langle rdk, rdv, v, w\rangle_j)$
12: **if** $(\forall z \in w \rightarrow \exists\langle z, encrypted_z\rangle \in rdk)$ **then**
13: $\widehat{k}_{ji} \leftarrow decrypt(encrypted_i)$
14: **if** $Th\_verifySK(\widehat{k}_{ji}, rdv)$ **then**
15: $\texttt{GOOD}_j^w \leftarrow \langle rdk, rdv, v, w\rangle_j$; $\texttt{SHARES}_j^w \leftarrow \widehat{k}_{ji}$
16: **else**
17: $\texttt{BAD}_j^w \leftarrow \langle j, \widehat{k}_{ji}, \langle rdk, rdv, v, w\rangle_j\rangle$
18: **end if**
19: **else**
20: $\texttt{BAD}_j^w \leftarrow \langle j, \perp, \langle rdk, rdv, v, w\rangle_j\rangle$
21: **end if**

procedure $updateSK(v, w)$
22: $\texttt{RD\_INFO}^w.good\_set \leftarrow$ *select* $v.q$ *members with messages in* $\texttt{GOOD}^w$ *with lower ids*
23: **if** $|\texttt{RD\_INFO}^w.good\_set| = v.q$ **then**
24: $\texttt{RD\_INFO}^w.bad\_set \leftarrow$ *select all members with messages in* $\texttt{BAD}^w$
25: **if** $\texttt{RD\_INFO}^w$ updated on line 24 or 26 **then**
26: $\langle SK_i, VK_i\rangle \leftarrow Th\_combineSK(\texttt{SHARES}^w, \texttt{RD\_INFO}^w.good\_set)$
27: $\forall j \in w, send\langle$REDIST-INFO, $\texttt{GOOD}^w, \texttt{BAD}^w, v, w\rangle$ to j
28: $\forall j \in w$, *update* VK_j *by fetching it from* j *for* $\texttt{RD\_INFO}^w$
29: **end if**
30: **end if**

Upon receipt of $\langle$REDIST-INFO, $good, bad, v, w\rangle$ from j
31: **for all** $\langle z, \widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z\rangle \in bad$: $(z \in v) \wedge (\texttt{BAD}_z^w = \perp) \wedge isBad(\widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z)$ **do**
32: $\texttt{BAD}_z^w \leftarrow \langle z, \widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z\rangle$; $\texttt{GOOD}_z^w \leftarrow \perp$; $\texttt{SHARES}_z^w \leftarrow \perp$
33: **end for**
34: **for all** $\langle rdk, rdv, v, w\rangle_z \in good$: $(z \in v) \wedge (\texttt{GOOD}_z^w = \perp) \wedge (\texttt{BAD}_z^w = \perp)$ **do**
35: $extractSK(\langle rdk, rdv, v, w\rangle_z)$
36: **end for**
37: $updateSK(v, w)$

Gerando e redistribuindo chaves parciais. Para iniciar o processo de redistribuição de chaves parciais da visão v para a visão w, todos os servidores de v geram os *shares* de suas chaves parciais (linha 3), juntamente com os verificadores que atestam a validade destes *shares* (linha 5), e enviam estes dados para os servidores de w (linha 6). Para cada portador $j \in w$, um servidor $i \in v$ gera o *share* $\widehat{k}_{ij}$ e o cifra com a chave pública de j, de modo que somente j consegue acessá-lo. Todos os verificadores são públicos. Note que, `REDIST` é a única mensagem assinada (assinatura tradicional) nos protocolos do QUINCUNX-BC, sendo possível provar para terceiros que algum servidor enviou *shares* inválidos, como veremos adiante.

As chaves parciais dos servidores de v, utilizadas no procedimento de redistribuição, devem ter sido obtidas a partir do mesmo conjunto de *shares* ($\texttt{RD\_INFO}^v.good\_set$) [Wong et al. 2002]. Então, caso um servidor de v atualizar sua chave parcial (atualizar $\texttt{RD\_INFO}^v$), o mesmo deve executar novamente este procedimento e os servidores de w devem obter suas chaves parciais a partir de *shares* obtidos de chaves parciais geradas através dos dados mais atuais em $\texttt{RD\_INFO}^v.good\_set$ (por simplicidade, isso não aparece no algoritmo).

Convergindo para o mesmo `RD_INFO`. Quando um servidor $i \in w$ recebe os *shares* de um servidor $j \in v$, i verifica se ainda não recebeu qualquer informação de j (linha 8), utilizando dois conjuntos: `GOOD` - conjunto de *shares* cifrados válidos; e `BAD` - conjunto de *shares* inválidos. Caso estas informações sejam novas, i decifra seu *share* e caso o mesmo seja válido armazena estes dados em `SHARES` e `GOOD` (linha 15). Do contrário, j é malicioso e estes dados são armazenados em `BAD` (linhas 17 e 20).

A função $updateSK$ é a parte principal do protocolo, pois através desta função sempre que alguma informação mais atual é recebida por algum servidor $i \in w$, i atualiza sua chave parcial (caso necessário). Além disso, sempre que alguma nova informação for recebida e a chave parcial atualizada, i envia uma mensagem REDIST-INFO para os servidores de w contendo todas as informações coletadas, de forma que todos estes servidores possam coletar as mesmas informações sobre o processo de redistribuição das chaves parciais. Para isso, sempre que um servidor receber uma mensagem REDIST-INFO (linhas 31-37), o mesmo atualiza seus conjuntos `GOOD`, `BAD` e `SHARES` de acordo com as informações recebidas e executa a função $updateSK$ para verificar se uma chave parcial nova deve ser obtida.

Um servidor $i \in w$ verifica a informação de que um servidor $z \in v$ gerou um *share* inválido (função $isBad(\widehat{k}_{zx}, \langle rdk, rdv, v, w\rangle_z)$) da seguinte forma: (1) i verifica se z gerou *shares* para todos os servidores de w (como na linha 12); (2) i cifra o *share* $\widehat{k}_{zx}$ (*share* enviado para um servidor $x \in w$) com a chave pública de x e verifica se o resultado é igual ao *share* cifrado enviado de z para x ($encrypted_x$ contido em rdk), e caso seja igual, i verifica a validade de $\widehat{k}_{zx}$ através da função $Th\_verifySK(\widehat{k}_{zx}, rdv)$. Uma vez que i descobre que z é malicioso, i adiciona z no conjunto `BAD`, de onde nunca é removido.

Comparando `RD_INFO`. Para que os processos possam obter assinaturas válidas, os mesmos devem usar assinaturas parciais geradas por chaves compatíveis. Para isso, as mensagens trocadas entre os processos (tanto entre servidores para redistribuição de chaves parciais quanto entre clientes e servidores para execução de operações de leitura e escrita) devem conter a informação `RD_INFO` usada na obtenção da chave parcial mais atual do emissor, de modo que os processos apenas considerem as mensagens acompanhadas de informações mais atuais. Como já discutido, este dado faz parte das visões do sistema (campo rd_info) e deve ser obtido um certificado para provar a autenticidade do mesmo, uma vez que são trocados entre os processos. Isto pode ser realizado através da obtenção de uma assinatura de serviço (usando o próprio mecanismo de criptografia de limiar) que deve ser adicionada ao campo `P` das visões (por simplicidade, não

apresentamos este passo no algoritmo).

Dadas duas informações sobre a redistribuição rd_info_1 e rd_info_2, rd_info_2 é mais atual do que rd_info_1 (representado por $rd\_info_1 \succ rd\_info_2$ – Algoritmo 1) se $max(rd\_info_1.good\_set) > max(rd\_info_2.good\_set) \wedge (\forall j \in rd\_info_2.good\_set \rightarrow j \notin rd\_info_1.bad\_set)$: o maior identificador em $rd\_info_2.good\_set$ é menor do que o maior identificador em $rd\_info_1.good\_set$, o que implica que rd_info_2 é formado por servidores com identificadores menores. A segunda parte da condição verifica se o conjunto $rd\_info_2.good\_set$ é formado apenas por servidores corretos.

3.2.1. *Corretude*

O seguinte teorema prova a propriedade fundamental garantida pelo Algoritmo 3.

Teorema 1 *Considere a redistribuição de chaves parciais de uma visão v para uma visão w através do Algoritmo 3. Cada servidor $i \in w$ obtêm uma nova chave parcial e este algoritmo mantém o invariante de que as chaves parciais definidas para os servidores de w sempre acabam por serem compatíveis.*

Prova. Cada servidor correto de v define os *shares* de sua chave parcial e envia estes dados para os servidores de w (linhas 1-7). Como v possui pelo menos $v.q$ servidores corretos e os canais são confiáveis, o predicado da linha 23 será verdadeiro pelo menos uma vez em cada servidor de w, que obterá uma nova chave parcial. Agora temos que provar que os servidores de w sempre acabam definindo o mesmo `RD_INFO`, obtendo suas chaves parciais a partir do mesmo conjunto de servidores de v. Sempre que um servidor $i \in w$ alterar sua chave parcial (linha 26) e seu `RD_INFO` (linhas 22 e 24), i envia todas as informações coletadas para os outros servidores de w (linha 27), que também atualizam seus conjuntos `RD_INFO` (linhas 31-37). Desta forma, todos os servidores de w acabam definindo um mesmo `RD_INFO`. Como os servidores de w obtém suas chaves parciais a partir dos *shares* gerados pelo quórum de servidores corretos de v com menores identificadores (linha 22) e acabam definindo o mesmo `RD_INFO`, todos estes servidores acabam obtendo novas chaves parciais compatíveis. Como sempre que algum servidor de w atualiza sua chave parcial e seu `RD_INFO` o mesmo envia estas informações para os outros servidores de w, este invariante sempre é mantido no sistema. $\square_{Teorema\ 1}$

4. Discussões

Lidando com clientes maliciosos. O QUINCUNX-BC limita o número de escritas incompletas de um cliente usando uma lista P_{list} em cada servidor, a qual armazena as escritas preparadas pelos clientes mas ainda não completadas (Algoritmo 2). Considerando que o sistema sofre uma reconfiguração da visão $view_{old}$ para a visão $view_{new}$. Caso um quórum de servidores ($view_{new}.q$) entraram no sistema em $view_{new}$ com suas P_{list} vazias, um cliente malicioso pode preparar uma escrita w em $view_{new}$ sem que tenha completado alguma escrita w' preparada anteriormente em $view_{old}$. Isso significa que o QUINCUNX-BC garante que cada cliente consegue ter no máximo uma escrita preparada mas incompleta por visão, diferentemente do PBFT-BC [Alchieri et al. 2009] que não sofre reconfigurações e, por isso, cada cliente pode ter no máximo uma escrita incompleta durante todo o ciclo de vida do sistema.

Desempenho. As otimizações propostas para o protocolo de escrita do PBFT-BC [Alchieri et al. 2009] são facilmente incorporadas ao protocolo de escrita do QUINCUNX-BC, possibilitando que este termine em apenas 2 fases de execução. Além disso, como os protocolos de reconfiguração e de redistribuição de chaves são desacoplados dos protocolos de leitura

e escrita, é possível que estas operações sejam executadas concorrentemente, sendo necessário apenas a reinicialização de alguma fase da leitura e/ou da escrita quando uma nova visão for instalada no sistema e/ou novas chaves parciais forem definidas.

5. Conclusões

Este artigo apresentou o QUINCUNX-BC, um sistema de quóruns Bizantinos dinâmico capaz de tolerar o comportamento malicioso tanto de servidores quanto de clientes através do emprego de criptografia de limiar, mecanismo que fornece a flexibilidade suficiente para operação em ambientes dinâmicos. Preservando as características dos protocolos utilizados como base, no QUINCUNX-BC as operações de R/W são completamente desacopladas dos procedimentos de reconfiguração, o que permite concorrência entre R/W e reconfigurações. Esta característica tende à aumentar o desempenho do sistema, principalmente durante reconfigurações.

Referências

Aguilera, M. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Aguilera, M. K., Keidar, I., Malkhi, D., and Shraer, A. (2011). Dynamic atomic storage without consensus. *JACM*, 58:7:1–7:32.

Alchieri, E. A., Bessani, A. N., Silva Fraga, J., and Greve, F. (2012). Memória compartilhada em sistemas bizantinos dinâmicos. In *Anais do XIII Workshop de Teste e Tolerância a Falhas- WTF 2012*.

Alchieri, E. A. P., Bessani, A. N., Pereira, F., and da Silva Fraga, J. (2009). Sistemas de quóruns bizantinos pró-ativos. *Revista Brasileira de Redes de Computadores e Sistemas Distribuídos*, 2:21–34.

Bazzi, R. A. and Ding, Y. (2004). Non-skipping timestamps for Byzantine data storage systems. In *Proc. of 18th Int. Symposium on Distributed Computing, DISC 2004*, volume 3274 of *LNCS*, pages 405–419.

Bellare, M. and Rogaway, P. (1993). Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, pages 62–73.

Desmedt, Y. and Frankel, Y. (1990). Threshold criptosystems. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'89*, pages 307–315. Springer-Verlag.

Gifford, D. (1979). Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162.

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

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

Liskov, B. and Rodrigues, R. S. M. (2006). Tolerating byzantine faulty clients in a quorum system. In *The 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

Lynch, N. and Shvartsman, A. A. (2002). Rambo: A reconfigurable atomic memory service for dynamic networks. In *16th International Symposium on Distributed Computing - DISC*, pages 173–190.

Malkhi, D. and Reiter, M. (1998). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Martin, J.-P. and Alvisi, L. (2004). A framework for dynamic Byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Rodrigues, R. and Liskov, B. (2004). Rosebud: A scalable Byzantine-fault-tolerant storage architecture. MIT-LCS-TR 932, MIT Laboratory for Computer Science.

Shoup, V. (2000). Practical threshold signatures. In *Advances in Cryptology: EUROCRYPT 2000, Lecture Notes in Computer Science*, volume 1807, pages 207–222. Springer-Verlag.

Wong, T. M., Wang, C., and Wing, J. M. (2002). Verifiable secret redistribution for archive systems. In *Proceedings of the 1th International IEEE Security in Storage Workshop*.

