Proposta de Implementação de Memória Compartilhada Dinâmica Tolerante a Falhas

Mateus Braga[1] **e Eduardo Alchieri**[1]

[1]Departamento de Ciência de Computação
Universidade de Brasília
Brasília - Brasil

Abstract. *Quorum systems are useful tools for implementing consistent and available storage in the presence of failures. These systems usually comprise a static set of servers that provide a fault-tolerant read/write register accessed by a set of clients. This paper proposes an implementation for* FREESTORE*, a set of fault-tolerant protocols that emulates a register in dynamic systems in which processes are able to join/leave the servers set by reconfigurations. A set of experiments analyses the performance of the implementation and brings some light to the shared memory reconfiguration procedure.*

Resumo. *Sistemas de quóruns são úteis na implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas. Estes sistemas geralmente compreendem um conjunto estático de servidores que implementam um registrador acessado através de operações de leitura e escrita. Este artigo propõe uma implementação para o* FREESTORE*, um conjunto de protocolos tolerantes a falhas capazes de emular um registrador em sistemas dinâmicos, onde processos podem entrar e sair, durante sua execução, através de reconfigurações. Um conjunto detalhado de experimentos avalia o desempenho dos protocolos implementados e possibilita uma maior compreensão a respeito do processo de reconfiguração de memória compartilhada.*

1. Introdução

Sistemas de quóruns [Gifford 1979] são abstrações fundamentais usadas para garantir consistência e disponibilidade de dados 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 em 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.

Sistemas de quóruns foram inicialmente estudados em ambientes estáticos [Gifford 1979, Attiya et al. 1995, Malkhi and Reiter 1998], onde não é permitida a entrada e saída de servidores durante a execução do sistema. Mais recentemente, surgiram também propostas para sistemas de quóruns reconfiguráveis [Alchieri et al. 2014, Aguilera et al. 2011, Gilbert et al. 2010, Martin and Alvisi 2004], i.e., sistemas em que os protocolos permitem modificações no conjunto de servidores que implementam a memória compartilhada (registrador). O processo de modificar o conjunto de processos que compõem o sistema chama-se *reconfiguração* e faz com que o sistema se mova de uma configuração (visão) antiga para uma nova configuração atualizada.

Estas propostas para sistemas reconfiguráveis podem ser dividas em soluções que empregam consenso [Lamport 1998] na definição da nova visão a ser instalada no sistema de forma que todos os processos concordem com a nova visão [Alchieri et al. 2014,

Gilbert et al. 2010, Martin and Alvisi 2004] e soluções que não precisam desta primitiva de sincronização [Alchieri et al. 2014, Aguilera et al. 2011]. Dentre estas soluções destaca-se o FREESTORE [Alchieri et al. 2014] por ser um conjunto de protocolos modulares que implementam reconfigurações tanto usando consenso quanto sem usar esta primitiva. De fato, através da modularidade do FREESTORE é possível aglutinar todos os requisitos de sincronia necessários para a reconfiguração do sistema em um módulo específico, o qual pode ser implementado destas duas formas. Outra característica importante desta solução é que os protocolos de leitura e escrita (R/W) são desacoplados dos protocolos de reconfiguração, simplificando a adaptação de protocolos estáticos de R/W para ambientes dinâmicos.

Mesmo com todas estas propostas para um sistema de quóruns reconfigurável, ainda falta uma implementação para os mesmos que possa ser usada no desenvolvimento de aplicações distribuídas. Este trabalho visa preencher esta lacuna e apresenta uma implementação para os protocolos do FREESTORE. Este sistema foi escolhido por fornecer as duas formas de reconfiguração acima discutidas, possibilitando uma melhor análise e compreensão a respeito do processo de reconfiguração destes sistemas.

O restante deste artigo está organizado da seguinte maneira. A Seção 2 apresenta o modelo de sistema assumido para a implementação dos protocolos, enquanto que a Seção 3 apresenta uma visão geral sobre o FREESTORE. A Seção 4 apresenta a nossa proposta de implementação e a Seção 5 discute uma série de experimentos realizados com a implementação desenvolvida. Finalmente, a Seção 6 apresenta nossas conclusões.

2. Modelo de Sistema

Nossa implementação considera um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois subconjuntos distintos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$; e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. Clientes acessam o sistema de armazenamento implementado por um subconjunto dos servidores (uma *visão*) executando operações de leitura e escrita (Seção 4.1). Cada processo do sistema (cliente ou servidor) possui um identificador único (Seção 4.2) e está sujeito a falhas por parada (*crash*). A chegada dos processos no sistema segue o modelo de chegadas infinitas com concorrência desconhecida mas finita [Aguilera 2004]. Além disso, os processos estão conectados através de canais confiáveis, os quais são implementáveis através do uso do protocolo TCP/IP.

Os protocolos de reconfiguração do FREESTORE são implementáveis em um sistema distribuído assíncrono, onde não existem limites para o tempo de transmissão de mensagens ou processamentos locais nos processos[1]. Além disso, cada servidor do sistema tem acesso a um relógio local usado para iniciar reconfigurações (Seção 4.4).

Para cada visão v são permitidas no máximo f falhas de processos, sendo $f \leq \lfloor \frac{n-1}{2} \rfloor$ onde n é o número de processos presentes em v, com quóruns de tamanho $v.q = \lceil \frac{n+1}{2} \rceil$. Finalmente, um processo que deseja deixar o sistema deve aguardar até que uma visão sem sua presença seja instalada e o número de reconfigurações concorrentes com uma operação de R/W deve ser finito [Alchieri et al. 2014].

[1]Apesar de não ser a única alternativa, é possível utilizar um protocolo de consenso durante uma reconfiguração (Seção 3). Neste caso precisamos de um sistema parcialmente síncrono [Dwork et al. 1988].

3. Visão Geral do FREESTORE

O FREESTORE [Alchieri et al. 2014] compreende um conjunto de protocolos para a realização de reconfigurações em sistemas de quóruns. Sua organização é modular e apresenta uma desejável separação de conceitos diferenciando claramente os processos de gerar uma nova visão[2], de reconfiguração do sistema (i.e., instalar uma visão atualizada) e de leitura e escrita no registrador.

Este sistema introduz a ideia de geradores de sequências de visões, os quais englobam todos os requisitos de sincronia necessários para reconfigurar o sistema. O resultado da execução destes geradores são sequências de visões para serem instaladas e com isso reconfigurar o sistema. De acordo com o nível de sincronia que o ambiente de execução deve apresentar, um gerador de sequência de visões pode ser construído de pelo menos duas formas diferentes.

- *Geradores para sistemas parcialmente síncronos*: esta abordagem utiliza um protocolo de consenso de forma que todos os servidores envolvidos concordam (entram em acordo) com a nova sequência de visões a ser instalada. Na verdade todos os servidores geram uma única sequência com uma única visão para ser instalada no sistema. Protocolos de consenso [Lamport 1998, Lamport 2001, Castro and Liskov 2002] são bem conhecidos e utilizados mas possuem a limitação de não serem implementáveis em um sistema distribuído completamente assíncrono quando falhas são previstas [Fischer et al. 1985].
- *Geradores para sistemas assíncronos*: outra possível construção para os geradores de sequências de visões é sem o uso de consenso, não adicionando nenhum requisito de sincronia para sua execução. Como não temos acordo nesta abordagem, diferentes geradores podem gerar diferentes sequências mas através das propriedades de intersecção dos quóruns o FREESTORE garante que todos os servidores acabam convergindo e instalando a mesma sequência de visões.

Uma das motivações deste trabalho é a análise prática dos ganhos e das perdas relacionadas com a utilização de cada uma destas abordagens na implementação de geradores de sequência de visões.

Em termos gerais, uma reconfiguração do FREESTORE segue três etapas básicas:

Primeira etapa
O sistema recebe os pedidos de entrada (*join*) e saída (*leave*) de processos (servidores). Esses pedidos são acumulados por um período de tempo até que o sistema passe para a segunda etapa.

Segunda etapa
O gerador de sequência de visões escolhido é executado para geração de uma nova sequência de visões para atualizar o sistema. As visões desta sequência sempre são mais atuais do que a visão atual instalada no sistema, contemplando os pedidos de atualizações coletados na primeira etapa. O protocolo vai para a terceira etapa quando uma sequência é gerada.

Terceira etapa
Uma sequência de visões gerada é instalada de forma que, no final, uma única

[2]Basicamente, uma visão nada mais é do que um conjunto com os identificadores dos servidores presentes no sistema.

visão mais atual w é efetivamente instalada no sistema. Mesmo que processos diferentes gerem sequências diferentes (este cenário é possível quando o gerador não utiliza protocolos de consenso), os protocolos do FREESTORE garantem que todos os servidores acabam instalando w.

Como os protocolos de reconfiguração são desacoplados dos protocolos de leitura e escrita (R/W) no registrador, qualquer protocolo de R/W pode ser adaptado para ambientes dinâmicos através do FREESTORE. Neste trabalho consideramos o clássico ABD [Attiya et al. 1995], que implementa um sistema de quóruns com semântica atômica [Lamport 1986] das operações de leitura e escrita. Além disso, o FREESTORE garante que todos os pedidos de entrada ou saída são executados e nenhum processo entra ou é removido (sai) do sistema sem ter realizado tal pedido.

4. Proposta de implementação dos protocolos do FREESTORE

Esta seção apresenta a nossa proposta de implementação para os protocolos do FREESTORE, cujo sistema resultante representa uma memória compartilhada dinâmica. Primeiramente apresentaremos os módulos que compõem o sistema e sua API, para então discutir as implementações realizadas.

Nossas implementações foram realizadas na linguagem de programação *Go*[3], que é aberta, compilada e com coletor de lixo. Além disso, esta linguagem apresenta várias características de programação concorrente, como por exemplo as *goroutines*, que provêem uma abstração eficiente de *threads*.

4.1. Módulos do Sistema e APIs

A implementação proposta busca manter a modularidade do FREESTORE. Assim, o sistema pode ser dividido em módulos que interagem entre si conforme mostra a Figura 1. Esses módulos são: visão atual, registrador *R/W*, reconfiguração e gerador de sequências de visões. O registrador R/W precisa acessar a visão atual para saber se uma operação de R/W está utilizando a visão atual do sistema. O módulo de reconfiguração altera a visão atual e o valor armazenado no registrador ao final de cada reconfiguração e, além disso, executa o gerador de sequências de visões que, por sua vez, retorna uma nova sequência de visões quando gerada. Note que o gerador ainda possui um módulo de consenso interno que pode ser empregado quando necessário, conforme já discutido.

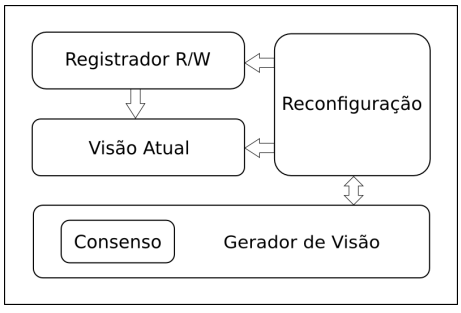


Figura 1. Módulos do sistema.

A implementação representa um sistema de quóruns com capacidade de reconfiguração e é dividida em clientes e servidores. Todas as funcionalidades do sistema

[3]Disponível em `http://golang.org`.

são implementadas em forma de bibliotecas, facilitando a integração do sistema dentro de uma aplicação maior que precise de memória compartilhada. As bibliotecas apresentam as seguintes APIs:

Funções do cliente:

Read() (int, error) – Retorna o valor armazenado no sistema e um erro, o valor só é válido se o erro for nulo.

Write(v int) error – Escreve o valor *v* no sistema e retorna um erro caso ocorra algum problema.

Funções dos servidores

Run(bindAddr string, initialView *view.View, useConsensus bool) – Inicializa a execução de um servidor do sistema.

bindAddr: o endereço em que o servidor irá fornecer seus serviços.

initialView: visão inicial do sistema. Se o servidor for membro dessa visão inicial, todos os seus serviços serão ativados no final da inicialização, caso contrário, o servidor irá enviar o pedido de *Join* para todos os servidores dessa visão.

useConsensus: se *true*, o servidor irá utilizar o gerador de sequência de visões com consenso, caso contrário não utilizará o protocolo do consenso. Esse parâmetro deve ser igual em todos os servidores em execução no sistema.

As próximas seções discutem os aspectos principais relacionados com a implementação das APIs acima descritas. Começamos apresentando as estruturas básicas que são vastamente utilizadas em nossa implementação, para então discutir a implementação dos protocolos de R/W e de reconfiguração do sistema.

4.2. Estruturas Básicas: Identificadores de Servidores, *Updates* e Visões

As estruturas abaixo servem como tipos básicos para a implementação dos protocolos.

Identificadores de Servidores: Um servidor é identificado pelo seu endereço.

```
1 type Process struct {
2   Addr string
3 }
```

***Updates*:** Representam atualizações no grupo dos participantes do sistemas. Um *update* é composto por um tipo (entrada/saída) e um servidor.

```
1 type Update struct {
2   Type  updateType
3   Process Process
4 }
5 type updateType string
6 const (
7   Join updateType = "+"
8   Leave updateType = "-"
9 )
```

Visão: Representa a composição do sistema, sendo formada por um conjunto de *updates*.

```
1 type View struct {
2   Entries map[Update]bool
3 }
```

Duas visões são iguais quando suas *Entries* são iguais. Dizemos que uma visão A é mais atualizada que uma visão B se A contém todas as atualizações que B

possui mais algumas outras atualizações. Um processo P pertence a uma visão caso a mesma possua o *update* $+P$ (*join*) e não possua o *update* $-P$ (*leave*). Cada servidor contém a visão atual do sistema, que é utilizada para verificar o quão atual são os eventos recebidos, sendo atualizada pelo módulo de reconfiguração.

4.3. Implementação do Protocolo de Leitura e Escrita

O protocolo de leitura e escrita em um registrador implementado é uma adaptação do clássico ABD [Attiya et al. 1995] para utilização com os protocolos do FREESTORE. A única mudança é que as requisições dos clientes agora precisam conter a visão mais atual conhecida pelo cliente. Um servidor que recebe um pedido com uma visão antiga apenas informa o cliente sobre a nova visão instalada, com a qual o mesmo deve reenviar seu pedido (reinicia a fase da operação que está executando).

Essa parte do sistema implementa a interface utilizada pelo cliente para processamento dos pedidos de *read* e *write*. O registrador é uma variável de tipo genérico, assim qualquer valor pode ser armazenado. Associado ao registrador temos um *mutex* para realizar o controle de concorrência entre *goroutines* concorrentes e para suspender as operações de R/W enquanto o servidor estiver atualizando a sua visão.

A Figura 2 ilustra o processamento das requisições em um servidor: ao receber uma dessas requisições, o servidor utiliza uma *goroutine* para atender a tal solicitação. Primeiro é verificado se a visão utilizada pelo cliente é atual (*cvMutex* controla a concorrência entre estes pedidos e reconfigurações que alteram a visão atual) para então acessar o valor do registrador (*registerMutex* controla a concorrência entre estes pedidos e também suspende a execução de operações de R/W enquanto a visão é atualizada).

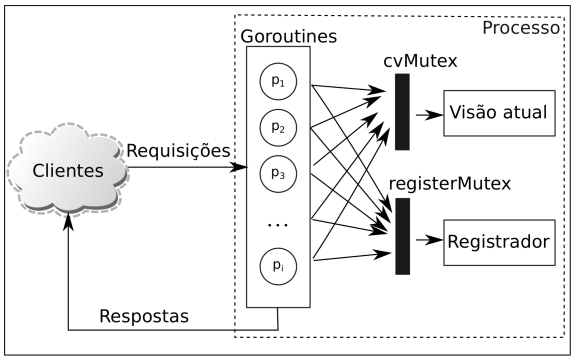


Figura 2. Processamento das requisições dos clientes.

As mensagens trocadas entre clientes e servidores contém a seguinte estrutura:

```
type Value struct {
  Value   interface{} // Valor do registrador
  Timestamp int // O timestamp do valor do registrador
  View view.View // A visão do cliente
  Err error  // Erro caso houver
}
```

Um cliente realiza as operações de leitura e escrita conforme o código abaixo. A função *readQuorum* solicita o valor atual do registrador de todos os membros da visão atual e retorna o valor com o maior *timestamp*, dentre os valores recebidos da maioria dos servidores. A função *writeQuorum* por sua vez faz com que todos os membros da visão

atual escrevam no registrador o valor em *writeMsg*, e retorna com sucesso quando recebe a confirmação de uma maioria. Ambas essas funções tratam o caso em que a visão atual do sistema esteja desatualizada e a fase em execução precisa ser reiniciada.

```
// Write v to the system's register.
func (cl *Client) Write(v interface{}) error {
  readValue, err := cl.readQuorum()
  if err != nil {
    return err
  }
  writeMsg := RegisterMsg{}
  writeMsg.Value = v
  writeMsg.Timestamp = readValue.Timestamp + 1
  writeMsg.ViewRef = cl.ViewRef()
  err = cl.writeQuorum(writeMsg)
  if err != nil {
    return err
  }
  return nil
}
// Read the register value.
func (cl *Client) Read() (interface{}, error) {
  readMsg, err := cl.readQuorum()
  if err != nil {
    // Special case:  diffResultsErr
    if err == diffResultsErr {
      log.Println("Found divergence: Going to write-back phase of read protocol")
      return cl.read2ndPhase(readMsg)
    } else {
      return nil, err
    }
  }
  return readMsg.Value, nil
}
func (cl *Client) read2ndPhase(readMsg RegisterMsg) (interface{}, error) {
  err := cl.writeQuorum(readMsg)
  if err != nil {
    return nil, err
  }
  return readMsg.Value, nil
}
```

4.4. Implementação do Protocolo de Reconfiguração

O protocolo de reconfiguração, ilustrado na Figura 3, pode ser encontrado em [Alchieri et al. 2014]. De forma resumida, os pedidos de *join* e *leave* são armazenados na variável *recv* do tipo *map* (um *mutex* controla acessos concorrentes). A reconfiguração só começa quando um *timeout* ocorre para a visão atual (este parâmetro pode ser configurado no sistema implementado). Neste caso, uma nova sequência de visões é gerada e instalada no sistema.

O gerador de sequências de visões (Seção 4.5) recebe como parâmetro a visão associada e a sequência inicial (que normalmente é a visão atual com as atualizações em *recv*) a ser proposta. Mais de uma sequência pode ser gerada em uma única execução do gerador e são enviadas novamente para o código de reconfiguração por um canal do tipo:

```
type newViewSeq struct {
  ViewSeq     []view.View // sequência de visões gerada
  AssociatedView view.View // visão associada
}
```

Assim que uma sequência é recebida, os servidores instalam a visão mais atual contida na mesma através das seguintes mensagens:

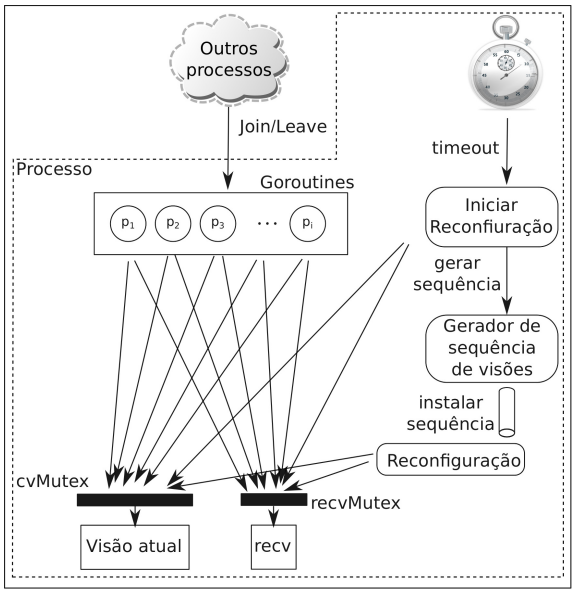


Figura 3. Reconfiguração do sistema

INSTALL-SEQ: É enviada por um servidor que quer instalar uma sequência de visões seq, inicializando este procedimento. As visões em seq são inicializadas uma a uma até que a visão mais atual seja instalada no sistema.

STATE-UPDATE: Os servidores atualizam o estado (valor do registrador e *timestamp*), além do conjunto *recv*, de uma visão instalada através destas mensagens.

VIEW-INSTALLED: É enviada por um servidor para informar que instalou uma visão, liberando os processos que desejam sair do sistema.

Um aspecto interessante do FREESTORE é a separação entre o registrador R/W e a reconfiguração do sistema (Figura 4). Apesar destes protocolos agirem de forma independente, dois *mutexes* (no registrador e na visão atual) devem controlar a execução concorrente de ambos.

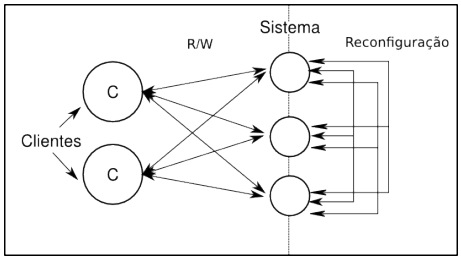


Figura 4. Protocolo de reconfiguração é separado do protocolo de R/W.

4.5. Implementação dos Geradores de Sequência de Visões

Um gerador de sequência de visões captura os requisitos de sincronia necessários entre os servidores para o protocolo de reconfiguração. Esse trabalho implementa dois geradores de sequência de visões. Um faz uso do protocolo de consenso e assim garante que todos os servidores recebam a mesma sequência de visões de seus geradores, enquanto o outro garante a propriedade de que toda sequência gerada contém ou está contida em

uma sequência gerada anteriormente, o que é suficiente para garantir as propriedades de segurança e vivacidade do FREESTORE.

4.5.1. Gerador de Sequência de Visões com Consenso

Esse gerador utiliza um algoritmo de consenso para alcançar acordo sobre a sequência gerada. A solução do consenso implementada foi a do **Paxos** [Lamport 2001]. A implementação considera todos os servidores da visão atual do sistema como *acceptors* e *learners* e funciona da seguinte forma: quando o protocolo de reconfiguração requisita uma sequência, um líder é selecionado para propor uma sequência de visões seq, a qual contém apenas uma visão cujo os membros são definidos pela visão atual mais os pedidos de *join* e *leave* que o líder recebeu desde a última reconfiguração; o protocolo termina com todos decidindo por seq.

4.5.2. Gerador de Sequência de Visões sem Consenso

Este gerador não utiliza consenso, podendo gerar sequências diferentes em diferentes servidores. Através das propriedades de intersecção dos quóruns, uma sequência gerada sempre vai estar contida em outra sequência gerada posteriormente, garantindo as propriedades do sistema. A Figura 5 ilustra o processamento deste gerador. Cada servidor que requisita uma sequência de visões, cria uma instância do gerador que possui sua própria *goroutine*. As mensagens desse gerador são enviadas para o canal da visão associada, onde a *goroutine* correspondente realiza o devido processamento. Para gerar uma sequência de visões, as seguintes mensagens são utilizadas:

SEQ-VIEW: Usada para enviar uma proposta de sequência. Ao receber uma proposta diferente da atual, um servidor atualiza sua proposta computando os novos pedidos de atualização (*updates*) e envia novamente esta mensagem. Ao receber um quórum destas mensagens, um servidor converge para uma sequência.

SEQ-CONV: Um servidor envia esta mensagem para informar que convergiu para uma sequência. Ao receber um quórum destas mensagens, um servidor gera uma sequência para ser instalada no sistema, retornando a mesma para o algoritmo de reconfiguração.

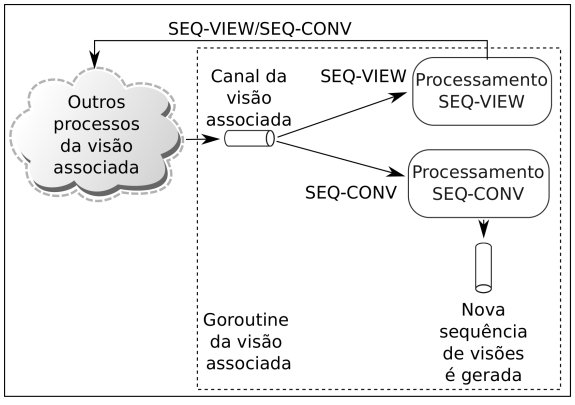


Figura 5. Gerador de sequências de visões sem consenso.

4.6. Inicialização de Servidores e Clientes

Um servidor do sistema, quando inicializado, realiza as seguintes principais preparações:

- **Inicializa *listener*:** Variável que contém a conexão do servidor com a rede.
- **Inicializa *register*:** Variável que representa o registrador, sendo inicializada para um valor inicial nulo com *timestamp* 0. Um *mutex* bloqueia o acesso ao registrador, apenas sendo desbloqueado quando o servidor instala uma visão a que pertence, i.e., habilita a execução de operações de *R/W*.
- **Inicializa *currentView*:** Variável que contém a atual visão instalada no sistema, sendo inicializada da seguinte forma:
  - **Servidor pertence à visão inicial do sistema:** O servidor simplesmente instala a visão inicial do sistema.
  - **Servidor não pertence à visão inicial do sistema:** O servidor requisita a visão atual do sistema a outro servidor do sistema. O endereço desse servidor é passado na sua inicialização.
- **Outros:** Os módulos responsáveis pela escuta de novas conexões são inicializados; as estruturas principais são inicializadas (ex.: canais e *maps*); as *goroutines* referentes ao protocolo de reconfiguração (ex.: *goroutine* que mantém o timer que dispara uma nova reconfiguração) que sempre estão ativas são inicializadas.

Por outro lado, quando um cliente é inicializado, o mesmo apenas realiza a inicialização da *currentView* requisitando a um servidor do sistema a visão atual. Após isso, tal servidor está apto a realizar operações de *R/W*.

5. Experimentos

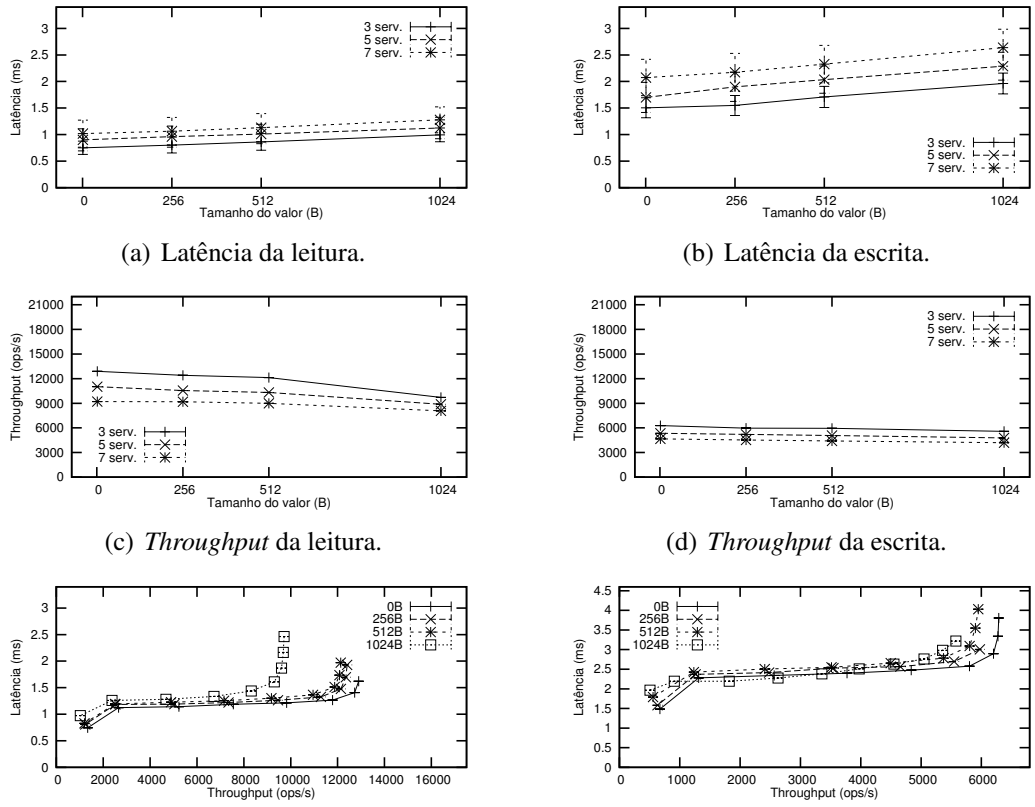
Visando analisar o desempenho da implementação desenvolvida, alguns experimentos foram realizados no Emulab [White et al. 2002], em um ambiente consistindo por até 20 máquinas *pc3000* (3.0 GHz 64-*bit* Pentium Xeon com 2GB de RAM e interface de rede gigabit) conectadas a um *switch* de 100Mb. Cada servidor executou em uma máquina separada, enquanto que até 26 clientes foram distribuídos uniformemente nas máquinas restantes (2 por máquina). O ambiente de *software* utilizado foi o sistema operacional Fedora 15 64-*bit* com *kernel* 2.6.20 e compilador *Go* 1.2.

Os experimentos realizados foram: (1) alguns *micro-benchmarks* para avaliar a latência e o *throughput*; (2) uma comparação entre o modelo estático e o modelo dinâmico; e (3) um experimento que mostra o comportamento do sistema na presença de falhas e reconfigurações. Tanto a latência quanto o *throughput* foram medidos nos clientes, sendo que o *throughput* foi calculado somando-se quantas operações o conjunto de clientes conseguiu executar.

***Micro-benchmarks*.** Começamos pela análise de uma série de *micro-benchmarks* comumente utilizados para avaliar estes sistemas, os quais consistem na leitura e escrita de valores com diferentes tamanhos: 0, 256, 512 e 1024 *bytes*.

As figuras 6(a) e 6(b) mostram a latência da leitura e escrita, respectivamente, para o sistema configurado com 3,5,e 7 servidores (i.e., tolerando 1,2 ou 3 falhas), enquanto que as figuras 6(c) e 6(d) apresentam os valores de *throughput* para as mesmas configurações. Já as figuras 6(e) e 6(f) apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores.

Neste primeiro conjunto de experimentos, podemos perceber que tanto o aumento no número de servidores quanto no tamanho do valor a ser lido/escrito faz com que o desempenho do sistema diminua. Este comportamento era esperado e deve-se ao fato de que com mais servidores um maior número de mensagens são enviadas e esperadas para atingir um quórum, enquanto que o aumento no tamanho do valor faz com que estas mensagens também tenham um tamanho maior. Finalmente, podemos perceber que os custos de uma leitura são praticamente a metade dos custos de uma escrita. Isso deve-se ao protocolo de leitura e escrita empregado [Attiya et al. 1995], que requer dois passos de comunicação para leituras e quatro passos para escritas, considerando execuções sem concorrência entre leituras e escritas.



(a) Latência da leitura.

(b) Latência da escrita.

(c) *Throughput* da leitura.

(d) *Throughput* da escrita.

(e) Leitura ($n = 3$ servidores).

(f) Escrita ($n = 3$ servidores).

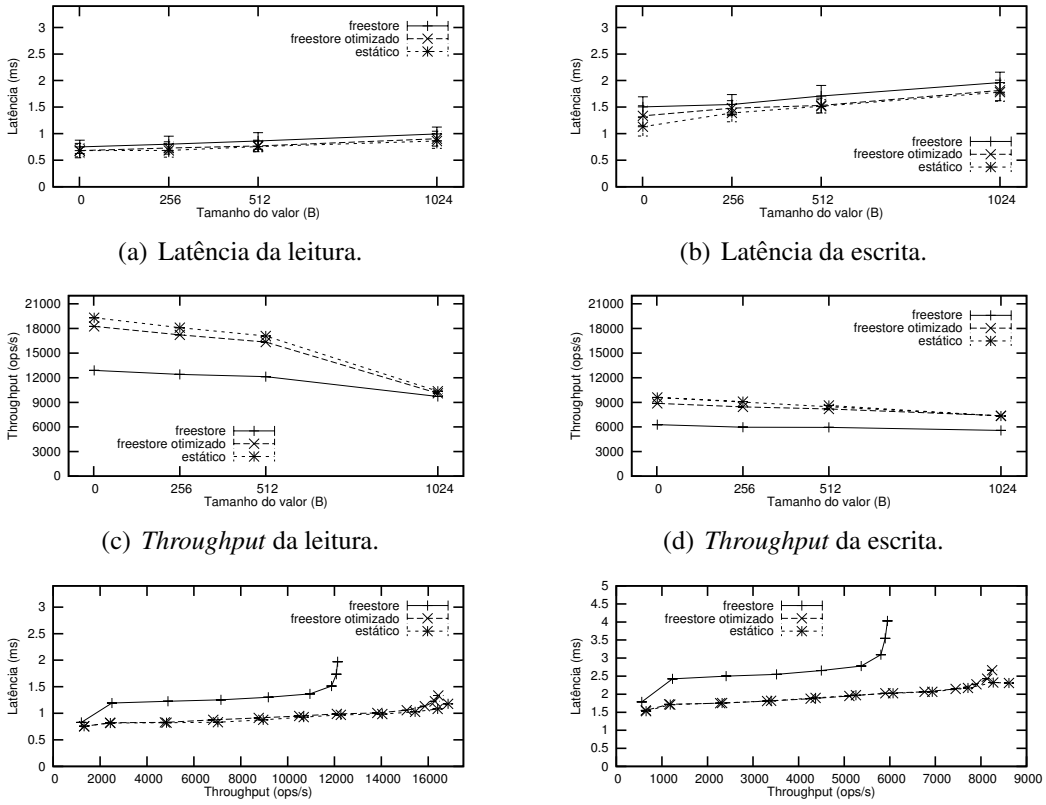
Figura 6. Desempenho do sistema.

Dinâmico vs. Estático. Este segundo experimento visa comparar o desempenho do sistema implementado com a sua versão para ambientes estáticos (onde o conjunto de servidores é fixo). Basicamente, os custos adicionais na versão dinâmica são o envio da visão atual conhecida pelo cliente em cada mensagem e a verificação se a mesma é a visão mais atualizada do servidor. Reportamos também os resultados para uma implementação otimizada do FREESTORE, onde apenas um *hash* criptográfico (SHA-1) das visões é anexado nas mensagens, diminuindo o tamanho das mesmas.

As figuras 7(a) e 7(b) apresentam os valores para a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores, enquanto que as figuras 7(c)

e 7(d) apresentam os valores de *throughput*. Já as figuras 7(e) e 7(f) apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, considerando um valor de 512 *bytes*.

Nestes experimentos podemos perceber que a versão otimizada do FREESTORE apresenta um desempenho praticamente igual a sua versão estática. Conforme já comentado, a modularidade do FREESTORE faz com que os protocolos de reconfiguração praticamente não interfiram nos protocolos de leitura e escrita, quando executados em períodos onde não estão ocorrendo reconfigurações. Pelos mesmos motivos anteriormente descritos, as escritas apresentaram praticamente a metade do desempenho das leituras.



(a) Latência da leitura.

(b) Latência da escrita.

(c) *Throughput* da leitura.

(d) *Throughput* da escrita.

(e) Leitura (valor de 512 *bytes*).

(f) Escrita (valor de 512 *bytes*).

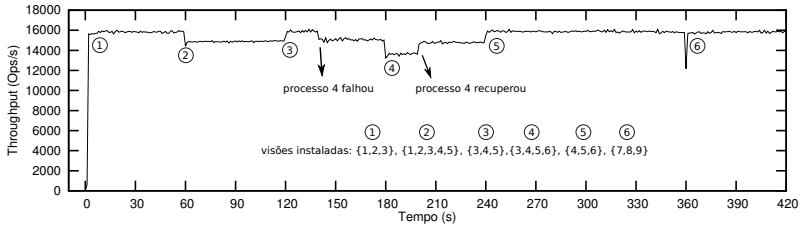
Figura 7. Dinâmico vs. Estático ($n = 3$ servidores).

Reconfigurações e Falhas. Este último experimento visa analisar o comportamento do sistema em execuções com reconfigurações e falhas de processos. Para isso, o sistema foi inicialmente configurado com 3 servidores (visão inicial = $\{1,2,3\}$) e os clientes executaram operações de leitura de um valor com tamanho de 512 *bytes*. O período entre reconfigurações foi configurado em 60 segundos, i.e., a cada 60 segundos uma reconfiguração é executada.

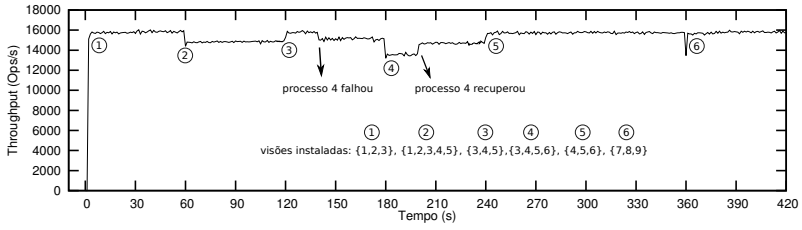
As figuras 8(a) e 8(b) mostram o *throughput* apresentado pelo sistema durante a sua execução utilizando ou não um protocolo de consenso para reconfigurações, respectivamente. Os eventos que ocorreram durante a execução foram os seguintes:

- 0s - início do experimento.

- 30s - o servidor 4 envia um pedido de *join* e aguarda sua entrada no sistema.
- 40s - o servidor 5 envia um pedido de *join* e aguarda sua entrada no sistema.
- 60s - uma reconfiguração adiciona os servidores 4 e 5 no sistema, instalando a visão $\{1,2,3,4,5\}$.
- 80s - o servidor 1 envia um pedido de *leave* e aguarda sua saída do sistema.
- 100s - o servidor 2 envia um pedido de *leave* e aguarda sua saída do sistema.
- 120s - uma reconfiguração remove os servidores 1 e 2 do sistema, instalando a visão $\{3,4,5\}$.
- 140s - o servidor 4 sofre um *crash* (falha).
- 160s - o servidor 6 envia um pedido de *join* e aguarda sua entrada no sistema.
- 180s - uma reconfiguração adiciona o servidor 6 no sistema, instalando a visão $\{3,4,5,6\}$.
- 200s - o servidor 4 se recupera e volta ao sistema.
- 220s - o servidor 3 envia um pedido de *leave* e aguarda sua saída do sistema.
- 240s - uma reconfiguração remove o servidor 3 do sistema, instalando a visão $\{4,5,6\}$.
- 300s - o sistema tenta executar uma reconfiguração, mas como não havia nenhum pedido, nada é processado.
- 320s - os servidores 4,5 e 6 enviam um pedido de *leave* e aguardam pelas suas saídas do sistema.
- 340s - os servidores 7,8 e 9 enviam um pedido de *join* e aguardam pelas suas entradas no sistema.
- 360s - uma reconfiguração removes os servidores 4,5 e 6 e adiciona os servidores 7,8 e 9 no sistema, instalando a visão $\{7,8,9\}$.
- 420s - fim do experimento.



(a) Com consenso.



(b) Sem consenso.

Figura 8. Reconfigurações e falhas.

Neste experimento podemos perceber que o desempenho do sistema diminui quando mais servidores estiverem presentes no mesmo. Além disso, devido às características de desacoplamento e modularidade do FREESTORE, o desempenho é praticamente o mesmo para ambas as abordagens de reconfiguração (com ou sem consenso).

De fato, o tempo médio de uma reconfiguração foi de 19ms na abordagem sem consenso (como os pedidos de reconfiguração foram recebidos por todos os processos antes do início da reconfiguração, todos tinham a mesma proposta e o protocolo do gerador converge com apenas um passo – *SEQ-VIEW*, além disso este tempo também engloba os passos do algoritmo de reconfiguração), bloqueando as operações de R/W por apenas 4ms. Para a solução com consenso, o tempo médio de uma reconfiguração foi de 40ms (tempo para executar o protocolo de consenso e os passos de algoritmo de reconfiguração), bloqueando as operações de R/W por 4ms. Note que o tempo em que as operações de R/W ficaram bloqueadas foi o mesmo em ambas as abordagens, pois este bloqueio somente ocorre depois de uma sequência ser gerada (i.e., após a execução do gerador).

6. Conclusões

Este artigo discutiu uma implementação para os protocolos do FREESTORE e através de um conjunto de experimentos possibilitou uma maior compreensão sobre o comportamento destes protocolos quando inseridos nos mais variados cenários de execução, possibilitando também uma maior entendimento sobre o processo de reconfiguração de memória compartilhada. Como trabalhos futuros, pretendemos implementar os protocolos do DynaStore [Aguilera et al. 2011] e comparar o desempenho destes sistemas.

Referências

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

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

Alchieri, E., Bessani, A., Greve, F., and Fraga, J. (2014). Reconfiguração modular de sistemas de quóruns. In *Anais do 32º Simpósio Brasileiro de Redes de Computadores*.

Attiya, H., Bar-Noy, A., and Dolev, D. (1995). Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142.

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

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Gifford, D. (1979). Weighted voting for replicated data. In *7th ACM Symposium on Operating Systems Principles*.

Gilbert, S., Lynch, N., and Shvartsman, A. (2010). Rambo: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4).

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

Lamport, L. (1998). The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169.

Lamport, L. (2001). Paxos Made Simple. *ACM SIGACT News*, 32(4):18–25.

Malkhi, D. and Reiter, M. (1998). Secure and scalable replication in Phalanx. In *17th Symposium on Reliable Distributed Systems*, pages 51–60.

Martin, J.-P. and Alvisi, L. (2004). A Framework for Dynamic Byzantine Storage. In *34th International Conference on Dependable Systems and Networks*.

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*.