Sistemas de Quóruns Bizantinos Pró-Ativos*

Eduardo Adilio Pelinson Alchieri[1], Alysson Neves Bessani[2], Fernando Carlos Pereira[1], Joni da Silva Fraga[1]

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

Resumo. *Sistemas de quóruns são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. Este trabalho apresenta um sistema de quóruns com semântica atômica das operações, que tolera clientes maliciosos e funciona em um ambiente completamente assíncrono, como a Internet. Além disso, apresenta um protocolo para recuperação pró-ativa dos servidores, de forma que o sistema tolera qualquer número de faltas durante seu ciclo de vida, dado que no máximo f de $3f + 1$ servidores falhem (sejam invadidos) durante um pequeno intervalo de tempo entre as recuperações. Uma análise acerca do desempenho dos protocolos propostos também é realizada.*

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. 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 que compõem o sistema, mas apenas por um quórum dos mesmos, sendo que a consistência do sistema é assegurada pela propriedade de intersecção destes quóruns. Estes sistemas implementam um registrador que permite operações de leitura e escrita com diferentes semânticas de implementação (segura, regular ou atômica) [Lamport 1986].

O conceito de sistemas de quóruns foi inicialmente estudado em um modelo onde os servidores apenas poderiam falhar por parada [Gifford 1979]. Posteriormente, o modelo foi estendido para tolerar comportamento malicioso dos servidores [Malkhi and Reiter 1998a]. No entanto, o grande desafio em sistemas de quóruns é desenvolver protocolos eficientes para tolerar comportamento malicioso por parte dos 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.

Neste sentido, os primeiros protocolos a suportar clientes maliciosos necessitam que o sistema seja replicado em $4f + 1$ servidores para suportar até f falhas, não utilizando assinaturas [Malkhi and Reiter 1998a, Malkhi and Reiter 1998b]. Porém, estes protocolos permitem que um cliente malicioso execute uma série de ações maliciosas, como preparar várias escritas para serem executadas por um aliado após sua exclusão do sistema, interferindo assim no funcionamento dos clientes corretos. Recentemente estes problemas foram tratados no protocolo BFT-BC (*Byzantine fault-tolerance for Byzantine clients*) [Liskov and Rodrigues 2006], que requer apenas $3f + 1$ servidores e permite que um cliente prepare uma escrita somente após terminar a escrita anterior. Para isso, os servidores utilizam assinaturas assimétricas (principal fonte de atrasos em protocolos tolerantes a faltas bizantinas [Castro and Liskov 2002]) no controle das ações dos clientes e na garantia da integridade dos dados armazenados. O sistema de

*Alchieri, Pereira e Fraga são bolsistas CNPq.

quóruns proposto neste trabalho estende este modelo, modificando a forma de garantir a integridade dos dados e de controlar as ações dos clientes, que passam a ser realizadas através de criptografia de limiar [Desmedt and Frankel 1990].

Uma característica fundamental em sistemas tolerantes a faltas bizantinas projetados para operarem por um longo período de tempo é a recuperação pró-ativa de servidores, pois a limitação do número de faltas suportadas durante todo o ciclo de vida do sistema é uma premissa pouco realista quando consideramos um sistema que ficará disponível por um longo tempo. De fato, em sistemas sem recuperação de servidores, um adversário tem o tempo que precisar para corromper mais servidores do que o limite suportado. No entanto, através do processo de recuperação, um número infinito de faltas pode ocorrer no sistema dado que apenas uma parte delas ocorra em cada pequeno intervalo de tempo entre as recuperações. Deste modo, desenvolvemos um protocolo para recuperação pró-ativa dos servidores que implementam nosso sistema de quóruns. Neste sentido, as características advindas com o uso de criptografia de limiar facilitam o desenvolvimento de mecanismos para este fim. Assim, em nosso sistema um adversário tem apenas um pequeno intervalo de tempo para corromper mais servidores do que o suportado, pois após a recuperação um servidor invadido volta a funcionar corretamente.

Este trabalho apresenta três contribuições principais: ***(1)*** um novo sistema de quóruns bizantinos que utiliza criptografia de limiar para garantir a integridade dos dados armazenados e controlar as ações dos clientes; ***(2)*** um protocolo para recuperação pró-ativa dos servidores que implementam este sistema de quóruns; e ***(3)*** uma avaliação experimental do protocolo proposto neste trabalho, comparando-o com outro protocolo encontrado na literatura que oferece um serviço com as mesmas características em termos de resiliência e semântica das operações.

2. Preliminares

2.1. Modelo de Sistema

O sistema é formado por um conjunto ilimitado $C = \{c_1, c_2, ...\}$ de clientes que interagem com um conjunto de n servidores $S = \{s_1, ..., s_n\}$, que implementam um sistema de quóruns com semântica atômica das operações. Assim, clientes e servidores possuem identificadores únicos.

Todos os processos do sistema (tanto clientes quanto servidores) estão sujeitos a *faltas bizantinas* [Lamport et al. 1982]: um processo que apresenta este tipo de falha pode exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de sua especificação arbitrariamente. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. No entanto, após um processo faltoso ser recuperado, o mesmo volta a ficar correto (recuperação pró-ativa). Além disso, neste trabalho consideramos independência de falhas nos processos: a probabilidade de um processo sofrer uma falha é independente da probabilidade de outro processo também sofrer uma falha. A independência de falhas pode ser alcançada através do uso sistemático de diversidade [Obelheiro et al. 2006].

Em termos de garantias, o sistema permanece correto enquanto apresentar no máximo f servidores faltosos em um dado momento, sendo necessários $n = 3f + 1$ servidores. Além disso, um número ilimitado de clientes pode apresentar comportamento malicioso.

Consideramos o modelo de sistema assíncrono, onde os tempos para transmissões de mensagens e para computações locais nos processos são desconhecidos. Para terminação, a única exigência de nossos protocolos é que se um processo envia uma mensagem infinitas vezes para outro processo correto, então tal mensagem acabará por ser entregue no receptor (*fair-lossy link*). No entanto, para simplificar a apresentação dos protocolos, toma-se como hipótese que as comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados. Estes canais podem ser implementados sobre um sistema assíncrono através de SSL/TLS [Dierks and Allen 1999].

Como comentado, nossos protocolos utilizam criptografia de limiar para controlar as ações dos clientes e garantir a integridade dos dados armazenados. Assim, consideramos que na inicialização do sistema cada servidor recebe a sua chave parcial que será utilizada na elaboração de assinaturas parciais. Um servidor correto nunca revela a sua chave parcial. Estas chaves são geradas e distribuídas por um administrador correto que somente é necessário na iniciação do sistema. 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.

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 (anexados) em certas mensagens, evitando a utilização de mensagens antigas em algumas operações executadas pelos clientes. Consideramos que os clientes não escolhem *nonces* repetidos, i.e., já utilizados.

2.2. Sistemas de Quóruns Bizantinos

Sistemas de quóruns Bizantinos [Malkhi and Reiter 1998a], 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 fazem acesso, de fato, a somente um conjunto particular de servidores ao invés de todos os servidores.

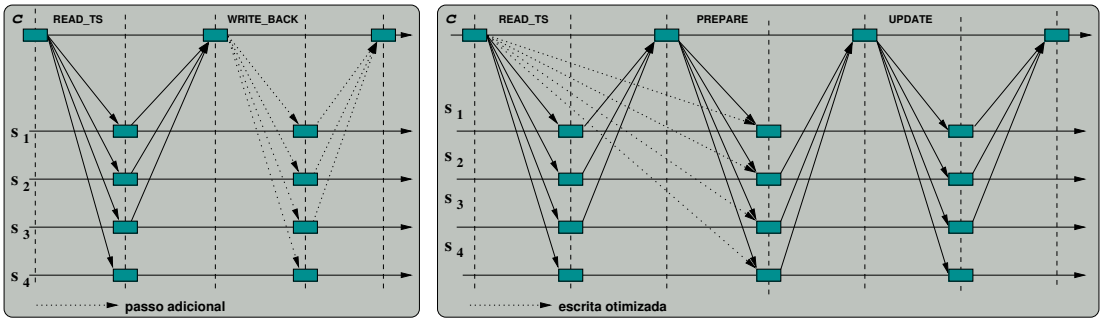
Servidores em um sistema de quóruns organizam-se em subconjuntos denominados quóruns. Cada dois quóruns de um sistema mantêm um número suficiente de servidores corretos em comum (garantia de consistência), sendo que existe pelo menos um quórum no sistema formado somente por servidores corretos (garantia de disponibilidade) [Malkhi and Reiter 1998a]. Os clientes realizam operações de leitura e escrita em registradores replicados por estes quóruns, cujos tamanhos para operações de leitura e escrita podem ser iguais (quóruns simétricos) ou não (quóruns assimétricos). Cada registrador detém um par $\langle v, t \rangle$ com um valor v do dado armazenado e uma estampilha de tempo (*timestamp*) t associada.

Dentre as várias propostas para sistemas de quóruns [Malkhi and Reiter 1998a, Malkhi and Reiter 1998b, Liskov and Rodrigues 2006], este trabalho propõe uma extensão ao BFT-BC [Liskov and Rodrigues 2006], permitindo a recuperação pró-ativa dos servidores do sistema. Este protocolo foi escolhido por apresentar resiliência ótima ($n = 3f + 1$, utilizando quóruns de $2f + 1$ servidores), tolerar clientes maliciosos e implementar um registrador com semântica atômica das operações [Lamport 1978].

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 BFT-BC em relação a outros sistemas que armazenam dados auto-verificáveis está justamente na forma de garantir a integridade destes dados, onde o BFT-BC utiliza um conjunto de assinaturas de servidores e outros protocolos, como o *f-dissemination quorum system* [Malkhi and Reiter 1998a, Malkhi and Reiter 1998b], utilizam assinaturas de clientes e, portanto, não toleram clientes maliciosos[1].

[1]O protocolo *f-masking quorum system* [Malkhi and Reiter 1998a, Malkhi and Reiter 1998b] também tolera clientes maliciosos, mas requer replicação em $4f + 1$ servidores, pois não armazena dados auto-verificáveis.

Assim, para manter suas semânticas de consistência, o BFT-BC utiliza um mecanismo de provas assinadas pelos servidores em todas as suas etapas de execução. Desta maneira, para um cliente ingressar em uma nova fase do algoritmo, é necessário que o mesmo apresente uma prova de que completou a fase anterior. Esta prova, chamada de certificado, nada mais é que o conjunto das mensagens (assinadas) de resposta coletadas de um quórum de servidores na fase anterior. Por exemplo, para o cliente escrever no quórum é preciso que ele tenha terminado uma escrita anterior. Através do uso desta técnica, o BFT-BC emprega 2 ou 4 passos de comunicação para realizar leituras e 4 ou 6 passos de comunicação para executar escritas (Figura 1).



(a) Leitura (b) Escrita

Figura 1. Execuções de operações usando o BFT-BC.

De maneira geral, o algoritmo BFT-BC executa da seguinte forma: **Leitura** (Figura 1(a)) – o cliente requisita um conjunto de pares $\langle v, t \rangle$ válidos de um quórum e seleciona aquele com o maior *timestamp*. Caso todos os pares retornados possuam este mesmo maior *timestamp* e um mesmo valor (o que ocorre em execuções sem escritas concorrentes e sem falhas), a execução termina. Caso contrário, o cliente realiza um passo adicional de reescrita (*write back*) no sistema e espera confirmações até determinar que um quórum de servidores possui o valor mais atual; **Escrita** (Figura 1(b)) – o cliente tem duas opções. Em um cenário normal, o cliente obtém um conjunto de *timestamps* de um quórum como na leitura e então realiza a preparação de sua escrita, fase em que tenta obter de um quórum um conjunto de provas necessário para completar sua operação. Em caso de sucesso nesta preparação, o cliente escreve no quórum, esperando um conjunto de confirmações. Em um cenário alternativo, o cliente pode, em vez da escrita normal, executar um protocolo otimizado de escrita, realizando em uma única fase as etapas de coleta de *timestamp* e de preparação da escrita (linhas pontilhadas da Figura 1(b)).

2.3. Criptografia de Limiar

A principal ferramenta utilizada neste trabalho é um **esquema de assinatura de limiar** (*threshold signature scheme* - TSS) [Shoup 2000] através do qual é possível controlar as ações dos clientes e garantir a integridade dos dados armazenados pelos servidores, além de facilitar o processo de recuperação dos mesmos. 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. Além disso, o distribuidor envia estas chaves para n partes diferentes, chamados **portadores**. Assim, 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 fase de configuração, o sistema está apto à gerar assinaturas. Deste modo, na obtenção de uma assinatura 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:

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

Neste trabalho, utilizamos 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.

3. PBFT-BC: Sistemas de Quóruns Bizantinos Pró-Ativos

Esta seção apresenta nossa abordagem para construção de um registrador que oferece operações de leitura e escrita, implementado por replicação através de quóruns bizantinos. Nosso protocolo é inspirado no BFT-BC [Liskov and Rodrigues 2006] e portanto tolera clientes maliciosos. A principal característica introduzida em nosso protocolo é a possibilidade de recuperação pró-ativa de servidores (Seção 3.2), sendo portanto chamado de PBFT-BC (*proactive Byzantine fault-tolerance for Byzantine clients*). Além disso, o PBFT-BC possui um desempenho superior em relação ao BFT-BC (Seção 4) e também apresenta resiliência ótima ($n = 3f + 1$).

É fundamental tolerar clientes maliciosos em sistemas de quóruns, pois os mesmos podem executar as seguintes ações com o intuito de corromper o sistema [Liskov and Rodrigues 2006]: *(i)* escrever valores diferentes associados com o mesmo *timestamp*; *(ii)* executar o protocolo parcialmente, atualizando os valores armazenados 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.

Diante disso, protocolos desenvolvidos para este fim devem impossibilitar (ou mascarar) estas ações executadas por clientes maliciosos. Neste sentido, nosso protocolo utiliza um esquema de assinatura de limiar para controlar as ações executadas pelos clientes e garantir a integridade dos dados armazenados. Desta forma, consideramos a existência de um administrador correto que desempenhará a função de distribuidor no esquema, onde os servidores desempenham o papel de portadores e os clientes de combinadores. Assim, o administrador gera e distribui as chaves parciais para os servidores. Além disso, toda informação pública (chave pública e as chaves de verificação) é enviada para todos os servidores. Posteriormente, os clientes obtêm estas informações a partir destes servidores, enviando uma requisição e aguardando por $f + 1$ respostas iguais. Note que o **administrador somente é necessário na iniciação do sistema**, durante a fase de geração e distribuição de chaves e verificadores.

Nosso protocolo utiliza quóruns de $2f + 1$ réplicas, deste modo utilizamos um esquema $(n, 2f + 1)$-TSS, i.e., para gerar uma assinatura válida é necessário um quórum de servidores. Assim, todas as ações realizadas pelos clientes precisam ser autorizadas por um quórum de servidores. Como o BFT-BC (Figura 1), o PBFT-BC emprega 2 a 4 passos de comunicação (1 a 2 fases) para realizar leituras e 4 a 6 passos de comunicação (2 a 3 fases) para executar escritas.

Para progredir nas operações o 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 que garante a integridade destes dados. Esta assinatura é gerada através do esquema $(n, 2f + 1)$-TSS, garantindo a presença de um quórum de servidores na sua elaboração. Dois tipos de certificados são utilizados:

Certificado de Preparação (*Prepare Certificate*): o cliente utiliza este certificado para provar que preparou uma escrita (um quórum de servidores aprovam a escrita). Já os servidores o utilizam 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 aprovam a escrita de v com o *timestamp* $cp.ts$. Para um certificado de preparação ser válido 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 (*Write Certificate*): 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. Para um certificado de escrita ser válido 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. No entanto, para simplificar a apresentação do protocolo, consideraremos que os servidores armazenam um único objeto. Deste modo, 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 que i sabe que foi executada por pelo menos um quórum de servidores; (5) SK_i – chave parcial de i usada pelo mecanismo de assinatura de limiar; (6) VK_i e VK – chaves de verificação usadas para gerar provas de validade das assinaturas parciais; e (7) PK – chave pública do serviço usada para validar certificados. Além disso, 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 $VK_1, ..., VK_n$ – chaves de verificação usadas para validar assinaturas parciais.

3.1. Protocolos de Escrita e Leitura

Esta seção apresenta os protocolos para escrita e leitura de um objeto no sistema. Para o correto funcionamento destes protocolos, 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 seqüência sejam idênticos. *Timestamps* são incrementados através da função $succ(ts, c) = \langle ts.seq + 1, c\rangle$.

Os Pseudocódigos 1 e 2 apresentam o protocolo de escrita executado por clientes e servidores, respectivamente. Estes pseudocódigos representam a versão não otimizada do protocolo, a qual demanda 3 fases para executar uma escrita (6 passos de comunicação). No entanto, é possível aglutinar as funções das duas primeiras fases em apenas uma fase (Seção 2.2), diminuindo o número de trocas de mensagens entre os processos.

Na primeira etapa o cliente define o *timestamp* da escrita e na segunda obtém um certificado de preparação para esta escrita que é definitivamente processada na terceira etapa. A evolução entre as fases do protocolo é baseada no uso de certificados, sendo este processamento a principal diferença do PBFT-BC em relação ao BFT-BC, além da recuperação pró-ativa (Seção 3.2). Assim, para gerar um certificado no PBFT-BC o cliente deve esperar por um quórum de

assinaturas parciais válidas (etapas $w2.2$ e $w3.2$) e então combiná-las (etapas $w2.3$ e $w3.3$) para obter a assinatura do serviço, a qual provará a validade deste certificado. Note que na validação de um certificado é necessário verificar apenas uma única assinatura (assinatura do serviço), diferindo do BFT-BC onde um quórum completo de assinaturas deve ser verificado.

Pseudocódigo 1 Protocolo usado pelo cliente c para escrever o valor $value$.

- $w1.1$ Client c sends a message $\langle \text{READ\_TS}, nonce \rangle$ to all replicas.
- $w1.2$ c waits for a quorum $(2f+1)$ of valid $read\_ts$ replies from different servers. A reply m_i from server i is valid if it is well-formed, i.e., $m_i = \langle \text{READ\_TS\_REPLY}, p, nonce \rangle$ where p is a valid prepare certificate (well-formed and the service signature is verified). Moreover, m_i should be correctly authenticated, i.e., its $nonce$ matches the $nonce$ used in step $w1.1$.
- $w1.3$ Among the prepare certificates received in step $w1.2$, c selects the certificate containing the largest timestamp, called P_{max}.
- $w2.1$ c sends a message $\langle \text{PREPARE}, P_{max}, ts, h(value), W_{cert} \rangle$ to all replicas. Here $ts \leftarrow succ(P_{max}.ts, c)$, h is a hash function and W_{cert} is a write certificate of c's last write or $null$ if this is c's first write.
- $w2.2$ c waits for a quorum $(2f+1)$ of valid $prepare$ replies from different servers. A reply m_i from server i is valid if it is well-formed, i.e., $m_i = \langle \text{PREPARE\_REPLY}, \langle ts_i, hash_i \rangle, \langle a_i, v_i \rangle \rangle$ where ts_i and $hash_i$ match the values defined in step $w2.1$ (ts and $h(value)$, respectively). Moreover, m_i is valid if $Thresh\_verifyS(\langle ts_i, hash_i \rangle, \langle a_i, v_i \rangle, PK, VK, VK_i)$ is $true$.
- $w2.3$ c combines the $2f+1$ correct signature shares received in step $w2.2$, calling $Thresh\_combineS(a_1, ..., a_{2f+1}, PK)$, and obtains the service signature A for the tuple $\langle ts, h(value) \rangle$. Then, c forms a prepare certificate P_{new} for ts and $h(value)$ by using A.
- $w3.1$ c sends a message $\langle \text{WRITE}, value, P_{new} \rangle$ to all replicas.
- $w3.2$ c waits for a quorum $(2f+1)$ of valid $write$ replies from different servers. A reply m_i from server i is valid if it is well-formed, i.e., $m_i = \langle \text{WRITE\_REPLY}, ts_i, \langle a_i, v_i \rangle \rangle$ where ts_i matches the value ts defined in step $w2.1$ and $Thresh\_verifyS(ts_i, \langle a_i, v_i \rangle, PK, VK, VK_i)$ is $true$.
- $w3.3$ c combines the $2f+1$ correct signature shares received in step $w3.2$, calling $Thresh\_combineS(a_1, ..., a_{2f+1}, PK)$, and obtains the service signature A for the timestamp ts. Then, c forms a write certificate W_{cert} for ts by using A. This certificate is used in the c's next write.

Pseudocódigo 2 Protocolo de escrita executado pelo servidor i.

Upon receipt of $\langle \text{READ\_TS}, nonce \rangle$ **from** client c

- $w1.1$ i sends a reply $\langle \text{READ\_TS\_REPLY}, P_{cert}, nonce \rangle$ to c.

Upon receipt of $\langle \text{PREPARE}, P_c, ts, hash, W_c \rangle$ **from** client c

- $w2.1$ if request is invalid or $ts \neq succ(P_c.ts, c)$, discart request without replying to c. A $prepare$ request is invalid if either certificate P_c or W_c is invalid (not well-formed or the service signature does not verify).
- $w2.2$ if W_c is not $null$, set $max_{ts} \leftarrow max(max_{ts}, W_c.ts)$, and remove from P_{list} all entries e such that $e.ts \leq max_{ts}$.
- $w2.3$ if P_{list} contains an entry for c with a different ts or $hash$, discart request without replying to c.
- $w2.4$ if $\langle c, ts, hash \rangle \notin P_{list}$ and $ts > max_{ts}$, add $\langle c, ts, hash \rangle$ to P_{list}.
- $w2.5$ i generates its signature share $\langle a_i, v_i \rangle \leftarrow Thresh\_sign(SK_i, VK_i, VK, \langle ts, hash \rangle)$.
- $w2.6$ i sends a reply $\langle \text{PREPARE\_REPLY}, \langle ts, hash \rangle, \langle a_i, v_i \rangle \rangle$ to c.

Upon receipt of $\langle \text{WRITE}, value, P_{new} \rangle$ **from** client c

- $w3.1$ if request is invalid or $P_{new}.hash \neq h(value)$, discart request without replying to c. A $write$ request is invalid if the prepare certificate P_{new} is invalid (not well-formed or the service signature does not verify). Here h is a hash function.
- $w3.2$ if $P_{new}.ts > P_{cert}.ts$, set $data \leftarrow value$ and $P_{cert} \leftarrow P_{new}$.
- $w3.3$ i generates its signature share $\langle a_i, v_i \rangle \leftarrow Thresh\_sign(SK_i, , VK_i, VK, P_{new}.ts)$.
- $w3.4$ i sends a reply $\langle \text{WRITE\_REPLY}, P_{new}.ts, \langle a_i, v_i \rangle \rangle$ to c.

A segunda fase da escrita é a mais importante, pois nesta fase os servidores (Pseudocódigo 2) verificam se: (1) o *timestamp* da escrita é correto; (2) o cliente está realizando apenas uma preparação de escrita; (3) o valor proposto para escrita não é diferente de um possível valor anteriormente proposto para o mesmo *timestamp*; e (4) o cliente completou sua escrita anterior. O item (1) é verificado na etapa $w2.1$. Já os itens (2), (3) e (4) são verificados nas etapas $w2.2$ e $w2.3$, onde cada servidor utiliza a lista das escritas preparadas (P_{list}). Uma característica

importante do uso desta lista é que um cliente não é capaz de preparar diversas escritas. Assim, um cliente malicioso não consegue preparar múltiplas escritas para serem executadas por um aliado após sua exclusão do sistema, limitando os danos causados por estes clientes.

Os Pseudocódigos 3 e 4 apresentam o protocolo de leitura executado por clientes e servidores, respectivamente. As leituras geralmente são completadas em apenas uma fase. No entanto, pode ser necessária uma fase adicional de *write back* [Malkhi and Reiter 1998b], onde o cliente escreve o valor lido em um número suficiente de servidores, de forma a garantir que esta informação mais atual está armazenada em pelo menos um quórum de servidores. Esta fase adicional garante linerialização [Herlihy and Wing 1990], i.e., as operações realizadas no sistema parecem seguir uma ordem seqüencial garantindo a semântica atômica das mesmas.

Pseudocódigo 3 Protocolo de leitura executado pelo cliente c.

$r1.1$ Client c sends a message $\langle \text{READ}, nonce \rangle$ to all replicas.

$r1.2$ c waits for a quorum $(2f + 1)$ of valid $read$ replies from different servers. A reply m_i from server i is valid if it is well-formed, i.e., $m_i = \langle \text{READ\_REPLY}, value, P_{cert_i}, nonce \rangle$ where P_{cert_i} is a valid prepare certificate (well-formed and the service signature is verified) and $P_{cert_i}.hash = h(value)$. Here h is a hash function. Moreover, m_i should be correctly authenticated, i.e., its $nonce$ matches the $nonce$ used in step $r1.1$.

$r1.3$ Among all replies received in step $r1.2$, c selects the reply with the prepare certificate containing the largest timestamp and returns the $value$ related with this reply. Also, if all timestamps obtained in step $r1.2$ are equal the read protocol ends.

$r2.1$ Otherwise the client performs the write back phase for the largest timestamp. This is identical to phase 3 of writing (steps $w3.1$, $w3.2$ and $w3.3$), except that the client needs to send only to replicas that are out of date, and it must wait only for enough responses to ensure that a quorum $(2f + 1)$ of replicas now has the new information.

O protocolo de leitura executado pelos servidores é bastante simples. Nestas operações, os servidores apenas necessitam enviar uma resposta para o cliente com o valor armazenado e o certificado que prova a integridade deste valor.

Pseudocódigo 4 Protocolo de leitura executado pelo servidor i.

Upon receipt of $\langle \text{READ}, nonce \rangle$ **from** client c

r1.1 i sends a reply $\langle \text{READ\_REPLY}, data, P_{cert}, nonce \rangle$ to c.

Corretude. As condições de corretude do PBFT-BC, bem como as provas de que o PBFT-BC contempla estas condições, são iguais às apresentadas em [Liskov and Rodrigues 2006], pois as mesmas são baseadas na validade dos certificados utilizados no protocolo e nas propriedades dos quóruns em si. A semântica atômica das operações é assegurada pela fase de *write back* do valor lido, através da qual as leituras seguintes retornarão este valor ou um valor mais atual. Além disso, nosso protocolo é livre de esperas (*wait-freedom*) pois cada fase requer respostas de somente $n - f$ servidores (um quorum), os quais sempre estarão disponíveis no sistema.

Otimização do Protocolo*: Eliminando a verificação de assinaturas parciais*. Dois dos passos mais custosos do esquema de assinaturas de limiar são: (1) verificação da validade das assinaturas parciais (passos $w2.2$ e $w3.2$ do Pseudocódigo 1), executado pelos clientes; e (2) geração das provas de validade das assinaturas parciais (passos $w2.5$ e $w3.3$ do Pseudocódigo 2), executado pelos servidores. Nestas etapas onde assinaturas são geradas, caso não exista servidores faltosos no sistema, o primeiro quórum de respostas recebido por um cliente terá o número suficiente de assinaturas parciais corretas (todas as assinaturas parciais deste quórum) para gerar a assinatura do serviço. Então, modificamos o algoritmo de forma que o cliente primeiro tenta obter a assinatura do serviço a partir do primeiro quórum de assinaturas parciais recebido, sem verificar a validade das mesmas. Caso a assinatura obtida não seja válida para o dado que o cliente está tentando obter a assinatura (certificado), então existem assinaturas parciais inválidas dentre as utilizadas na combinação. Neste caso, o cliente deve aguardar por novas assinaturas

parciais e fazer todas as combinações possíveis (usando $2f + 1$ assinaturas parciais) até obter uma assinatura válida. Note que, no pior caso, o cliente obtém f assinaturas parciais inválidas no primeiro quórum de respostas, sendo necessário aguardar por mais f respostas para então obter uma assinatura válida. Além disso, como sempre teremos pelo menos um quórum de servidores corretos, sempre é possível obter uma assinatura válida. Em um cenário livre de falhas esta otimização reduz drasticamente o tempo de processamento criptográfico requerido nas operações de escrita. Outra vantagem desta otimização está relacionada com a recuperação dos servidores (Seção 3.2), onde suas chaves parciais e de verificação são atualizadas. Neste sentido, não é necessário que os clientes atualizem as chaves de verificação após cada processo de recuperação, pois os mesmos não verificam assinaturas parciais. Alternativamente, para evitar excessivas combinações em cenários com falhas, o cliente poderia executar o protocolo normal quando não obtivesse a assinatura correta a partir do primeiro quórum de respostas.

3.2. Recuperação Pró-Ativa

Grande parte dos protocolos desenvolvidos para sistemas tolerantes a faltas bizantinas considera que um determinado número de servidores pode falhar durante todo o tempo de vida do sistema. No entanto, muitos sistemas (ou a grande maioria deles) são projetados para permanecerem em funcionamento durante muito tempo, fazendo com que esta premissa seja difícil de ser observada na prática. De fato, quanto maior for o tempo de vida de um sistema, mais tempo um adversário terá para invadir um número de servidores maior do que o limite suportado pelo sistema, fazendo com que o mesmo passe a funcionar de forma incorreta.

Neste sentido, desenvolvemos um protocolo para recuperação de servidores[2] através do qual um servidor faltoso volta a se comportar corretamente. Deste modo, o sistema passa a tolerar qualquer número de faltas durante o seu ciclo de vida, desde que apenas um determinado número de faltas (f) ocorra simultaneamente dentro de um pequeno período predeterminado, chamado **janela de vulnerabilidade**. Assim, um servidor invadido (faltoso) pode ser recuperado através das seguintes ações: *(1)* reinicialização do *hardware* (computador) e configurações do sistema; *(2)* recarregamento do código a partir de um local seguro; *(3)* recuperação do estado do servidor, que pode estar corrompido; e *(4)* modificação (tornar obsoleta) de qualquer informação confidencial que um adversário possa ter obtido.

3.2.1. Premissas Adicionais

Para implementar recuperação pró-ativa algumas suposições adicionais são necessárias. Neste trabalho, utilizamos as mesmas premissas adotadas em [Castro and Liskov 2002]:

Par de chaves. Cada processo possui um par de chaves pública-privada, sendo a chave privada conhecida apenas pelo próprio processo. As chaves públicas são conhecidas por todos os participantes do sistema através de certificados. Estas chaves apenas são utilizadas no restabelecimento de canais autenticados entre os processos, i.e., para compartilhamento de um segredo.

Criptografia segura. Cada servidor possui um co-processador seguro, que armazena a sua chave privada. Assim, mensagens podem ser assinadas ou decifradas sem que sua chave privada seja exposta[3]. Este co-processador também armazena um contador incremental que nunca é decrementado, cujo valor é adicionado nas mensagens assinadas para evitar ataques de *replay*.

Memória somente de leitura. Cada servidor armazena as chaves públicas dos outros servidores, bem como a chave pública do serviço PK em uma memória que não pode ser violada através de uma invasão. Além disso, um *hash* do código do servidor também é armazenado nesta memória.

[2]Como não é possível determinar se um servidor foi ou não invadido, a recuperação é pró-ativa, i.e., até mesmo servidores corretos executam o processo de recuperação.

[3]Caso considerássemos que as chaves privadas pudessem ser obtidas por um adversário que obtivesse sucesso em uma invasão, as mesmas deveriam ser atualizadas durante o processo de recuperação.

Watchdog Timer. Cada servidor possui um *watchdog timer* que periodicamente interrompe o processamento e passa o controle para um monitor de recuperação (que está armazenado na memória somente de leitura). Um adversário é incapaz de mudar o ciclo de interrupções deste *timer* sem ter acesso físico ao servidor.

3.2.2. Protocolo Modificado

A principal modificação no protocolo está relacionada com as janelas de vulnerabilidade. Neste sentido, estas janelas são numeradas de forma crescente e os servidores adicionam esta informação nas respostas enviadas para os clientes, i.e., cada resposta é acompanhada de um parâmetro adicional informando em qual janela encontra-se o servidor que a enviou. Assim, um cliente descobre que os servidores avançaram para a próxima janela quando receber pelo menos $f + 1$ respostas indicando esta mudança no sistema. Além disso, os certificados de preparação também passam a ter um novo atributo que indica em qual janela o mesmo foi gerado.

Em escritas concorrentes com uma recuperação (troca de janela de vulnerabilidade), pode ser necessário o reinício de uma etapa da escrita pelos clientes, caso os mesmos estejam obtendo respostas para gerar um certificado (etapas 2 e 3). De fato, neste procedimento as chaves parciais dos servidores são atualizadas e uma assinatura somente é gerada com sucesso caso todas as assinaturas parciais utilizadas na sua elaboração foram realizadas com chaves parciais referentes à mesma janela de vulnerabilidade. Já as operações de leitura não sofrem interferência do processo de recuperação, a não ser, é claro, durante a reinicialização do sistema onde o servidor fica indisponível por um pequeno tempo.

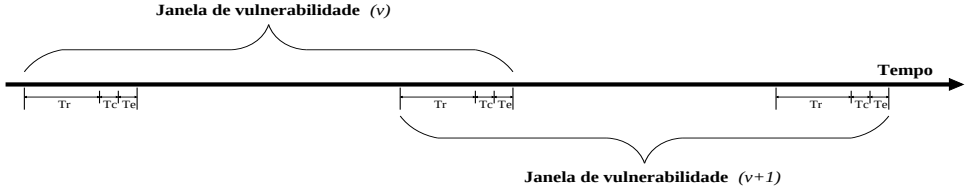


Figura 2. Relação entre as janelas de vulnerabilidade e o processo de recuperação.

O tempo gasto na recuperação de um servidor pode ser dividido em três partes (Figura 2): Tr – tempo para reiniciar o sistema; Tc – tempo para a atualização das chaves; e Te – tempo para atualização do estado. Uma janela de vulnerabilidade começa no início de uma recuperação e termina no fim da próxima recuperação (Figura 2). Durante este período, até f de $3f + 1$ servidores podem falhar no sistema.

É impossível desenvolver protocolos de recuperação pró-ativa em ambientes totalmente assíncronos [Sousa et al. 2005], pois nestes ambientes não podemos considerar nenhuma premissa relacionada com o tempo. Assim, tem-se o problema de garantir o início da execução do protocolo de recuperação periodicamente e, ainda mais complicado, garantir que este protocolo termine. O primeiro problema é resolvido através da utilização de *watchdogs* que geram interrupções periódicas. Já para garantir a terminação do protocolo podemos utilizar três métodos diferentes: *(1)* assumir que um adversário não consegue controlar (atrasar mensagens) canais entre processos corretos [Zhou et al. 2002b]; *(2)* utilizar um sistema híbrido, onde a parte responsável pela recuperação é síncrona [Sousa et al. 2007a]; ou *(3)* caso a recuperação não terminar dentro de um determinado tempo, o servidor em questão alerta um administrador que pode realizar ações administrativas com o objetivo de permitir o fim da recuperação [Castro and Liskov 2002]. Nosso sistema suporta qualquer uma destas técnicas de controle do fim do processo de recuperação nos servidores, cujas etapas são apresentadas a seguir:

Reinicialização do Sistema. Cada servidor é reiniciado quando seu *watchdog timer* gerar uma interrupção. Porém, antes de reiniciar, cada servidor envia uma mensagem para os outros servidores informando que irá executar a recuperação, i.e., avançar para a próxima janela de vul-

nerabilidade. Assim, um servidor reinicia caso receber $f + 1$ destas mensagens, mesmo que seu *watchdog timer* ainda não tenha gerado a interrupção. Este procedimento visa acelerar o processo de recuperação. Além disso, o monitor de recuperação salva o estado do servidor ($data$ e P_{cert}) e as suas chaves parcial e de verificação. Então, o monitor reinicia o sistema com o código correto e inicializa o servidor com o estado anteriormente armazenado. A integridade do sistema operacional e do código do servidor é verificada através do *hash* dos mesmos armazenados na memória somente de leitura. Assim, é garantido que o código do servidor está correto e que o mesmo mantém o seu estado. Este estado pode estar corrompido mas deve ser utilizado durante o processo de recuperação para garantir as propriedades do sistema quando o servidor que está realizando a recuperação não é faltoso. Caso contrário, o procedimento de recuperação poderia fazer com que o limite de faltas fosse excedido. No entanto, o servidor poderia ter sido invadido. Então, ainda é necessário recuperar o estado do servidor e atualizar os dados confidenciais que puderam ser obtidos por um invasor (chaves parciais e de sessão).

Atualização das Chaves. As chaves de sessão são atualizadas da mesma forma que em [Castro and Liskov 2002]. Assim, um processo i atualiza sua chave de sessão usada para autenticar um canal com o processo j enviando para j um segredo cifrado com a chave pública de j (para apenas j ser capaz de acessar este segredo) e assinado com a chave privada de i (para garantir autenticidade). Deste modo, uma nova chave de sessão é estabelecida para autenticar as mensagens que i envia para j. Os servidores assinam suas mensagens através do co-processador seguro, que adiciona o valor do contador para evitar ataques de *replay*, i.e, j apenas considerará esta mensagem caso o valor atachado seja maior do que o valor recebido na última mensagem de i. Além disso, os servidores atualizam suas chaves parciais e de verificação através de um protocolo de atualização pró-ativa como o APSS [Zhou et al. 2002b], que também é usado por outros sistemas pró-ativos [Zhou et al. 2002a, Marsh and Schneider 2004]. Neste procedimento, as chaves assimétricas dos servidores são utilizadas para envio de mensagens confidenciais.

Recuperação do Estado. Para recuperar seu estado, basta que um servidor realize uma leitura normal em um quórum de servidores (considerando o servidor que está executando a recuperação). O valor lido é usado para atualizar as variáveis $data$ e P_{cert}. As outras variáveis são reiniciadas como na inicialização do sistema, i.e., P_{list} como um conjunto vazio e max_{ts} como nulo. Os dados destas variáveis são utilizados para impossibilitar que um cliente malicioso prepare mais de uma escrita e como são "perdidos" durante a recuperação, a seguinte técnica é utilizada para controlar as ações dos clientes. Como os certificados de preparação também contêm o número v da janela de vulnerabilidade, os servidores não consideram corretos (na terceira etapa da escrita) os certificados de preparação gerados em janelas de vulnerabilidade anteriores. Assim, caso um cliente receba respostas de $f + 1$ servidores indicando que avançaram para a próxima janela de vulnerabilidade, tal cliente deve reiniciar a segunda etapa do protocolo e esperar por respostas de um quórum de servidores que estão nesta nova janela[4].

4. Experimentos

Esta seção apresenta uma análise acerca do desempenho do PBFT-BC através de uma comparação com seu precursor, o BFT-BC [Liskov and Rodrigues 2006]. Em particular, estamos interessados em observar a latência envolvida na execução das operações, uma vez que os efeitos causados por concorrência e falhas são praticamente os mesmos em ambos os protocolos. Assim, implementamos estes protocolos usando a linguagem de programação Java[5]. Os

[4]Poderíamos evitar a necessidade de reiniciar esta etapa fazendo com que os servidores considerem como válidos os certificados gerados na janela de vulnerabilidade presente ou na anterior. Neste caso, um cliente faltoso poderia preparar no máximo duas (sem otimização – 3 fases) ou quatro (versão otimizada – 2 fases) escritas.

[5]Para criptografia de limiar, adaptamos a biblioteca encontrada em `http://threshsig.sf.net/`, que é uma implementação do protocolo proposto em [Shoup 2000].

mecanismos de recuperação pró-ativa não foram analisados, pois somente terão influência na latência das operações realizadas durante o processo de recuperação.

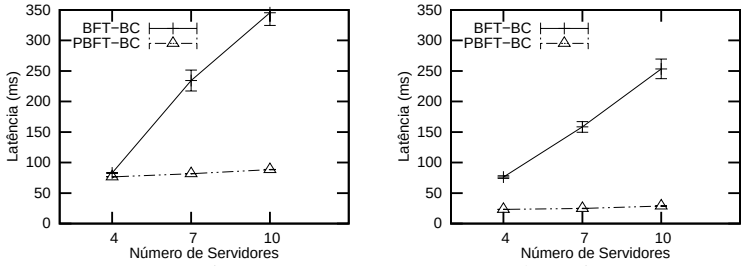
A Tabela 4 apresenta uma análise dos custos envolvidos nas operações de escrita, onde percebemos que no BFT-BC é necessário realizar verificações na ordem quadrática em relação ao número de faltas suportadas pelo sistema, enquanto que no PBFT-BC este custo é apenas linear. Isso está relacionado com a verificação de um quórum de certificados na primeira fase dos protocolos, onde no BFT-BC cada certificado é validado por um quórum de assinaturas e no PBFT-BC por apenas uma assinatura. Este custo também se reflete nas operações de leitura, onde, desconsiderando a possível fase de *write back*, no BFT-BC um cliente precisa realizar $4f^2+4f+1$ verificações enquanto que no PBFT-BC apenas $2f+1$ verificações são necessárias.

Tabela 1. Custos do protocolo de escrita para um cenário sem falhas.

Fase	BFT-BC		PBFT-BC	
	cliente	servidor	cliente	servidor
1ª	$(4f^2+4f+1)$ verificações	—	$(2f+1)$ verificações	—
2ª	$(2f+1)$ verificações	$(4f+2)$ verificações + 1 assinatura	1 combinação de $(2f+1)$ assinaturas parciais	2 verificações + 1 assinatura parcial
3ª	$(2f+1)$ verificações	$(2f+1)$ verificações + 1 assinatura	1 combinação de $(2f+1)$ assinaturas parciais	1 verificação + 1 assinatura parcial
Custo Total	$(4f^2+8f+3)$ verificações	$(6f+3)$ verificações + 2 assinaturas	$(2f+1)$ verificações + 2 combinações	3 verificações + 2 assinaturas parciais
	$(4f^2+14f+6)$ verificações + 2 assinaturas		$(2f+4)$ verificações + 2 assinaturas parciais + 2 combinações	

Visando analisar o comportamento destes sistemas na prática, alguns experimentos foram realizados no Emulab [White et al. 2002], em um ambiente consistindo de 11 máquinas *pc3000* (3.0 GHz 64-*bit* Pentium Xeon com 2GB de RAM e interface de rede gigabit) conectadas a um *switch* de 100Mb. A rede local é emulada como uma VLAN em um *switch* Cisco 4509 onde adicionamos uma latência de 10ms nas comunicações (para aproximar o ambiente a uma rede de larga escala). O ambiente de *software* utilizado foi o Red Hat Linux 6 com *kernel* 2.4.20 e máquina virtual Java de 32 *bits* versão 1.6.0_02. Todos os experimentos foram realizados com o compilador *Just-In-Time* (JIT) do Java ativado e uma fase de *warm-up* antecedeu cada experimento para transformar os *bytecodes* em código nativo.

Nos experimentos, utilizamos assinaturas RSA de 512 *bits* (tanto no esquema tradicional adotado pelo BFT-BC quanto no esquema baseado em criptografia de limiar utilizado pelo PBFT-BC) e fixamos o tamanho dos objetos, que eram escritos e lidos do sistema, em 1024 *bits*. Então, variamos o número de servidores com o objetivo de verificar a capacidade de escalabilidade dos sistemas em relação a este fator. Os valores aqui reportados compreendem o tempo médio necessário para a execução de uma operação por um cliente do sistema, obtido a partir de 1000 execuções da operação e excluindo-se 5% dos valores com maior desvio.



(a) Escrita (b) Leitura

Figura 3. PBFT-BC X BFT-BC.

A Figura 3 apresenta a latência média para a execução das operações, onde percebemos que o PBFT-BC apresentou um desempenho superior ao BFT-BC. Isto se deve principalmente

à otimização apresentada na seção 3.1 e ao fato dos certificados do PBFT-BC terem tamanho constante, com apenas uma assinatura, enquanto que os certificados do BFT-BC contêm um quórum de assinaturas. De fato, o número total de *bytes* (somando todas as fases do protocolo) trocados entre o cliente e cada servidor no PBFT-BC foi de aproximadamente 2756 na escrita e 1466 na leitura, para qualquer configuração do sistema. Já no BFT-BC foram trocados 5884/2229, 8120/2824 e 10292/3419 *bytes* na operação de escrita/leitura nos experimentos com 4, 7 e 10 servidores, respectivamente. Outro ponto a destacar é o baixo tempo necessário para elaborar uma assinatura parcial (aprox. 4.5ms) e para combinar um quórum destas assinaturas (0.45ms para quóruns de 3 servidores). Além disso, o tempo para elaborar uma assinatura RSA normal foi aprox. 1.5ms e as verificações consumiram aprox. 0.26ms, em ambos os protocolos.

5. Trabalhos Relacionados

Os trabalhos sobre sistemas de quóruns encontrados na literatura apresentam muitas variações, incluindo o tipo de falta suportada (parada ou maliciosa) e a semântica provida pelas operações (segura, regular ou atômica). Os primeiros protocolos que permitem o comportamento malicioso de processos são apresentados em [Malkhi and Reiter 1998a, Malkhi and Reiter 1998b]. Estes trabalhos abordam dois tipos de quóruns: *(1) f-dissemination quorum system*, que não suporta clientes maliciosos e requer $3f + 1$ servidores; e *(2) f-masking quorum system*, que suporta clientes maliciosos mas requer $4f + 1$ servidores. Além disso, [Malkhi and Reiter 1998b] é o primeiro trabalho que apresenta a fase de *write back* no protocolo de leitura, o que garante a semântica atômica das operações realizadas no sistema.

O sistema de quóruns que mais se assemelha ao PBFT-BC é apresentado em [Liskov and Rodrigues 2006]. Este sistema também suporta clientes maliciosos necessitando de apenas $3f + 1$ servidores. Para isso, utiliza assinaturas dos servidores na garantia da integridade dos dados armazenados. A grande diferença do nosso trabalho está relacionada com o uso de criptografia de limiar para tornar os dados auto-verificáveis, que além de melhorar significativamente o desempenho do sistema, possibilita a atualização das chaves parciais, o que facilita o desenvolvimento de mecanismos para recuperação pró-ativa dos servidores, pois nenhum dado precisa ser atualizado nos clientes.

Alguns protocolos para recuperação pró-ativa de servidores são encontrados na literatura [Castro and Liskov 2002, Zhou et al. 2002a, Marsh and Schneider 2004]. Nosso protocolo assume as mesmas premissas adotadas em [Castro and Liskov 2002], que apresenta um protocolo para replicação ativa, mas não utiliza criptografia de limiar para assinaturas. Outros trabalhos que utilizam criptografia de limiar são o COCA [Zhou et al. 2002a], que implementa uma autoridade certificadora e o CODEX [Marsh and Schneider 2004], que implementa um sistema de armazenamento de segredos. Estes trabalhos utilizam um mecanismo de atualização de chaves parciais chamado APSS [Zhou et al. 2002b], que também é usado por nosso protocolo. No entanto, a arquitetura destes sistemas é significativamente diferente da adotada neste trabalho, principalmente por utilizarem um servidor como representante do cliente, o que diminui a performance do sistema uma vez que são necessários mais passos de comunicação para realizar uma operação. Além disso, as premissas adicionais para recuperação pró-ativa, adotadas pelo COCA e CODEX, não são suficientes para garantir o progresso do sistema [Sousa et al. 2007b].

6. Conclusões e Trabalhos Futuros

Neste trabalho apresentamos um novo protocolo para sistemas de quóruns bizantinos (PBFT-BC), que tolera clientes maliciosos e possui resiliência ótima, além de fornecer um registrador com semântica atômica das operações. O PBFT-BC apresentou desempenho superior ao BFT-BC, protocolo que fornece um serviço com as mesmas características. Além disso, desenvolvemos um protocolo para recuperação pró-ativa dos servidores que implementam o PBFT-BC.

Os próximos passos deste trabalho consistem em estender este modelo para funcionamento em um ambiente dinâmico, onde processos podem entrar ou sair do sistema em qualquer momento. Neste sentido, a utilização de criptografia de limiar fornece a flexibilidade necessária para adaptação do protocolo às mudanças que ocorrem na composição do grupo de servidores.

Referências

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.

Castro, M. and Liskov, B. (2002). Practical Byzantine Fault-Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461.

Desmedt, Y. and Frankel, Y. (1990). Threshold Criptosystems. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'90*, pages 307–315. Springer-Verlag.

Dierks, T. and Allen, C. (1999). The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments.

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

Herlihy, M. and Wing, J. M. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565.

Lamport, L. (1986). On Interprocess Communication (part II). *Distributed Computing*, 1(1):203–213.

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

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*, pages 34–44.

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

Malkhi, D. and Reiter, M. (1998b). Secure and Scalable Replication in Phalanx. In *Proc. of 17th Symposium on Reliable Distributed Systems*, pages 51–60.

Marsh, M.-M. A. and Schneider, M.-F. B. (2004). CODEX: A Robust and Secure Secret Distribution System. *IEEE Transactions on Dependable Secure Computing*, 1(1):34–47.

Obelheiro, R. R., Bessani, A. N., Lung, L. C., and Correia, M. (2006). How Practical are Intrusion-Tolerant Distributed Systems? DI-FCUL TR 06–15, Dep. of Informatics, University of Lisbon.

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.

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

Sousa, P., Bessani, A., Correia, M., Neves, N., and Verissimo, P. (2007a). Resilient Intrusion Tolerance through Proactive and Reactive Recovery. *13th Pacific Rim International Symposium on Dependable Computing – PRDC 2007*, pages 373–380.

Sousa, P., Neves, N. F., and Verissimo, P. (2005). How Resilient are Distributed f Fault/Intrusion-Tolerant Systems? In *Proceedings of the International Conference on Dependable Systems and Networks - DSN'05*, pages 98–107.

Sousa, P., Neves, N. F., and Verissimo, P. (2007b). Hidden Problems of Asynchronous Proactive Recovery. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability - HotDep 2007*, page 5.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symposium on Operating Systems Design and Implementations*, pages 255 – 270.

Zhou, L., Schneider, F., and Van Rennesse, R. (2002a). COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368.

Zhou, L., Schneider, F. B., and Renesse, R. V. (2002b). APSS: Proactive Secret Sharing in Asynchronous Systems. Technical Report TR 2002-1877, Computer Science Department, Cornell University, Ithaca - New York.