Memória Compartilhada em Sistemas Bizantinos Dinâmicos*

**Eduardo Adilio Pelinson Alchieri[1], Alysson Neves Bessani[2] ,
Joni da Silva Fraga[3] , Fabíola Greve[4]**

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

Resumo. *Sistemas de quóruns Bizantinos são uma ferramenta útil para a implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas arbitrárias. Um dos grandes desafios na implementação desses sistemas está na reconfiguração do conjunto de servidores em redes dinâmicas devido à entradas e saídas. Esse trabalho apresenta os principais aspectos do* QUINCUNX*, um conjunto de protocolos tolerantes a faltas Bizantinas capazes de emular registradores em sistemas dinâmicos. O protocolo de reconfiguração é a principal contribuição do artigo e vem a ser o primeiro não baseado no consenso tolerante a faltas Bizantinas. Além disso, seu funcionamento é independente dos protocolos de leitura/escrita, o que permite seu uso com outras implementações de memória compartilhada.*

Abstract. *Byzantine quorum systems are a useful tool to implement consistent and available data storage systems in the presence of arbitrary faults. In this work we consider a dynamic variant of this system and propose a set of Byzantine fault-tolerant protocols, called* QUINCUNX*, that emulates a register in dynamic asynchronous systems. Particularly, we present a reconfiguration protocol that does not require consensus and that is loosely coupled with read/write protocols, making it easy to use with any other static Byzantine fault-tolerant register implementation.*

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 1998a, Malkhi and Reiter 1998b]. 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, 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

*Alysson Bessani é financiado pela FCT através de seu programa multi-anual (LaSIGE) e através dos projetos ReD (PTDC/EIA-EIA/109044/2008) e CMU-Portugal (CMU-PT/0002/2007).

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 dos maiores desafios no projeto dos sistemas de quóruns e, mais especificamente, no projeto de reconfiguração do conjunto de servidores que mantêm a memória compartilhada. Devido à entradas, saídas e falhas dos servidores, será preciso manter a consistência dos dados e garantir a sua disponibilidade. Muitas questões devem ser resolvidas. Há a necessidade de gerenciamento de visões do grupo de servidores que compõem o sistema, mas também da sua quantidade. Se muitos processos deixam o sistema, este pode perder a sua vivacidade. Ademais, o que fazer em caso de falhas sucessivas? E na ocorrência de reconfigurações concorrentes, quando novas reconfigurações se iniciam enquanto outras ainda estão em andamento ? Em qualquer situação, é preciso prover protocolos que mantenham as propriedades de vivacidade e de segurança do sistema, apesar do dinamismo. 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 [Lynch and Shvartsman 2002, Martin and Alvisi 2004, Rodrigues and Liskov 2004]. Todos esses trabalhos utilizam o protocolo fundamental do consenso como forma de implementar a reconfiguração e assim concordar com o conjunto de servidores que irá compor o sistema. No consenso, todos os processos concordam com uma valor proposto por um dos participantes. Essa abordagem, embora adequada, dado que o serviço de mudança de visões do conjunto de servidores é um protocolo de acordo (ou de consenso), não é a mais eficaz ou a mais indicada. De fato, o consenso torna-se impossível num ambiente assíncrono sujeito a falhas de servidores e, para resolvê-lo, abstrações de sincronia precisam ser incorporadas ao sistema, como os detectores de falhas [Chandra and Toueg 1996]. Ocorre que a manutenção da consistência de uma memória compartilhada com um conjunto estático de servidores pode ser feita sem necessidade de acordo e, portanto, num sistema totalmente assíncrono. Protocolos que implementam sistemas de quóruns em ambientes estáticos [Bazzi and Ding 2004, Malkhi and Reiter 1998a, Malkhi and Reiter 1998b] não precisam de uma primitiva forte de sincronização, como o consenso.

Até bem recentemente, ainda não se sabia se seria possível implementar reconfigurações (ou mudança de visões) sem necessidade de acordo. O trabalho seminal proposto por [Aguilera et al. 2011] responde a essa questão. Ele apresenta o *DynaStore*, um conjunto de algoritmos baseados na abstração *weak snapshot objects*, capaz de implementar uma memória dinâmica tolerante a faltas por parada (*crash*), onde as reconfigurações são executadas sem o auxilio de consenso. Neste sistema, as reconfigurações ocorrem em qualquer momento, gerando um grafo de visões a partir do qual é possível identificar uma sequência de visões onde os clientes executam operações.

Até onde sabemos, o *DynaStore* é o único sistema de memória compartilhada não baseado em consenso. Porém, ele apresenta uma grande desvantagem: o seu protocolo de reconfiguração é fortemente acoplado aos protocolos de leitura e escrita. Um estudo recente [Shraer et al. 2010] confirma que, nesse caso, o desempenho do sistema é fortemente afetado, inclusive, quando comparado com protocolos baseados no consenso. No caso, as leituras e escritas são severamente retardadas em caso de reconfigurações concorrentes com essas operações. Outra característica do *DynaStore*, é que ele somente suporta faltas por parada. Alguns protocolos de reconfiguração foram propostos para o modelo Bizantino [Martin and Alvisi 2004, Rodrigues and Liskov 2004], e de fato, este modelo seria o mais

adequado para redes dinâmicas (MANETs, P2P, etc.), que são altamente vulneráveis a ataques de agentes maliciosos, além de falhas devido à falta de recursos (memória, energia, etc.).

Neste artigo, apresentamos um novo protocolo de reconfiguração que responde às necessidades das redes dinâmicas e possui as seguintes características inovadoras: (i) é o primeiro protocolo de reconfiguração não baseado no consenso capaz de tolerar faltas Bizantinas. Além disso, (ii) o processo de mudança de visões é independente dos protocolos de leitura/escrita. Tal característica contribui para a melhoria de desempenho do sistema, em caso de concorrência das várias operações. A independência permite com que o protocolo de reconfiguração possa ser usado com outras implementações de memória compartilhada.

Neste trabalho, consideramos protocolos para sistemas de quóruns bizantinos do tipo f-disseminação [Malkhi and Reiter 1998b], que toleram faltas maliciosas nos servidores e nos clientes leitores, além de um número ilimitado de faltas por parada de clientes escritores. Os protocolos fazem parte de um sistema chamado QUINCUNX[1], proposto em [Alchieri 2011]. O QUINCUNX implementa um registrador atômico livre de espera (*wait-free*) [Herlihy 1991] e, ao mesmo tempo, permite reconfigurações no conjunto de servidores. Este sistema é composto por dois conjuntos de protocolos: *(1)* protocolos de leitura e escrita; e *(2)* protocolos para atualização de visões. Os protocolos de leitura e escrita são uma variante do PHALANX [Malkhi and Reiter 1998b], que foi a primeira implementação de registrador atômico tolerante a faltas bizantinas. A única diferença é que cada cliente deve verificar se está acessando a configuração mais atual do sistema[2], por isso estes protocolos não são explorados neste trabalho. Já os protocolos para atualização de visões implementam os procedimentos de reconfiguração do conjunto de servidores em tempo de execução e são o foco deste artigo. Para esses protocolos propomos uma abstração chamada *geradores de visões*. Os geradores são classificados de acordo com propriedades de vivacidade (asseguram que um gerador sempre gera alguma visão) e segurança (asseguram que cada gerador gera uma única visão). Neste trabalho, apresentamos o *gerador de visões vivo* que garante que alguma visão sempre será instalada pelo sistema.

O resto deste artigo é organizado da seguinte forma. A Seção 2 apresenta algumas definições preliminares. Os protocolos para reconfiguração do sistema são apresentados na Seção 3. A Seção 4 apresenta algumas discussões sobre os protocolos propostos. As conclusões do trabalho são apresentadas na Seção 5.

2. Definições Preliminares

2.1. Modelo de Sistema

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em três subconjuntos distintos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; um conjunto infinito de clientes leitores $R = \{r_1, r_2, ...\}$; e um conjunto infinito de clientes escritores $W = \{w_1, w_2, ...\}$. Cada processo do sistema (cliente ou servidor) possui um identificador único. Os servidores e os leitores estão sujeitos a faltas Bizantinas [Lamport et al. 1982], enquanto que os escritores apenas podem exibir comportamento de falha por parada. 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].

Consideramos um sistema distribuído assíncrono, onde não existem limites para o tempo

[1]Uma formação militar romana que permite reconfigurações nas linhas de batalha.

[2]Este é o único requisito para qualquer protocolo de leitura e escrita, definido para ambientes estáticos, utilizar os protocolos de atualização de visões do QUINCUNX e transformar-se em um protocolo dinâmico.

de transmissão de mensagens ou processamentos locais nos processos. Além disso, cada servidor do sistema tem acesso a um relógio local usado para iniciar reconfigurações. Estes relógios não são sincronizados e não existe qualquer limite (*bounds*) para seus desvios (*drifts*), sendo portanto apenas contadores. As comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados. Estes canais são implementáveis em um sistema assíncrono através de SSL/TLS [Dierks and Allen 1999]. Cada servidor possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Todos os escritores também compartilham um par de chaves: as *chaves pública e privada de escrita*. Cada chave privada é conhecida apenas pelo seu próprio dono (ou donos no caso dos escritores), 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.

2.2. Dinamicidade e Propriedades do Sistema

Nesta seção definimos as terminologias básicas e as propriedades relevantes para os protocolos do QUINCUNX. Definimos um $update = \{+, -\} \times \Pi$, onde a tupla $\langle +, i\rangle_i$ ou $\langle -, i\rangle_i$ (assinada por i) indica que o servidor i requisitou um *join* (entrada) ou *leave* (saída) do sistema, respectivamente. Dizemos que um *update* foi proposto por i na reconfiguração r caso i execute a operação de *join/leave* antes de r iniciar e nenhuma reconfiguração anterior processou este *update*. Para um conjunto de *updates* u, o conjunto de remoções de u, denotado $u.remove$, é o conjunto $\{i \in \Pi\text{: } \langle -, i\rangle_i \in u\}$. Similarmente, o conjunto de adições de u, denotado $u.join$, é o conjunto $\{i \in \Pi\text{: } \langle +, i\rangle_i \in u\}$. O *membership* de u, denotado $u.members$, é o conjunto $u.join \setminus u.remove$. O *membership* de uma visão v é definido pela união de todos os *updates* computados em v. Um servidor i pertence a v sse $i \in v.members$ (usamos a notação $i \in v$).

Definição 1 (Visão atual no tempo *t*) *Em qualquer tempo t da execução, definimos $V(t)$ como sendo a visão mais atual que algum servidor correto $i \in V(t)$ instalou desde o início de sua execução até t.*

Para qualquer tempo t, os protocolos de reconfiguração do QUINCUNX garantem que pelo menos um quórum de servidores da(s) visão(ões) anterior(es) conhece(m) $V(t)$. Desta maneira , $V(t)$ será a única visão onde clientes podem executar operações de leitura e escrita no tempo t. $V(t)$ permanece *ativa* desde o momento em que foi instalada por i até que uma nova reconfiguração aconteça, de tal maneira que todos os servidores corretos de uma outra visão mais atualizada instalem esta nova visão. Consideramos uma visão inicial $V(0)$ não vazia, que é inicialmente conhecida por todos os processos do sistema no tempo t_0. Também definimos $U(t)$ como sendo o conjunto de *updates pendentes* no tempo t, onde para cada $i \in \Pi$ tal que $\langle +, i\rangle_i \in U(t).join$ temos que $i \notin V(t).join$ e para cada $i \in \Pi$ tal que $\langle -, i\rangle_i \in U(t).remove$ temos que $i \notin V(t).remove$ (i.e., o *update* não foi processado até $V(t)$). Dadas estas definições, as seguintes hipóteses devem ser atendidas para o correto funcionamento do QUINCUNX.

Suposição 1 (Gentle leaves) *Um servidor correto i que requisita um leave no tempo t ($i \in V(t).members \wedge i \in U(t).remove$) permanece no sistema até terminar a instalação de uma visão atualizada na qual i não pertença.*

Esta suposição garante a presença de pelo menos um quórum de servidores corretos em uma dada visão ativa do sistema, garantindo o término das operações e a vivacidade do mesmo.

Suposição 2 (Finite reconfigurations) *O número de updates (join ou leave) requisitados em uma execução é finito.*

Esta suposição permite que as operações de leitura e escrita de clientes, bem como as operações de *join* e *leave* dos servidores, possam terminar apesar das falhas (propriedade de *wait-free*). Para não acessar dados obsoletos, estas operações devem ser executadas na visão mais atual do sistema. Assim, caso um processo muito lento não consiga completar sua requisição antes que a visão do sistema seja atualizada, tal processo deve reiniciar esta requisição utilizando a visão já atualizada. De fato, somente o número de *updates* requisitados concorrentemente com cada operação precisa ser finito. Outros protocolos, como o *DynaStore* [Aguilera et al. 2011], também consideram que requisições de *updates* são finitas. Na verdade, todas as propostas para sistemas de quóruns dinâmicos [Lynch and Shvartsman 2002, Martin and Alvisi 2004, Rodrigues and Liskov 2004] necessitam reiniciar operações devido à reconfigurações concorrentes e, então, devem fazer uso desta suposição para fornecer operações livre de espera (*wait-free*).

Suposição 3 (Fault threshold) *No máximo* $\lfloor \frac{|V(t).members|-1}{3} \rfloor$ *servidores de* $V(t)$ *podem falhar.*

Esta suposição restringe o número máximo de falhas de servidores que podem ocorrer em cada visão do sistema, representando a resiliência ótima para sistemas de quóruns bizantinos de disseminação [Malkhi and Reiter 1998b]: este limite é uma generalização da usual equação $n \geq 3f + 1$, sendo n a cardinalidade do sistema e f o número máximo de falhas. Em qualquer tempo t, somente servidores em $V(t)$ podem enviar respostas para as operações de leitura e escrita executadas por clientes. Quando um servidor i requisita um *join* no sistema, estas operações permanecem desabilitadas em i até a ocorrência de um evento *enable operations*. Após isso, i permanecerá apto para responder estas operações até que i solicite o *leave* do sistema, o que acontece com a ocorrência do evento *disable operations*.

Definição 2 (Propriedades do QUINCUNX) *As seguintes propriedades definem e devem ser garantidas pelo* QUINCUNX*:*

- ***Segurança 1 (Armazenamento)****: Os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico [Lamport 1986].*
- ***Segurança 2 (Reconfiguração – Join)****: Se um servidor correto* $i \notin V(t)$ *executa a operação de* $join$ *no tempo* t*, então existe um tempo* $t' > t$ *tal que* $i \in V(t')$*.*
- ***Segurança 3 (Reconfiguração – Leave)****: Se um servidor correto* $i \in V(t)$ *executa a operação de* $leave$ *no tempo* t*, então existe um tempo* $t' > t$ *tal que* $i \notin V(t')$*.*
- ***Terminação 1 (Armazenamento)****: Todas operações de leitura e escrita executadas por clientes corretos terminam.*
- ***Terminação 2 (Reconfiguração – Join)****: O evento enable operations terminará por ocorrer em todo servidor correto que executa a operação de* $join$*.*
- ***Terminação 3 (Reconfiguração – Leave)****: O evento disable operations terminará por ocorrer em todo servidor correto que executa a operação de* $leave$*.*

2.3. Visões: Estruturas e Operações

Cada visão v é uma tupla $\langle ov, entries, \mathtt{P} \rangle$, onde: ov é a visão que gerou v, i.e., na reconfiguração de ov a visão v é gerada pelos servidores em ov; $entries$ é um conjunto de $updates$ (Seção 2.2) e define o *membership* de v, como veremos a seguir; e $\mathtt{P}$ é um certificado contendo as provas que garantem tanto a integridade de v como também que v foi instalada no sistema. O campo $entries$ é incremental: para cada visão v, temos que $v.entries = v.ov.entries \cup updates$, onde $updates$ é um conjunto de atualizações válidas usadas para reconfigurar o sistema (Seção 2.2). Como mencionado, $v.entries$ define o *membership* de v: definimos $v.members = \{i \in \Pi : \langle +, i \rangle_i \in v.entries \wedge \langle -, i \rangle_i \notin v.entries\}$ (usamos a notação $i \in v.members$ ou $i \in v$).

Note que um servidor pode entrar e sair do sistema apenas uma única vez, mas podemos relaxar esta condição na prática adicionando números de época em cada requisição de reconfiguração.

***Bootstrapping*.** Na inicialização do sistema, cada servidor ativo $i \in V(0)$ recebe a visão inicial $v_0 = \langle \perp, u_0, \emptyset \rangle$, onde $u_0 = \{\langle +, j \rangle_j : j \in V(0)\}$. $V(0)$ é conhecida por todos os processos do sistema, não sendo necessárias provas para garantir sua integridade.

Limites de falhas e tamanho dos quóruns. Para cada visão v, denotamos $v.f$ como sendo o número de falhas toleradas em v. Seguindo a Suposição 3, $v.f = \lfloor \frac{|v.members|-1}{3} \rfloor$. Nossos protocolos utilizam quóruns de tamanho $v.q = \lceil \frac{|v.members|+v.f+1}{2} \rceil$ [Malkhi and Reiter 1998b].

Comparando e validando visões. Comparamos duas visões v_1 e v_2 através de seus campos $entries$. Usamos a notação $v_1 \subset v_2$ e $v_1 = v_2$ 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. Uma visão é válida caso possua as provas de que foi gerada e instalada no sistema (Seção 3.2.2).

3. Protocolos de Atualização de Visões

Esta seção apresenta os protocolos para reconfiguração do sistema, que são divididos em duas partes: protocolos para geração de uma nova visão e protocolos para sua instalação no sistema.

3.1. Geradores de Visões

Geradores de visões são abstrações utilizadas pelos servidores do sistema na obtenção de novas visões. Para cada visão instalada no sistema v, existe um gerador de visões associado, o qual gerará uma visão atualizada w. Um gerador de visões é acessado através de duas primitivas: (1) $generate\_view(v, updates)$, usada pelos servidores de v para iniciar a geração de uma nova visão contendo as atualizações contidas em $updates$; e (2) $new\_view(v, w)$, função usada pelo gerador de visões para informar aos processos que a nova visão atualizada w foi gerada.

Definição 3 (GERADORES DE VISÕES) *Os geradores de visões são definidos pelas seguintes propriedades:*

- ***Exatidão**: existem duas variantes desta propriedade:*
  - ***Exatidão forte**: o gerador de visões gera uma única visão w em todos os processos corretos.*
  - ***Exatidão fraca**: o gerador de visões pode gerar diferentes visões em diferentes processos. No entanto, para qualquer duas visões w e w' geradas, temos que $w \subset w' \vee w' \subset w$.*
- ***Terminação**: após inicializado, o gerador de visões acabará por gerar uma nova visão.*
- ***Não trivialidade**: para qualquer visão w, gerada pelo gerador de visões associado a um visão v, temos que $v \subset w$.*

As propriedades de exatidão estão relacionadas com a quantidade de visões geradas (segurança na geração de visões), enquanto que a propriedade de terminação abstrai a possibilidade de um gerador de visões não terminar. A combinação destas propriedades define quatro classes de geradores de visões, apresentadas na Tabela 1. A propriedade de não trivialidade deve ser satisfeita por qualquer gerador, garantindo que as visões geradas são sempre mais atuais.

Geradores das classes $\mathcal{P}$ e $\mathcal{S}$ são considerados seguros, pois os mesmos geram uma única visão (propriedade de exatidão forte). Já os geradores das classes $\mathcal{P}$ e $\mathcal{L}$ são considerados vivos, pois os mesmos sempre geram alguma visão. Os geradores da classe $\mathcal{W}$ são considerados fracos pois podem não terminar (gerar uma visão) ou ainda gerar várias visões. Nesse artigo, apresentamos protocolos para implementação da classe $\mathcal{L}$, com respectiva reconfiguração do QUINCUNX. Em [Alchieri 2011], o leitor encontrará os protocolos que implementam as demais classes (geradores $\mathcal{P}$, $\mathcal{S}$), além dos respectivos protocolos para reconfiguração do sistema.

Tabela 1. Classes de geradores de visões.

Terminação	Exatidão	
	Forte	Fraca
Sim	*Perfeito* $\mathcal{P}$	*Vivo* $\mathcal{L}$
Não	*Seguro* $\mathcal{S}$	*Fraco* $\mathcal{W}$

3.1.1. Gerador de Visões Vivo

O gerador de visões vivo garante que, após ser inicializado, uma nova visão acabará sendo gerada. No entanto, este gerador não garante que a mesma visão será gerada em todos os processos. O Algoritmo 1 apresenta uma implementação desse gerador para sistemas assíncronos.

Algoritmo 1 Gerador de visões vivo (servidor $i \in v$)

variables: {Sets and arrays used in the protocol}
PROP_PR - proofs for the new view entries proposal
NVENTR, NVENTR_PR - expected next view entries and its corresponding proofs

procedure $generate\_view(v, updates)$
1: obtain signed tuples $\langle updates, v, i\rangle_*$ from $v.q$ different servers in v and store it on $\texttt{PROP\_PR}^v$
2: $\texttt{NVENTR}^v \leftarrow v.entries \cup updates$
3: $\forall j \in v, send\langle$NEW-VIEW, $\langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP\_PR}^v\rangle$ to j

Upon receipt of $\langle$NEW-VIEW, $\langle v, entries, \emptyset\rangle_j, proof\rangle$ from j
4: **if** $isValidViewProposal(\langle v, entries, \emptyset\rangle_j, proof) \wedge (i, j \in v)$ **then**
5: **wait until** $\texttt{NVENTR}^v \neq \perp$
6: **if** $(\texttt{NVENTR}^v \neq entries) \wedge (entries \not\subset \texttt{NVENTR}^v)$ **then**
7: $\texttt{NVENTR}^v \leftarrow \texttt{NVENTR}^v \cup entries$
8: $\texttt{PROP\_PR}^v \leftarrow \texttt{PROP\_PR}^v \cup \{proofs\}$
9: $\texttt{NVENTR\_PR}^v \leftarrow \emptyset$
10: $\forall k \in v, send\langle$NEW-VIEW, $\langle v, \texttt{NVENTR}^v, \emptyset\rangle_i, \texttt{PROP\_PR}^v\rangle$ to k
11: **end if**
12: **if** $\texttt{NVENTR}^v = entries$ **then**
13: $\texttt{NVENTR\_PR}^v \leftarrow \texttt{NVENTR\_PR}^v \cup \{\langle v, entries, \emptyset\rangle_j\}$
14: **if** $(|\texttt{NVENTR\_PR}^v| \geq v.q)$ **then**
15: $new\_view(v, \langle v, entries, \texttt{NVENTR\_PR}^v\rangle)$
16: **end if**
17: **end if**
18: **end if**

Function $isValidViewProposal(\langle v, entries, \emptyset\rangle_j, proof)$ verifies if:
a) $(\forall e \in entries) \rightarrow (\exists z\colon \#_{\langle u_z, v, z\rangle_*} proof \geq v.q \wedge e \in u_z) \vee (e \in v.entries)$; and
b) $\#_{\langle u_j, v, j\rangle_*} proof \geq v.q \wedge \forall e \in u_j \rightarrow e \in v.entries$;

O Algoritmo 1 requer 3 passos de comunicação para gerar uma visão. No entanto, pode ser necessário executar mais de uma vez o passo crucial NEW-VIEW. A seguir é descrito como um gerador de visões vivo, associado a uma visão v, gera novas visões para atualização do sistema. Cada servidor correto $i \in v$ certifica sua proposta e envia uma mensagem NEW-VIEW para todos os servidores de v. Os servidores em v atualizam suas propostas caso recebam propostas diferentes (linhas 6-11) e convergem para uma nova visão w assim que os mesmos receberem um quórum de propostas para w (linhas 12-17). Como sempre temos pelo menos um quórum de servidores corretos em v, é garantido que cada servidor $i \in v$ sempre acabará gerando pelo menos uma visão. Vale ressaltar que servidores poderão gerar visões diferentes. No entanto, através das propriedades de intersecção dos quóruns, é garantido que existe uma relação entre qualquer duas visões w e w' geradas por este gerador: $w \subset w'$ ou $w' \subset w$. Assim, estas visões podem ser organizadas em uma sequência, de acordo com o operador "$\subset$".

A Figura 1 apresenta a sequência de visões que pode ser gerada. Como é necessário um quórum de propostas para convergir para uma visão, esta sequência é limitada e conterá no máximo $|v.members| - v.q + 1$ visões: a primeira visão obtida pela computação dos *updates*

de um quórum de servidores ($v.q$) e $|v.members| - v.q$ outras visões obtidas pela adição dos *updates* de cada um dos outros servidores restantes, que devem possuir diferentes propostas para nova visão. As provas de corretude deste algoritmo podem ser encontradas em [Alchieri 2011].

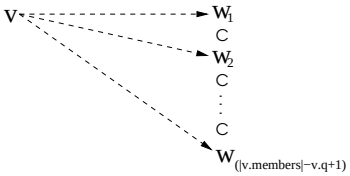


Figura 1. Sequência de visões geradas pelo gerador associado a uma visão v.

3.2. Reconfiguração do Sistema

Esta seção apresenta protocolos para atualizar o conjunto de servidores que implementa o sistema de quóruns, através da utilização do gerador de visões. Estes protocolos não são executados cada vez que um processo solicita um *join*/*leave* do sistema, mas periodicamente quando o tempo de duração de determinada visão se esgotar e existirem requisições de reconfiguração pendentes. Esta abordagem diminui as necessidades de processamento ao executar estas requisições em lotes dentro de uma única atualização de visão. Além disso, esta solução desacopla os protocolos de reconfiguração dos protocolos para acesso (leitura e escrita) ao sistema: as reconfigurações do QUINCUNX podem ser executadas em paralelo com os protocolos de leitura e escrita no registrador, melhorando o desempenho do sistema quando comparado com soluções onde estes protocolos são fortemente acoplados (ex.: [Aguilera et al. 2011]).

3.2.1. Iniciando o Gerador de Visões

Esta seção discute os procedimentos que fazem com que todos os servidores da visão corrente do sistema cv iniciem o gerador de visões associado a esta visão, a fim de reconfigurar o sistema. Este protocolo, apresentado no Algoritmo 2, é utilizado para iniciar qualquer gerador de visões, sendo que apenas os procedimentos após as visões serem geradas são diferentes e dependem do gerador utilizado. Primeiramente, processos devem requisitar sua entrada ou saída do sistema, enquanto cv estiver ativa. Após o tempo de duração de cv se esgotar, o seu gerador de visões associado é iniciado para que novas visões sejam geradas e o sistema reconfigurado.

Requisitando a entrada e/ou saída do sistema. Caso um servidor j desejar entrar no sistema, o mesmo deve encontrar uma visão com algum(s) membro(s) em comum com a visão corrente do sistema cv e executar a operação de $join$ (linas 1-2), onde j envia a tupla assinada $\langle +, j \rangle_j$ indicando que deseja entrar no sistema. Por outro lado, para sair do sistema, j executa a operação de $leave$ (linhas 3-4) e envia a tupla assinada $\langle -, j \rangle_j$. Quando um servidor i receber estas mensagens, i verifica se ambos os servidores (j e i) estão usando a mesma visão mais atual. Caso a visão de j esteja desatualizada, i envia a visão corrente para j, o qual executa novamente sua operação utilizando a visão atualizada (por simplicidade, este processamento não aparece no algoritmo). Do contrário, e caso j ainda não executou a operação de $join$ ou $leave$, i adiciona esta requisição de atualização em seu conjunto de *updates* (`RECV`), para ser processada na próxima reconfiguração do sistema, e envia uma mensagem para j confirmando que a requisição foi recebida (linhas 5-6).

Iniciando o gerador de visões. Cada servidor $i \in cv$ inicia uma reconfiguração para cv através do envio de uma mensagem START-RECONF (linha 7), o que acontece quando o tempo de

duração de cv se esgotar e i possuir *updates* para propor (caso contrário o *timeout* para cv é reiniciado) ou quando i receber mais do que $cv.f$ destas mensagens de outros servidores em cv (linha 9), garantindo que o tempo de duração de cv se esgotou em pelo menos um servidor correto. Assim, servidores maliciosos não são capazes de iniciar uma reconfiguração prematuramente. Cada mensagem START-RECONF contêm a visão corrente do emissor e o conjunto de *updates* proposto para ser aplicado nesta visão. Quando um servidor i recebe estas mensagens, i adiciona os *updates* recebidos em seu conjunto `RECV` (linha 8), a fim de realizar uma proposta com todos os *updates* conhecidos e, principalmente, para garantir que i iniciará a reconfiguração quando o próximo *timeout* para cv se esgotar, pois $\texttt{RECV} \neq \emptyset$. Um servidor i espera por $cv.q$ destas mensagens para então iniciar seu gerador de visões (linha 10), propondo todos os *updates* conhecidos até o momento para atualização de cv. Desta forma, todos os servidores corretos de cv iniciarão o gerador de visões associado a esta visão.

Algoritmo 2 Iniciando o gerador de visões (servidor *i*)

variables: {Sets and view generator object used in the protocol}
`RECV` - set of received updates
`VIEW_GEN` - the view generator

procedure $join(v)$
1: $\forall j \in v, send\langle$RECONFIG, $\langle +, i\rangle_i, v\rangle$ to j
2: **wait** for $v.q$ ⟨REC-CONFIRM⟩ replies from different servers in v

procedure $leave()$
3: $\forall j \in cv, send\langle$RECONFIG, $\langle -, i\rangle_i, cv\rangle$ to j
4: **wait** for $cv.q$ ⟨REC-CONFIRM⟩ replies from different servers in cv

Upon receipt of ⟨RECONFIG, $\langle *, j\rangle_j, cv\rangle$ from $j \wedge \langle *, j\rangle_j \notin cv$
5: $\texttt{RECV} \leftarrow \texttt{RECV} \cup \{\langle *, j\rangle_j\}$
6: $send\langle$REC-CONFIRM⟩ to j

Upon ((timeout for cv and $\texttt{RECV} \neq \emptyset$) ∨ (start)) ∧ (not started reconfig. for cv)
7: $\forall j \in cv, send\langle$START-RECONF, `RECV`, $cv\rangle$ to j

Upon receipt of ⟨START-RECONF, $recv, cv\rangle$ from $j \in cv \wedge (\forall u \in recv \rightarrow u \notin cv)$
8: $\texttt{RECV} \leftarrow \texttt{RECV} \cup recv$
9: **if** (received START-RECONF from $cv.f + 1$ servers) **then** start $\leftarrow true$ **end if**

Upon receipt of ⟨START-RECONF, $recv, cv\rangle$ from $cv.q$ different servers in cv
10: $\texttt{VIEW\_GEN}.generate\_view(cv, \texttt{RECV})$

3.2.2. Reconfiguração com Geradores de Visões Vivos

Esta seção apresenta o protocolo para reconfiguração do sistema através de um gerador de visões vivo $\mathcal{L}$. Apesar da possibilidade de várias visões serem geradas por este gerador, apenas algumas delas são instaladas no sistema, formando uma sequência de visões instaladas. Os clientes apenas poderão executar operações nas visões que formam esta sequência, o que garante as propriedades tanto de terminação quanto de segurança do QUINCUNX.

A principal ideia deste protocolo é fazer com que os servidores organizem as várias visões geradas em sequências. O ponto fundamental para entender o funcionamento deste protocolo é a separação do algoritmo que gera visões atualizadas (encapsulado nos geradores de visões) do algoritmo que organiza estas visões em sequências (apresentado nesta seção). Deste modo, cada visão v estabelece uma sequência de visões para sua atualização e encaminha estes dados para que a visão seguinte na sequência w seja instalada. Após isto, os servidores de w utilizam os dados recebidos de v e definem uma nova sequência de visões para atualização de w, e assim por diante até que existam visões mais atuais para serem instaladas no sistema. Como não utiliza consenso, mais de uma sequência seq_1 e seq_2 de atualização podem ser definidas por uma visão v. No entanto, através da propriedade de intersecção dos quóruns é possível garantir que uma sequência sempre estará contida em outra definida posteriormente, i.e., $seq_1 \subset seq_2$.

Por exemplo, seq_1 pode ser $w_1 \rightarrow w_2$ e então seq_2 é composta por $w_1 \rightarrow w_2 \rightarrow w_3$ (ou $w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3$, ou $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4$, etc...). Note que todas estas sequências podem ser agrupadas em uma única sequência que engloba todas as visões geradas por determinado gerador, uma vez que as mesmas possuem uma relação que permite esta estruturação. Dada uma sequência seq definida para atualização de uma visão v, apenas a última visão w de seq é efetivamente instalada no sistema (em nosso exemplo: seq_1 instalará w_2 e seq_2 instalará w_3). As visões restantes apenas são utilizadas no processo de instalação de w, pois como é possível que uma sequência anterior definida para atualização de v tenha instalado uma visão $w' \in seq$, o valor do registrador dos servidores em w deve ser atualizado através de uma leitura "realizada em cadeia" nesta sequência (cada visão lê o valor armazenado na visão imediatamente anterior na sequência). Em nosso exemplo, quando da instalação de w_3 (instalada pela seq_2) uma leitura deve ser realizada em w_2, que lê de w_1, que lê de v. Note que w_2 pode ter sido instalada anteriormente através de seq_1, neste caso qualquer valor mais atual escrito em w_2 será enviado para w_3, não sendo perdido durante a atualização da visão do sistema.

As visões que não são instaladas no sistema, chamadas de *visões auxiliares*, não possuem um gerador de visões associado, pois não é necessário atualizá-las (estas visões já se encontram no meio de alguma sequência de visões, i.e., já existe uma visão mais atual para ser instalada no sistema). Desta forma, apenas visões instaladas possuem geradores de visões para geração de novas visões. Neste sentido, um ponto fundamental para o funcionamento correto deste algoritmo é que cada visão efetivamente instalada no sistema pode mudar uma sequência previamente definida para sua atualização, definindo uma nova sequência que "mistura" visões de sequências previamente definidas com visões geradas por seu gerador de visões associado. Também é possível que novas visões sejam criadas através da união de visões previamente definidas com visões geradas pelo seu gerador. Com isto, é possível resolver impasses e organizar todas as visões em uma única sequência. Em nosso exemplo, a visão w_2 poderia iniciar uma reconfiguração antes de receber seq_2 (que faria o sistema atualizar para w_3). Neste caso, o gerador associado à w_2 poderia gerar uma visão w_{2_1} e não seria possível definir se w_2 atualizaria para w_{2_1} (por meio de seu gerador) ou w_3 (por meio de seq_2). Pior ainda, alguns servidores de w_2 poderiam receber primeiro w_{2_1} e tentar atualizar o sistema para esta visão, enquanto que outros servidores poderiam receber primeiro w_3 (seq_2) e tentar instalar esta visão, causando uma divisão de modo que nenhuma das visões pudesse ser instalada. Nossa abordagem resolve este problema fazendo com que w_2 defina uma nova sequência de atualização do sistema, a qual poderá conter tanto a visão w_3 quando a nova visão w_{2_1} ou ainda uma visão representando a união de ambas $w_3 \cup w_{2_1}$, mantendo uma sequência única de visões, como veremos adiante.

Neste protocolo (Algoritmos 3 e 4), para uma visão v ser válida é necessário que: *(1)* uma sequência contendo v foi gerada (Algoritmo 3) – $v.P$ contém um quórum de assinaturas seq_pr^w geradas por servidores de w, para uma sequência seq^w, e $v \in seq^w$ (sequência contendo v gerada pelos servidores da visão w). Então, deve-se também verificar se w é válida seguindo estes mesmos critérios; e *(2)* v foi instalada (Algoritmo 4) – $v.P$ contém um quórum de tuplas $\langle \text{INSTALLED}, w, v \rangle_*$ assinadas por servidores de w.

Este protocolo funciona da seguinte forma. Quando o gerador de visões reporta a geração de uma nova visão w para um servidor $i \in cv$, o mesmo verifica se já fez qualquer proposta para sequência de atualização de cv (linha 1), para então propor uma sequência contendo apenas w (linhas 2-3). Esta visão w será a única visão proposta por i, mas diferentes servidores podem propor diferentes visões (propriedade de exatidão fraca do gerador de visões vivo) e as mesmas devem ser organizadas em uma sequência.

A forma como os servidores convergem para sequências é semelhante com a forma de

como os mesmos convergem para visões no gerador de visões vivo (Seção 3.1.1). Deste modo, quando um servidor $i \in cv$ receber uma proposta para sequência de visões de outro servidor j, o mesmo verifica se todas as visões propostas por j são válidas, i.e., foram geradas pelo gerador de visões (linha 15 do algoritmo 1) ou durante a organização de outras visões válidas (linha 15 do algoritmo 3). Então, dois casos (não mutualmente exclusivos) são possíveis:

Algoritmo 3 Reconfigurando o sistema com geradores vivos (servidor *i*)

variables: {Sets and arrays used in the protocol}
SEQ, SEQ_PR - proposed sequence of views and its corresponding proofs
LCSEQ, LCSEQ_PR - last converged sequence of views known and its corresponding proofs
VIEW_GEN - the view generator (from Algorithm 2)

Upon VIEW_GEN.$new\_view(cv, w)$
1: **if** $\texttt{SEQ}^{cv} = \emptyset$ **then** {$w.ov = cv$ (line 15 of Alg. 1}
2: $\texttt{SEQ}^{cv} \leftarrow \{w\}$
3: $\forall j \in cv, send\langle$SEQ-VIEW, $\texttt{SEQ}_i^{cv}, \emptyset\rangle$ to j
4: **end if**

Upon receipt of $\langle$SEQ-VIEW, $seq_j^{cv}, \langle lcseq, proof\rangle\rangle$ from j {i only executes this message if it remains in cv }
5: **if** $isValidSequenceProposal(seq_j^{cv}, \langle lcseq, proof\rangle) \wedge (i, j \in cv)$ **then**
6: **if** $\exists v \in seq_j^{cv}$: $v \notin \texttt{SEQ}^{cv}$ **then**
7: **if** $\exists v, w$: $v \in seq_j^{cv} \wedge w \in \texttt{SEQ}^{cv} \wedge v \not\subset w \wedge w \not\subset v$ **then** {v and w was generated by different reconfig.}
8: $v \leftarrow w$: $(w \in \texttt{LCSEQ}^{cv}) \wedge (\nexists w' \in \texttt{LCSEQ}^{cv}$: $w \subset w')$ {the most updated view in $\texttt{LCSEQ}^{cv}$}
9: $v' \leftarrow w$: $(w \in lcseq) \wedge (\nexists w' \in lcseq$: $w \subset w')$ {the most updated view in $lcseq$}
10: **if** $v \subset v'$ **then** {$lcseq$ is up to date than $\texttt{LCSEQ}^{cv}$}
11: $\langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ\_PR}^{cv}\rangle \leftarrow \langle lcseq, proof\rangle$
12: **end if**
13: $v \leftarrow w$: $(w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv}$: $w \subset w')$ {the most updated view in $\texttt{SEQ}^{cv}$}
14: $v' \leftarrow w$: $(w \in seq_j^{cv}) \wedge (\nexists w' \in seq_j^{cv}$: $w \subset w')$ {the most updated view in seq_j^{cv}}
15: $\texttt{SEQ}^{cv} \leftarrow \texttt{LCSEQ}^{cv} \cup \{\langle cv, v.entries \cup v'.entries, \langle v, v'\rangle\rangle\}$
16: **else**
17: $\texttt{SEQ}^{cv} \leftarrow \texttt{SEQ}^{cv} \cup seq^{cv}$
18: **end if**
19: $\texttt{SEQ\_PR}^{cv} \leftarrow \emptyset$
20: $\forall k \in cv, send\langle$SEQ-VIEW, $\texttt{SEQ}_i^{cv}, \langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ\_PR}^{cv}\rangle\rangle$ to k
21: **end if**
22: **if** $\forall v \in seq_j^{cv} \leftrightarrow v \in \texttt{SEQ}^{cv}$ **then**
23: $\texttt{SEQ\_PR}^{cv} \leftarrow \texttt{SEQ\_PR}^{cv} \cup seq_j^{cv}$
24: **if** $(|\texttt{SEQ\_PR}^{cv}| \geq cv.q)$ **then**
25: $\langle \texttt{LCSEQ}^{cv}, \texttt{LCSEQ\_PR}^{cv}\rangle \leftarrow \langle \texttt{SEQ}^{cv}, \texttt{SEQ\_PR}^{cv}\rangle$
26: $v \leftarrow w$: $(w \in \texttt{SEQ}^{cv}) \wedge (\nexists w' \in \texttt{SEQ}^{cv}$: $w' \subset w)$ {the next view in the sequence $\texttt{SEQ}^{cv}$}
27: $\forall k \in cv \cup v, send\langle$INSTALL-SEQ, $v, \texttt{SEQ}^{cv}, \texttt{SEQ\_PR}^{cv}\rangle$ to k
28: **end if**
29: **end if**
30: **end if**

Upon receipt of $\langle$INSTALL-SEQ, $v, seq^{ov}, seq\_pr^{ov}\rangle$ from $ov.f + 1$ different servers in ov and $i \in ov$
31: $\forall k \in ov \cup v, send\langle$INSTALL-SEQ, $v, seq^{ov}, seq\_pr^{ov}\rangle$ to k
Upon receipt of $\langle$INSTALL-SEQ, $v, seq^{ov}, seq\_pr^{ov}\rangle$ from $ov.q$ different servers in ov and $i \in \{ov \cup v\}$
32: **reconfigure**$(v, seq^{ov}, seq\_pr^{ov})$

Function $isValidSequenceProposal(seq_j^{cv}, \langle lcseq, proof\rangle)$ verifies if:
a) $\forall w \in seq_j^{cv}$ we have that $cv \subset w$ and all views in seq_j^{cv} form a sequence according with the $\subset$ operator;
b) $\forall w \in seq_j^{cv}$ we have one of the following:
1 $w.P$ is a set of $\langle w.ov, w.entries, \emptyset\rangle_*$ tuples and: $\#_{\langle w.ov, w.entries, \emptyset\rangle_*} w.P \geq w.ov.q$
2 $w.P$ is a tuple $\langle v, v'\rangle$ and: *(i)* v and v' satisfy the previous condition (case 1) and *(ii)* $((w.entries = v.entries \cup v'.entries) \wedge (v.ov \neq v'.ov))$.
c) $\#_{lcseq_*} proof \geq cv.q$.

(1) i ainda não fez proposta ou a mesma é diferente da proposta de j, onde alguma visão proposta por j ainda não foi proposta por i (linhas 6-21): Então, dois cenários são possíveis:

• As visões já propostas por i e as novas visões recebidas não formam uma sequência (linhas 8-15): isso pode acontecer quando visões geradas por diferentes geradores devem ser organizadas em uma sequência. Então, i define a mais atual sequência de visões já convergida (linhas 8-12) e determina sua proposta de sequência adicionando uma visão que representa a união das duas sequências (a já proposta por i e a recebida de j - linhas 13-15). Isto resolve o

conflito devido à necessidade de organizar sequência de visões geradas por diferentes geradores.

• As visões já propostas por i e as novas visões recebidas formam uma sequência (linha 17): então, a nova sequência a ser proposta por i é formada por todas estas visões conhecidas.

(2) i fez a mesma proposta de j (linhas 22-29): neste caso i coleta a proposta assinada. Quando i obtém a prova de que um quórum fez esta proposta, i converge para uma sequência de visões e envia uma mensagem para que a visão seguinte na sequência v seja inicializada.

Algoritmo 4 Reconfigurando o sistema com geradores vivos (servidor *i*), continuação...

variables: {Sets and arrays used in the protocol}
RECV - set of received updates (from Algorithm 2)
SEQ - proposed sequence of views (from Algorithm 3)

procedure $reconfigure(v, seq^{ov}, seq\_pr^{ov})$
1: $v.P \leftarrow seq\_pr^{ov}$ {proofs that v is in the sequence}
2: **if** $i \notin v$ **then** {i is leaving the system}
3: *disable operations* {stops accepting R/W client requests}
4: **end if**
5: **if** $cv \subset v$ **then** {i updates its current view to v }
6: $cv \leftarrow v$
7: **end if**
8: RECV $\leftarrow$ RECV $\setminus\ v.entries$
9: **if** $i \in ov$ **then** {i is member of the previous view in the sequence}
10: **if** $|seq^{ov} \setminus v = \emptyset|$ **then** {v is the last view in the sequence}
11: $\forall j \in v, send\langle$STATE-UPDATE, $data, ts, \langle$INSTALLED, $ov, v\rangle_i$, RECV$\rangle$ to j
12: **else**
13: $\forall j \in v, send\langle$STATE-UPDATE, $data, ts, \perp$, RECV$\rangle$ to j
14: **end if**
15: **end if**
16: **if** $i \in v$ **then** {i is member of the updated view}
17: **wait** for $ov.q$ $\langle$STATE-UPDATE, $*, *, *, *\rangle$ for different servers in ov
18: **if** $(|\{seq^{ov} \setminus v\}| = \emptyset) \wedge cv = v$ **then** {v is the last view in the sequence}
19: $cv.P \leftarrow cv.P \cup$ the $ov.q$ $\langle$INSTALLED, $ov, v\rangle_*$ received {proofs that v was installed}
20: **end if**
21: $\langle data, ts\rangle \leftarrow$ read from the quorum of STATE-UPDATE messages received
22: RECV $\leftarrow$ RECV $\cup$ {all valid updates requests collected from STATE-UPDATE messages} $\setminus\ cv.entries$
23: **if** $i \notin ov$ **then** {i is entering on the system}
24: *enable operations* {starts to accept R/W client requests}
25: **end if**
26: $\forall j \in ov$: $j \notin v, send\langle$VIEW-INSTALLED, $v\rangle$ to j
27: **if** $(\text{SEQ}^v = \emptyset) \wedge (\{seq^{ov} \setminus v\} \neq \emptyset)$ **then**
28: $\text{SEQ}^v \leftarrow \{seq^{ov} \setminus v\}$
29: $\forall k \in v, send\langle$SEQ-VIEW, $\text{SEQ}_i^v, \emptyset\rangle$ to k
30: **else if** $\text{SEQ}^v = \emptyset$ **then** {v is the last view in the sequence seq^{ov}}
31: start timer for cv {if updated on line 6}
32: **end if**
33: **else** {i is leaving the system}
34: **wait** for $v.f + 1$ $\langle$VIEW-INSTALLED, $v\rangle$ from different servers in v
35: discard keys and halt
36: **end if**

Os servidores aguardam por um quórum de INSTALL-SEQ para então "instalar" (lembrando que v apenas será instalada caso não exista uma visão seguinte em seq) a visão seguinte v de uma sequência seq (linha 32). Pela propriedade de intersecção dos quóruns, qualquer sequência posteriormente obtida seq' conterá seq, de modo que qualquer visão instalada através de seq estará contida em seq' e será acessada na atualização do estado da visão instalada através de seq'. Para instalar v, o sistema é reconfigurado da seguinte forma (Algoritmo 4): *(1)* as provas de que v pertence a uma sequência (linha 1) e foi instalada (linhas 10-14,18-20) são coletadas; *(2)* caso exista alguma visão seguinte em seq, uma sequência para atualização de v é proposta (linhas 27-29); e *(3)* um servidor deixa o sistema somente após garantir que a visão mais atual, para a qual enviou o estado, foi instalada (ou teve seu estado atualizado caso seja uma visão auxiliar – mensagens VIEW-INSTALLED). Por exemplo, caso o algoritmo para atualização de cv produza $seq_1 : v_1 \rightarrow v_2$ e depois $seq_2 : v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$, alguns servido-

res de cv podem receber primeiro seq_1 e outros seq_2 de forma que alguns enviam as mensagens de atualização de estado antes para v_1 e outros para v_0. Caso saíssem do sistema após o primeiro envio desta mensagem, seria possível que nenhuma das visões v_0 ou v_1 recebesse um quórum destas mensagens, de modo que o sistema perderia vivacidade. As provas de corretude deste algoritmo podem ser encontradas em [Alchieri 2011].

Exemplo de funcionamento do protocolo. Imagine que na reconfiguração de uma visão v_0, o gerador de visões associado a esta visão gera as visões atualizadas v_1, v_2 e v_3, sendo que $v_1 \subset v_2 \subset v_3$. Após isso, os servidores em v_0 geram as sequências $seq_1^{v_0} : v_1 \rightarrow v_2$ e $seq_2^{v_0} : v_1 \rightarrow v_2 \rightarrow v_3$. Então, após v_1 ser inicializada (estado atualizado), a mesma pode gerar as sequências $seq_1^{v_1} : v_2$ e $seq_2^{v_1} : v_2 \rightarrow v_3$. Neste exemplo, v_1 é uma visão auxiliar enquanto que v_2 e v_3 poderão ser instaladas. Os servidores de v_2 que receberem $seq_2^{v_1}$ antes de $seq_1^{v_1}$ também não instalam v_2. Por outro lado, caso os servidores em v_2 receberem primeiro $seq_1^{v_1}$, então v_2 é instalada no sistema. Como estamos em um sistema assíncrono, é possível que os servidores de v_2 iniciem o gerador de visões gv para atualização de v_2 e só então recebam $seq_2^{v_1}$ para atualização de v_2 através da instalação de v_3. Neste caso, é possível que concorrentemente gv gere uma nova visão w_1 para atualização de v_2.

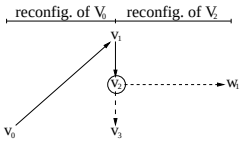
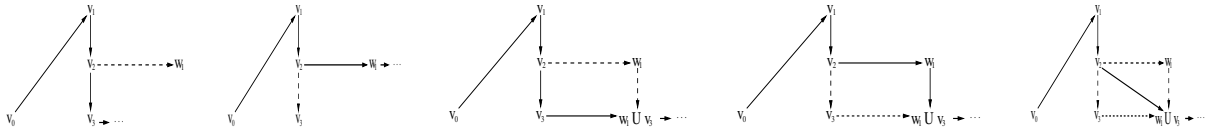


Figura 2. Servidores de v_2 em conflito.

Neste cenário, os servidores de v_2 podem "entrar em conflito" onde alguns recebem primeiro as mensagens para gerar a sequência com w_1 enquanto outros recebem primeiramente as mensagens para gerar a sequência com v_3, como mostra a Figura 2. Neste caso, tanto v_3 quanto w_1 devem ser organizadas em uma sequência a ser instalada a partir de v_2. Para isso, é possível que uma nova visão representando a união destas sequências ($w_1 \cup v_3$) seja gerada. No processamento das mensagens SEQ-VIEW, dependendo de quais mensagens compondo um quorum são entregues nos servidores, é possível que seja instalada apenas uma das duas visões (v_3 ou w_1), ou uma destas visões seguida pela união das mesmas ($w_1 \cup v_3$), ou ainda apenas a união de ambas (Figura 3). Porém, como é necessário um quórum de mensagens para definir uma nova sequência, nunca teremos a instalação de ambas, w_1 e v_3, fazendo com que a sequência de visões instaladas não seja quebrada.



(a) $v_2 \rightarrow v_3$ (b) $v_2 \rightarrow w_1$ (c) $v_2 \rightarrow v_3 \rightarrow w_1 \cup v_3$ (d) $v_2 \rightarrow w_1 \rightarrow w_1 \cup v_3$ (e) $v_2 \rightarrow w_1 \cup v_3$

Figura 3. Possíveis sequências de visões instaladas.

4. Discussão

DynaStore vs. QUINCUNX. No DynaStore, reconfigurações geram um grafo de visões e uma operação de leitura ou escrita precisa percorrer este grafo para encontrar a visão mais atual instalada no sistema, onde é segura a sua execução. Neste processamento, cada aresta deste grafo deve ser acessada para verificar se existe uma visão mais atual instalada no sistema. Para

cada aresta, é necessário acessar um *weak snapshot object*, que é implementado por um conjunto de registradores estáticos (implementados através de sistemas de quóruns). Em contraste, as reconfigurações do QUINCUNX instalam apenas uma sequência de visões, i.e., visões geradas por diferentes geradores são organizadas em uma sequência única.

Usando consenso. Um gerador da classe $\mathcal{P}$ pode ser implementado diretamente através de um protocolo de consenso: quando o gerador é inicializado, as atualizações são propostas em uma instância do consenso de modo que todos os processos decidem pelo mesmo conjunto de atualizações, as quais são aplicadas na visão corrente do sistema, gerando uma nova visão. Caso um gerador desta classe seja utilizado no protocolo de reconfiguração do QUINCUNX, todos geradores geram a mesma visão w no Algoritmo 3 e todos os servidores corretos propõem a mesma sequência $\{w\}$, fazendo o sistema reconfigurar diretamente de cv para w.

5. Conclusões

Este artigo apresentou protocolos de reconfiguração do QUINCUNX, um sistema de quóruns bizantino dinâmico, que implementa um registrador atômico de leitura e escrita. A reconfiguração utiliza um gerador de visões que termina e não se baseia no consenso, além de ser independente em relação aos protocolos de leitura e escrita. Desta maneira, é possível termos concorrência entre estes diversos protocolos, aumentando o desempenho do sistema 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. P. (2011). *Protocolos Tolerantes a Faltas Bizantinas para Sistemas Distribuídos Dinâmicos*. Tese de doutorado em engenharia de automação e sistemas, Universidade Federal de Santa Catarina.

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.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

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. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149.

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. (1986). On interprocess communication (part II). *Distributed Computing*, 1(1):203–213.

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. (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.

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.

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

Shraer, A., Martin, J.-P., Malkhi, D., and Keidar, I. (2010). Data-centric reconfiguration with network-attached disks. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*.