Especificação de Replicação Máquina de Estados Dinâmica

Eduardo Adilio Pelinson Alchieri[1] **, Alysson Neves Bessani**[2] **, João Sousa**[2]

[1] Departamento de Ciência da Computação, Universidade de Brasília

[2]Faculdade de Ciências, Universidade de Lisboa

Resumo. *A replicação Máquina de Estados (RME), abordagem muito utilizada na implementação de sistemas tolerantes a falhas, consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas, com o intuito de que as mesmas apresentem a mesma evolução em seus estados. Apesar de alguns sistemas implementarem uma RME com capacidade de reconfiguração, não existe nenhuma especificação a respeito dos aspectos relacionados com a reconfiguração de uma RME. Neste sentido, este artigo especifica as propriedades e as suposições necessárias para uma RME dinâmica, além de propor um protocolo que satisfaz estas propriedades. Além disso, apresenta uma avaliação experimental que facilita o entendimento e mostra a viabilidade prática de uma RME dinâmica.*

Abstract. *State Machine Replication is an approach widely used to implement fault-tolerant systems. The idea behind this approach is to replicate the servers and to coordinate the interactions among clients and servers replicas, making all of these replicas present the same state evolution. Although some systems implement a reconfigurable RME, this approach has not yet been fully specified. In this sense, this paper specifies the properties and the assumptions that are necessary to a reconfigurable RME. Moreover, it presents a protocol that satisfies these properties and an experimental evaluation that shows the practical feasibility of this approach.*

1. Introdução

A Replicação Máquina de Estados (RME) [Schneider 1990, Lamport 1998] é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas, tanto por parada [Lamport 1998, Schneider 1990] quanto Bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e estas réplicas, com o intuito de que as mesmas apresentem a mesma evolução em seus estados, garantindo o determinismo de réplicas.

A grande maioria das concretizações existentes para RME (ex.: [Castro and Liskov 2002, Clement et al. 2009, Santos and Schiper 2013]) consideram que o conjunto de réplicas que implementam o sistema nunca sofre alterações. Esta abordagem impossibilita, por exemplo, que durante a execução do sistema novas réplicas sejam adicionadas e/ou réplicas antigas sejam removidas em tempo de execução. No entanto, muitos sistemas necessitam executar estas ações quando querem aumentar ou diminuir o limite de falhas toleradas ou ainda trocar um servidor antigo com uma configuração obsoleta por um servidor atualizado. Para isso, surge a necessidade de *reconfiguração* do sistema.

O sistema SMART [Lorch et al. 2006], que tolera apenas falhas por parada, e o sistema BFT-SMART [Alchieri et al. 2013, Bessani et al. 2014], que suporta falhas tanto por parada quanto Bizantinas, são gratas exceções que implementam uma RME reconfigurável. Lamport *et al.* [Lamport et al. 2010] também discutem, embora superficialmente (nem mesmo apresentam protocolos), aspectos sobre a reconfiguração de uma RME. Um sistema reconfigurável fornece as interfaces necessárias para sua reconfiguração, não se preocupando com as políticas relacionadas com a escolha tanto do momento da reconfiguração quanto da nova configuração a ser implantada no sistema. Os algoritmos que implementam estes sistemas devem prever a possibilidade de entradas e saídas de réplicas. Estas ações de reconfiguração devem ser sincronizadas com as execuções dos protocolos da RME, a fim de preservar a consistência dos estados das réplicas.

Embora existam estas implementações, ainda falta uma especificação formal e precisa a respeito dos aspectos relacionados com a reconfiguração de uma RME. Os trabalhos sobre RME dinâmica não discutem, por exemplo, quanto tempo um servidor precisa permanecer no sistema após ser removido pela execução de uma reconfiguração ou quais são as novas suposições necessárias para o sistema funcionar em um ambiente dinâmico.

Visando preencher esta lacuna, este artigo apresenta as seguintes contribuições: *(1)* especificação de propriedades e de suposições necessárias para uma RME dinâmica, além de uma série de definições relacionadas com o dinamismo do sistema; *(2)* proposta de um protocolo que, considerando as suposições definidas, satisfaz as propriedades especificadas para uma RME dinâmica; e *(3)* implementação e análise experimental do protocolo proposto, o que ajuda a entender o funcionamento de uma RME dinâmica.

2. Modelo de Sistema

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos U, que é dividido em dois subconjuntos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$ e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. A chegada dos processos segue o modelo de chegadas infinitas com concorrência desconhecida mas finita [Aguilera 2004]. Desta forma, em cada instante de tempo real t da execução, o número de processos executando alguma ação no sistema é desconhecido mas finito. Contudo, processos podem chegar em qualquer momento (chegadas infinitas) e também passarão a participar das computações executadas no sistema.

Com relação ao modelo de sincronia, consideramos um sistema parcialmente síncrono [Dwork et al. 1988], necessário para garantir a terminação dos protocolos da RME [Schneider 1990, Lamport et al. 1982, Castro and Liskov 2002]. Além disso, os protocolos propostos são genéricos suficientes para serem usados em um sistema que tolera tanto falhas por parada (CFT - *Crash Fault-Tolerant* RME) [Schneider 1990] quanto Bizantinas (BFT - *Byzantine Fault-Tolerant* RME) [Castro and Liskov 2002]. Na especificação da RME dinâmica (Seção 4), são definidos os tamanhos dos quóruns e o número de falhas toleradas pelos protocolos para cada um destes modelos de falhas.

3. Replicação Máquina de Estados

A Replicação Máquina de Estados (RME) consiste em replicar os servidores e coordenar as interações entre os clientes e estas réplicas, com o objetivo de que as mesmas

apresentem a mesma evolução em seus estados. Conceitualmente, o modelo básico de programação de uma RME envolve duas primitivas (Figura 1):

- *invoke(operation)*: utilizada pelos clientes para invocar operações no serviço implementado pela RME.
- *execute(operation)*: implementada pelos servidores replicados, sendo utiliza sempre que uma operação deve ser executada pela RME.

De um modo geral, um cliente utiliza a primitiva *invoke(operation)* para enviar a requisição *operation* para as réplicas, as quais executam a primitiva *execute(operation)* para processar *operation* e retornar a resposta a ser enviada para o cliente.

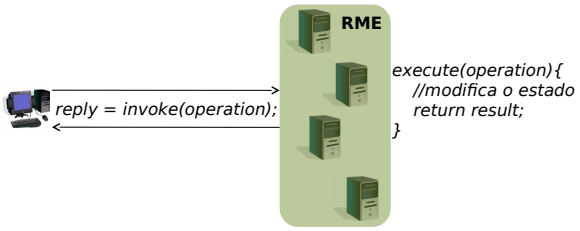


Figura 1. Modelo básico de programação de uma RME.

A implementação destas primitivas deve atender aos seguintes requisitos, que definem o determinismo de réplicas:

1. Estado inicial: todas as réplicas corretas devem iniciar a partir de um mesmo estado.
2. Determinismo: todas as réplicas corretas, que executam uma mesma operação sobre um mesmo estado, realizam a mesma mudança em seus estados e produzem a mesma resposta para o cliente.
3. Coordenação: todas as réplicas corretas executam a mesma sequência de operações.

Para garantir o item *(1.)* basta iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas), o que é trivial em uma RME estática mas, como veremos mais adiante, requer cuidados especiais em uma RME dinâmica. Para prover o item *(2.)*, é necessário que as operações executadas pelos servidores sejam determinísticas. Finalmente, para que as réplicas executem a mesma sequência de operações afim de garantir o item *(3.)*, é necessária a utilização de um protocolo de difusão atômica [Hadzilacos and Toueg 1994][1].

Assumindo que estes três requisitos sejam satisfeitos, uma implementação de RME deve satisfazer as seguintes propriedades de segurança e terminação:

- Segurança: todas as réplicas corretas executam a mesma sequência de operações.
- Terminação: todas as operações de clientes corretos são executadas.

A propriedade de segurança permite a implementação de consistência forte, conhecida como linearizabilidade (*linearizability*) [Herlihy and Wing 1990]. Já a propriedade de terminação garante que as operações invocadas por clientes corretos são exe-

[1]Difusão atômica, ou difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem.

cutadas e acabarão terminando. Os protocolos propostos para RME limitam a quantidade de falhas que o sistema pode apresentar para que estas propriedades sejam atendidas, sendo necessário $n \geq 2f + 1$ (CFT) [Lamport 1998] ou $n \geq 3f + 1$ (BFT) [Castro and Liskov 2002] réplicas para tolerar até f falhas.

4. Replicação Máquina de Estados Dinâmica

Esta seção apresenta algumas definições e propriedades que uma RME dinâmica deve satisfazer, juntamente com as suposições necessárias para sua execução. Posteriormente, são discutidas as etapas para uma reconfiguração.

4.1. Definições e Propriedades de uma RME Dinâmica

As seguintes definições são necessárias na especificação de uma RME dinâmica.

Updates. Definimos um $update = \{+, -\} \times \Pi$, onde a tupla $\langle +, i\rangle / \langle -, i\rangle$ representa o pedido de entrada/saída (*join/leave*) do servidor i. O processo de *reconfiguração* considera os *updates* solicitados para definir a nova composição do sistema.

Visões. Uma visão v é composta por um identificador único (representado por $v.id$) e seu *membership* (representado por $v.members$). Por simplicidade, usamos $i \in v$ para simbolizar $i \in v.members$. Duas visões v_1 e v_2 são comparadas através de seus identificadores. Caso $v_1.id < v_2.id$, então v_2 é *mais atual* do que v_1.

Dizemos que uma visão v é *instalada em um servidor correto* caso o mesmo considere v sua *visão atual* e passa a executar operações nesta visão. Neste caso, dizemos que v foi *instalada no sistema*. Em qualquer tempo real t, definimos $V(t)$ como sendo *a visão mais atual* instalada no sistema.

Bootstrapping. Assumimos uma visão inicial não vazia $V(0)$ conhecida por todos os processos do sistema. Durante a inicialização do sistema, cada servidor $i \in V(0)$ recebe a visão inicial $v_0 = \{\langle +, j\rangle : j \in V(0)\}$.

Visões vs. operações de clientes. Em qualquer tempo real t, operações de clientes são executadas apenas em $V(t)$. Quando é requisitado um *join* no sistema para um servidor i, estas operações permanecem desabilitadas em i até a ocorrência de um evento *enable operations*. Após isso, i permanecerá apto a processar e responder a estas operações até que seja requisitado um *leave* do sistema para i, o que acontece com a ocorrência do evento *disable operations*.

Definição 1 (Propriedades de uma RME Dinâmica) *As seguinte propriedades são satisfeitas por uma RME dinâmica:*

○ ***RME (Segurança)****: Todas as réplicas corretas que executam uma operação possuem o mesmo estado, o qual reflete a execução da mesma sequência de operações.*

○ ***RME (Terminação)****: Todas operações de clientes corretos são executadas.*

○ ***Reconfiguração – Join (Segurança)****: Se um servidor correto j instala a visão v tal que $i \in v$, então foi invocada a operação $invoke(\langle +, i\rangle)$ ou i é membro da visão inicial.*

○ ***Reconfiguração – Leave (Segurança)****: Se um servidor correto j instala a visão v tal que $i \notin v \wedge (\exists v' : i \in v' \wedge v'.id < v.id)$, então foi invocada a operação $invoke(\langle -, i\rangle)$.*

○ ***Reconfiguração – Join (Terminação)****: O evento enable operations terminará por ocorrer em todo servidor correto i cuja operação $invoke(\langle +, i\rangle)$ foi requisitada.*

○ ***Reconfiguration – Leave (Terminação)****: O evento disable operations terminará por ocorrer em todo servidor correto i cuja operação $invoke(\langle -, i\rangle)$ foi requisitada.*

Para suportar a dinamicidade do sistema, a propriedade de segurança da RME precisou ser modificada (e de certa forma enfraquecida) para permitir que nem todos os servidores executem a mesma sequência de operações. De fato, não faz sentido que um servidor que entra no sistema seja obrigado a executar todas as operações anteriores já executadas no mesmo. Ao invés disso, modificamos esta propriedade para especificar que o estado deste servidor deve refletir as execuções passadas, o que é suficiente para manter o determinismo de réplicas. Isso é garantido pela transferência de estado quando da instalação de uma nova visão. Por outro lado, a propriedade de terminação da RME não foi alterada e define que todas as operações de clientes corretos são executadas, possibilitando que as mesmas terminem.

Além disso, foram definidas *(a)* duas propriedades de segurança da reconfiguração que garantem que um processo entra e/ou sai do sistema apenas se a operação ($invoke$) correspondente foi executada e *(b)* duas propriedades de terminação da reconfiguração que garantem que as operações são habilitadas e/ou desabilitadas em um servidor, uma vez que a operação ($invoke$) correspondente foi requisitada.

4.2. Suposições para uma RME Dinâmica

Para lidar com a dinamicidade do sistema, as seguintes suposições são necessárias.

Suposição 1 (Limiar de falhas) *Para cada visão v, denotamos $v.f$ como sendo o número de falhas toleradas em v, $v.f \leq \lfloor \frac{|v.members|-1}{2} \rfloor$ (CFT) ou $v.f \leq \lfloor \frac{|v.members|-1}{3} \rfloor$ (BFT).*

Suposição 2 (Tamanho dos quóruns) *Para cada visão v, consideramos quóruns de escrita de tamanho $v.q_e = \lceil \frac{|v.members|+1}{2} \rceil$ (CFT) e $v.q_e = \lceil \frac{2|v.members|+1}{3} \rceil$ (BFT) e quóruns de leitura de tamanho $v.q_l = 1$ (CFT) e $v.q_l = \lceil \frac{|v.members|+2}{3} \rceil$ (BFT).*

Suposição 3 (Saídas graciosas) *Um servidor correto $i \in V(t)$, cuja operação $invoke(\langle -, i\rangle)$ é requisitada no tempo t, permanece no sistema até enviar seu estado para pelo menos um quórum de escrita da visão seguinte na sequência $V(t'), t' > t, i \notin V(t')$.*

Suposição 4 (Reconfigurações finitas) *O número de updates requisitados em uma execução é finito.*

As suposições 1 e 2 são adaptações diretas dos resultados para sistemas estáticos [Lamport 1998, Castro and Liskov 2002]. A Suposição 3 diz que um servidor correto que está deixando o sistema participará do protocolo de reconfiguração. Já a Suposição 4 garante que operações de clientes sempre terminam, i.e., garante que um cliente lento reiniciará sua operação, devido a reconfigurações, um número finito de vezes e, então, completará sua operação[2]. Esta suposição é necessária em todas as propostas de protocolos que envolve clientes acessando um sistema dinâmico [Alchieri et al. 2014, Aguilera et al. 2011, Martin and Alvisi 2004], pois para não utilizar dados obsoletos os mesmos precisam acessar a visão mais atual do sistema.

[2]Apenas o número de *updates* requisitados concorrentemente com uma operação deve ser finito.

4.3. Fases de uma Reconfiguração

Durante a execução de uma RME dinâmica, uma sequência de visões $V_0 \rightarrow ... \rightarrow V_{i-1} \rightarrow V_i \rightarrow V_{i+1} \rightarrow ...$ é instalada para computar os pedidos de entrada (*join*) e de saída (*leave*) do sistema. A dinamicidade do sistema é observada na medida em que novas visões são instaladas. O estado inicial das réplicas de cada visão V_i é definido como o estado final das réplicas da visão anterior na sequência V_{i-1}, garantindo que nada daquilo que foi processado em V_{i-1} seja perdido. A atualização de uma visão V_i, que compreende sua desativação e a instalação da visão seguinte V_{i+1}, ocorre através de um procedimento de reconfiguração do sistema, o qual pode ser dividido nas seguintes fases (Figura 2):

(Fase 1) Determinação da ordem de execução da operação de reconfiguração. A ideia é que as operações de reconfiguração sejam processadas pela RME como uma operação normal, i.e., o protocolo de difusão atômica é executado para definir a ordem de entrega desta operação. Assim, um processo que tenha o privilégio de invocar uma operação de reconfiguração (ex.: um administrador), executa a operação $invoke(\langle +/-, id_1 \rangle, ..., \langle +/-, id_n \rangle)$ passando o conjunto de *updates* (ver Seção 4.1) que deve ser aplicado na visão corrente (V_i), o que resultará na geração da visão seguinte V_{i+1}.

(Fase 2) Execução da operação de reconfiguração. Quando a operação de reconfiguração é executada, os *updates* requisitados são aplicados em V_i para gerar a visão seguinte na sequência V_{i+1} (como esta operação é ordenada, todas as réplicas em V_i a executam no mesmo ponto da execução distribuída, de forma que a mesma visão V_{i+1} é gerada em todas as réplicas). Neste ponto, os processos em V_i param de executar operações (nenhuma outra operação é executada em V_i), desativam V_i, enviam seus estados para os processos em V_{i+1}, enviam a resposta para o invocador da operação de reconfiguração, e deixam o sistema (para aqueles que estiverem saindo).

(Fase 3) Inicialização da nova visão. Por fim, os processos em V_{i+1}, após atualizarem seus estados, instalam V_{i+1} e passam a receber, ordenar e executar operações.

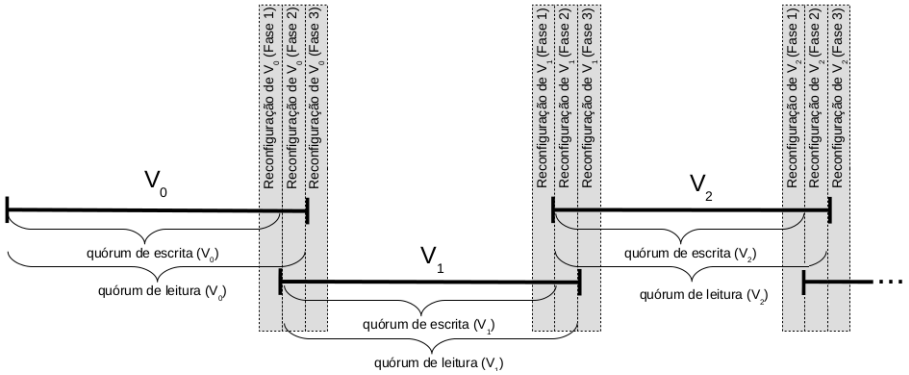


Figura 2. Visões X Reconfigurações.

Como podemos observar na Figura 2, o *ciclo de vida* de uma visão V_i começa com o início da fase 2 da reconfiguração (mesmo que V_i seja instalada apenas na fase 3), quando V_i é gerada através da aplicação de um conjunto de *updates* na visão anterior na sequência V_{i-1}, e termina no final da fase 2 da reconfiguração seguinte, quando V_i é desativada e seu estado é transferido para os servidores da visão seguinte V_{i+1} e a resposta é enviada para o processo que executou a operação de reconfiguração ($invoke$). Para tolerar até f falhas, deve-se aguardar por 1 (CFT) ou $f + 1$ (BFT) respostas iguais para então assumir que esta resposta é correta e finalizar a execução da primitiva *invoke*.

Apesar de serem necessárias $n \geq 2f+1$ (CFT) ou $n \geq 3f+1$ (BFT) réplicas para tolerar até f falhas, nem todas as fases da reconfiguração envolvem todas estas réplicas, mas sim um subconjunto delas chamado de quórum. Os protocolos subjacentes da RME, como o de ordenação, também utilizam a ideia de quóruns para votações. Desta forma, definimos quórums para escrita e para leitura (ver Seção 4.2). Considerando uma visão V_i, um quórum de escrita deve estar disponível deste o início da fase 2 da reconfiguração que a gerou, pois é necessário que as réplicas da visão anterior V_{i-1} atualizem o estado de pelo menos um quórum de escrita de V_i para garantir que o estado inicial de V_i corresponde ao estado final de V_{i-1}, até o final da fase 1 da reconfiguração seguinte que instalará V_{i+1} pois é necessário executar o protocolo de difusão atômica para definir a ordem de execução da operação de reconfiguração. Por outro lado, um quórum de leitura de V_i deve ficar disponível até o final da fase 2 para que a operação de reconfiguração seja executada, i.e., para que *(1)* a visão V_{i+1} seja gerada, *(2)* os servidores de V_{i+1} recebam um número suficiente de mensagens e consigam atualizar seus estados e *(3)* o processo que invocou a operação de reconfiguração receba um número suficiente de respostas e termine.

5. Protocolo para uma Replicação Máquina de Estados Dinâmica

Esta seção descreve o protocolo proposto para concretização de uma RME dinâmica. Para suportar reconfigurações, a ideia principal é ordenar os pedidos de reconfiguração juntamente com as requisições normais de clientes, de modo que os mesmos sejam executados por todas as réplicas no mesmo ponto da execução distribuída. Assim, uma única sequência de visões é instalada no sistema e o estado da aplicação (réplicas) é transferido entre estas visões, fazendo com que uma visão começe a funcionar com o estado da visão anterior na sequência. Já para executar operações normais, os clientes apenas devem se certificar de que estão utilizando a visão mais atual instalada no sistema, garantindo que dados atualizados são acessados.

Note que não desejamos modificar os protocolos subjacentes da RME. Por exemplo, não modificamos os protocolos da RME tradicional que fazem as contagens de respostas e a ordenação. Ao invés disso, utilizamos as primitivas da RME tradicional e incluimos os controles necessários para suportar o dinamismo.

Algoritmo. O Algoritmo 1 apresenta o protocolo utilizado pelos clientes para acesso à RME dinâmica. Basicamente, cada cliente deve armazenar uma variável cv com a visão mais atual conhecida, a qual deve ser incluida em cada requisição enviada para a RME (linha 6): através desta primitiva, a requisição $\langle operation, cv \rangle$ é enviada para todos os servidores em cv, aguardando por respostas $reply$ iguais de $cv.q_l$ (quórum de leitura) servidores em cv para retornar $reply$. Caso a visão do cliente esteja desatualizada, a resposta para sua requisição indicará que sua visão precisa ser atualizada (linhas 7-8), então a requisição é reenviada para a RME (linhas 5-9). Caso contrário, a requisição termina retornando a resposta (linha 10).

Este algoritmo também apresenta os procedimentos para o pedido de entrada (linhas1-2) e de saída (linhas 3-4) de uma réplica do sistema. Geralmente, somente um administrador pode acessar estes procedimentos, principalmente se consideramos um ambiente sujeito a falhas maliciosas. Para simplificar a apresentação dos protocolos, consideramos que apenas um *update* é solicitado em cada requisição, mas o protocolo apresentado pode ser facilmente estendido para processamento de vários *updates* por requisição.

Algoritmo 1 RME dinâmica – $join$, $leave$ e $invoke$ – cliente i.

variables: Visão atual do processo i
  $cv \leftarrow v_0$ // visão mais atual de i
procedure $join(id)$
1) $cv \leftarrow invoke(\langle +, id \rangle)$ //executa o $invoke$ solicitando a inclusão (*join*) da réplica id.
2) **return** cv
procedure $leave(id)$
3) $cv \leftarrow invoke(\langle -, id \rangle)$ //executa o $invoke$ solicitando a remoção (*leave*) da réplica id.
4) **return** cv
procedure $invoke(operation)$
5) **repeat**
6) $reply \leftarrow invoke(\langle operation, cv \rangle)$ //requisição é enviada e executada na RME.
7) **if** $reply = \langle need_update_view, v \rangle$ **then** //caso o cliente esteja com a visão desatualizada, ...
8) $cv \leftarrow v$ //... atualiza a visão e reenvia a requisição para a RME.
9) **until** $reply = \langle response, cv \rangle$ //caso a visão do cliente estava atualizada, ...
10) **return** $response$ //... termina retornando a resposta.

O Algoritmo 2 apresenta o protocolo executado por cada réplica que suporta a RME dinâmica. Cada servidor também deve armazenar a visão mais atual cv instalada no sistema e, para cada requisição recebida, verificar se a visão do cliente está atualizada para então processar a operação, seja ela uma reconfiguração (linhas 3-15) ou uma operação normal (linha 17). Caso a visão do cliente esteja desatualizada, a resposta para a requisição será a visão atualizada (linhas1-2), possibilitando que o cliente reenvie sua requisição para a nova visão.

No processamento de uma reconfiguração, primeiramente a nova visão é definida (linhas 5-9). Caso a réplica esteja saindo do sistema, suas operações são desabilitadas (linhas 10-11), i.e., nenhuma outra operação é executada pela réplica. Após isso, o estado da aplicação (função $getState$ retorna o estado da réplica) é enviado para uma réplica que esteja entrando no sistema (linhas 12-13) e a visão atualizada é instalada pela réplica (linha 14). A resposta desta operação será a nova visão do sistema (linha 15).

Uma réplica que está entrando no sistema atualiza sua visão e o estado da aplicação (função $setState$ atualiza o estado da réplica) quando receber estas informações de um quórum de leitura (linhas 19-21), i.e., um quórum de leitura enviou o mesmo estado e visão[3]. Quando isso acontece, as operações são habilitadas nesta réplica que passa a executá-las (linhas 1-18). Note que apenas é necessário atualizar o estado das réplicas que estão entrando no sistema, pois as que estavam no sistema já estão atualizadas.

Transferência de Estado. A atualização do estado das réplicas que estão entrando no sistema pode ser fator determinante para o desempenho do protocolo de reconfiguração, pois dependendo do tamanho do estado, um tempo considerável pode ser dispendido nesta tarefa. Tentando diminuir este impacto, o estado pode ser transferido antes ou após a reconfiguração. A transferência antes da reconfiguração apenas alivia o trabalho durante a reconfiguração, pois tudo aquilo que foi modificado desde a transferência deve ser novamente enviado para a réplica. Por outro lado, transferir o estado após a reconfiguração faz com que processos que estejam deixando o sistema precisem aguardar até que o estado das réplicas que entraram seja atualizado, o que ocorrerá apenas após a reconfiguração.

[3]Adicionalmente, pode ser necessário enviar informações dos protocolos subjacentes da RME (ex.: protocolo de ordenação), a fim de que as réplicas sincronizem suas execuções.

Como as operações de clientes apenas serão executadas nas novas réplicas após as mesmas atualizarem seus estados, isso pode causar um atraso na execução das mesmas. No entanto, caso o *membership* da nova visão sofra pequenas alterações, de forma que um quórum de leitura da visão antiga permaneça na nova visão, nenhum atraso será gerado pois existe um número suficiente de réplicas com o estado atualizado aptas a executar e responder às requisições de clientes e, além disso, a reconfiguração será mais rápida pois não ocorre transferência de estado durante a mesma.

Algoritmo 2 RME dinâmica – $execute$ – réplica/servidor i.

variables: Visão atual do processo i
$cv \leftarrow v_0$ // visão mais atual de i

{***Réplicas em cv.***}
upon $execute(\langle operation, v_{req}\rangle)$ //vai executar uma requisição, caso já foram habilitadas na linha 21.
1) **if** $cv \neq v_{req}$ **then** //caso a visão do cliente esteja desatualizada,...
2) **return** $\langle need_update_view, cv\rangle$ //... "envia" a visão atualizada para o cliente.
3) **else**
4) **if** $operation = \langle +, id\rangle \vee operation = \langle -, id\rangle$ **then** //caso for uma operações de reconfiguração ...
5) $new_v.id \leftarrow cv.id + 1$ //... cria a nova visão incrementando seu identificador e ...
6) **if** $operation = \langle +, id\rangle$ **then**
7) $new_v.members \leftarrow cv.members \cup id$ //... atualiza seu *membership* adicionando ou ...
8) **else**
9) $new_v.members \leftarrow cv.members \setminus id$ //... removendo o identificador de alguma réplica.
10) **if** $i \notin new_v$ **then** //caso não esteja na visão atualizada, i.e., saiu do sistema, ...
11) *disable operations* //... desabilita a execução de operações.
12) **if** $operation = \langle +, id\rangle$ **then** //caso uma réplica esteja entrando no sistema, ...
13) $send\langle$STATE-UPDATE, $getState(), new_v, cv\rangle$ to id //... envia o estado para a mesma.
14) $cv \leftarrow new_v$ //então, desativa a visão antiga atualizando cv – instala a nova visão.
15) $response \leftarrow cv$ //a resposta será a visão atualizada.
16) **else** //caso for uma operação "normal" de clientes, ...
17) $response \leftarrow execute(operation)$ //executa a operação na réplica, gerando a resposta.
18) **return** $\langle response, v_{req}\rangle$ //"envia" a resposta para o cliente.

{***Réplicas aguardando para entrar no sistema.***}
upon receipt of $\langle$STATE-UPDATE, $state, new_v, old_v\rangle$ from $old_v.q_l$ servers in old_v
19) $cv \leftarrow new_v$ //quem está entrando instala a visão corrente ...
20) $setState(state)$ //... atualiza o estado da réplica ...
21) *enable operations* //... e habilita a execução de operações.

Lidando com a Dinamicidade do Sistema. Como explicado, clientes e servidores devem ter a informação mais atual sobre a composição do sistema. Este gerenciamento é trivial nos servidores, visto que os mesmos geram e instalam as visões, mantendo-se sempre com a configuração do sistema atualizada. No entanto, manter esta informação atualizada nos clientes é mais difícil e, para isso, os servidores devem informá-los sobre as atualizações na composição do sistema na medida em que operações são requisitadas.

No entanto, um cliente pode permanecer um longo período de tempo sem acessar o sistema, de forma que sua visão não contenha nenhum servidor da visão atual. Neste cenário, não é possível informá-lo que existe uma visão mais atual. Além disso, considerando v a visão atual do sistema, para uma requisição ser ordenada e entregue, a mesma deve atingir pelo menos um servidor correto de v. Consequentemente, um cliente que utiliza uma visão antiga w apenas terá sua requisição ordenada e obterá uma resposta contendo a visão atualizada v, caso $w \cap v$ contenha pelo menos um servidor correto.

Para resolver estes problemas, as visões do sistema podem ser armazenadas também em algum repositório, a partir do qual os clientes podem obtê-las. Para tolerar falhas, as visões podem ser armazenadas em vários repositórios ou em algum serviço tolerante a falhas. Nesta abordagem, sempre que um cliente não tenha sua requisição atendida dentro de um determinado tempo, o mesmo deve verificar no repositório se está utilizando a visão atual do sistema.

5.1. Síntese das Provas de Corretude

Esta seção apresenta um rascunho das provas de que o protocolo apresentado na seção anterior garante as propriedades de uma RME dinâmica (Definição 1).

Lema 1 *Todas as réplicas corretas que executam uma operação possuem o mesmo estado, o qual reflete a execução da mesma sequência de operações.*

Síntese da Prova. Através dos protocolos da RME tradicional [Lamport 1998, Castro and Liskov 2002], dentro de uma mesma visão as operações são executadas seguindo a mesma ordem e, por serem deterministas, geram a mesma alteração no estado das réplicas. Adicionalmente, todas as réplicas da visão inicial começam a execução com o mesmo estado. Desta forma, para provar esta propriedade, basta provar que *(a)* as visões instaladas no sistema formam uma única sequência $V_0 \rightarrow ... \rightarrow V_{i-1} \rightarrow V_i \rightarrow V_{i+1} \rightarrow ...$ e *(b)* o estado final da visão V_i é transferido e usado como estado inicial da visão V_{i+1}. Primeiramente vamos provar *(a)*. Como a execução de uma operação de reconfiguração r para uma visão V_i é ordenada juntamente com as outras operações antes de ser executada, quando da execução de r todos os servidores de V_i possuem o mesmo estado e instalam uma mesma visão seguinte V_{i+1} (linhas 4-14, 19-21, Alg. 2). Nenhuma outra visão diferente de V_{i+1} pode ser gerada para suceder V_i, uma vez que as réplicas que estão saindo do sistema desabilitam suas operações (linha 11, Alg. 2) e as réplicas que ficam no sistema atualizam suas visões para V_{i+1} (linha 14, Alg. 2). Consequentemente, uma única sequência de visões é instalada no sistema. A prova de *(b)* deriva diretamente do fato de que todas as réplicas de V_i possuem o mesmo estado final (quando da execução de r) e enviam (linha 13, Alg. 2) este estado para as réplicas que entram no sistema em V_{i+1} (linhas 19-21, Alg. 2). Consequentemente, o estado inicial de todas as réplicas em V_{i+1} é o estado final das réplicas de V_i. □

Lema 2 *Todas operações de clientes corretos são executadas.*

Síntese da Prova. Considere uma operação o executada por um cliente correto c (linhas 5-10, Alg. 1). Neste caso, o é enviada para todos os processos da visão corrente de c (cv_c), que através das propriedades dos protocolos da RME tradicional [Lamport 1998, Castro and Liskov 2002] acabarão por ordenar e executar o (linhas 1-18, Alg. 2). Desta forma, existem duas possibilidades: *(Caso 1)* cv_c é a visão atual do sistema – a resposta de todos servidores corretos será o resultado da execução de o e cv_c (linhas 17-18. Alg. 2), uma vez que em uma RME as operações são executadas segundo uma ordem, fazendo com que c termine (linhas 9-10, Alg. 1); e *(Caso 2)* cv_c está desatualizada – a resposta de todos servidores corretos será a indicação de que a visão do cliente precisa ser atualizada (linha 2. Alg. 2), o qual a atualiza e volta a enviar a requisição para a RME repetindo este processamento até que a mesma seja executada (linhas 5-9, Alg. 1), consequentemente, como o número de reconfigurações é finito (Suposição 4), cv_c acabará por ficar atualizada e chegamos ao *(Caso 1)*. Como em qualquer um dos casos o é executada, por generalização, todas as operações de clientes corretos são executadas. □

Lema 3 *Se um servidor correto j instala a visão v tal que $i \in v$, então foi invocada a operação $invoke(\langle +, i \rangle)$ ou i é membro da visão inicial.*

Síntese da Prova. Considere que um servidor correto j instala a visão v tal que $i \in v$. Neste caso, i é membro da visão inicial ou j executou a requisição $\langle operation, w \rangle$ (linhas 1-18, Alg. 2) tal que $operation = \langle +, i \rangle$ e $w.id = v.id - 1$. Consequentemente, a operação $invoke(\langle +, i \rangle)$ foi invocada (linhas 1-2, Alg. 1). □

Lema 4 *Se um servidor correto j instala a visão v tal que $i \notin v \wedge (\exists v' : i \in v' \wedge v'.id < v.id)$, então foi invocada a operação $invoke(\langle -, i \rangle)$.*

Síntese da Prova. Considere que um servidor correto j instala a visão v tal que $i \notin v \wedge (\exists v' : i \in v' \wedge v'.id < v.id)$. Neste caso, j executou a requisição $\langle operation, w \rangle$ (linhas 1-18, Alg. 2) tal que $operation = \langle -, i \rangle$ e $w.id = v.id - 1$. Consequentemente, a operação $invoke(\langle -, i \rangle)$ foi invocada (linhas 3-4, Alg. 1). □

Lema 5 *O evento enable operations terminará por ocorrer em todo servidor correto i cuja operação $invoke(\langle +, i \rangle)$ foi requisitada.*

Síntese da Prova. Considere que a operação $invoke(\langle +, i \rangle)$ foi requisitada (linhas 1-2, 5-10, Alg. 1). Pelo Lema 2, os servidores de uma visão v acabarão por executar a requisição $\langle \langle +, i \rangle, v \rangle$ (linhas 1-18, Alg. 2), gerando uma visão $w : w.id = v.id{+}1 \wedge i \in w$. Desta forma, os servidores corretos de v enviam os seus estados e w para i (linha 13, Alg. 2). Pelas suposições 1, 2 e 3, $v.q_l$ servidores corretos enviam seus estados para i, fazendo com que o evento *enable operations* ocorra em i (linhas 19-21, Alg. 2). □

Lema 6 *O evento disable operations terminará por ocorrer em todo servidor correto i cuja operação $invoke(\langle -, i \rangle)$ foi requisitada.*

Síntese da Prova. Considere que a operação $invoke(\langle -, i \rangle)$ foi requisitada (linhas 1-2, 5-10, Alg. 1). Pelo Lema 2, os servidores de uma visão v acabarão por executar a requisição $\langle \langle -, i \rangle, v \rangle$ (linhas 1-18, Alg. 2), gerando uma visão $w : w.id = v.id{+}1 \wedge i \notin w$. Consequentemente, o evento *disable operations* ocorrerá em i (linhas 10-11, Alg. 2). □

6. Experimentos

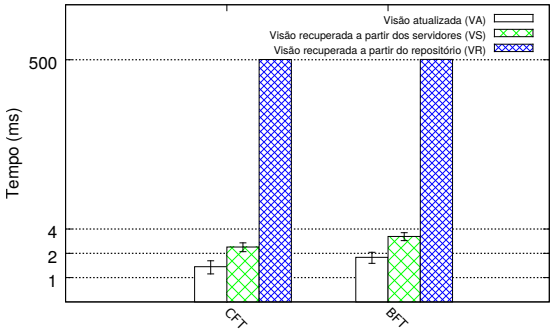
O protocolo proposto foi implementado no BFT-SMART [Alchieri et al. 2013, Bessani et al. 2014], que representa a concretização de uma RME desenvolvida na linguagem de programação Java. O BFT-SMART conta com protocolos para *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME. Para atualização do estado das réplicas que entram no sistema, utilizamos o módulo de *checkpoints* e transferência de estado já implementado pela biblioteca [Bessani et al. 2013], de forma que seguimos então a abordagem de atualização de estado após a reconfiguração.

Visando analisar o desempenho da implementação desenvolvida, bem como o comportamento de uma RME dinâmica, alguns experimentos foram realizados no Emulab [White et al. 2002], em um ambiente consistindo por 6 máquinas *d710* (2.4 GHz 64-*bit* Intel Quad Core Xeon E5530, 12GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. Cada servidor executou em uma máquina separada, enquanto que 10 clientes executaram em outra máquina. O ambiente de *software* utilizado foi o SO Ubuntu 12.04 LTS 64-*bit* com *kernel* 3.2.0-56 e máquina virtual Java de 64 *bits* versão 1.7.0_75.

A aplicação utilizada nos experimentos foi uma lista encadeada que armazena inteiros, sendo que a mesma foi inicializada com $100k$ entradas em cada réplica. Utilizamos nos clientes apenas a operação *Integer get(int index)*, que retorna o elemento da posição indicada[4]. Os clientes selecionam o parâmetro desta operação de forma aleatória, entre 0 e $100k - 1$, seguindo uma distribuição uniforme.

Os experimentos realizados foram: (1) *Latência* – primeiramente foi determinada a latência para a execução de operações considerando diferentes cenários para a visão utilizada pelos clientes; e (2) *Throughput* – também determinamos o *throughput* atingido nos servidores durante uma execução com diferentes reconfigurações que adicionam e removem servidores do sistema. Todos os experimentos tiveram uma fase de *warm-up*.

Latência. A Figura 3 apresenta a latência média, para o sistema configurado tanto como CFT (visão atual cv: $|cv| = 3 \wedge cv.f \leq 1$) quanto BFT (visão atual cv: $|cv| = 4 \wedge cv.f \leq 1$), para os três casos possíveis: *(a)* VA – a visão do cliente está atualizada; *(b)* VS – a visão do cliente está desatualizada e contém servidores que ainda estão no sistema, então a visão é recuperada a partir destes servidores; e *(c)* VR – a visão do cliente está desatualizada e não contém servidores que estão no sistema, então após um *timeout* (configurado em 500 ms) a visão é recuperada a partir do repositório. Neste experimento, utilizamos como repositório um diretório compartilhado na rede por todos os processos. A latência foi medida em um dos clientes (sempre o mesmo) e os valores apresentados representam a média de 1000 execuções, excluindo-se 10% dos valores com maior desvio. Em nenhuma das execuções o desvio padrão ultrapassou 0.32 ms.



		Atualização	Execução	Total
CFT	VA	0	1.364	1.364
	VS	1.252	1.144	2.396
	VR	501.456	1.867	503.323
BFT	VA	0	1.782	1.782
	VS	1.648	1.590	3.238
	VR	501.604	2.701	504.305

Figura 3. Latência (tempos em ms) quando o cliente possui a visão atualizada (VA) ou a recupera a partir dos servidores (VS) ou do repositório (VR).

Quando o cliente possui a visão atualizada o tempo apresentado reflete a execução normal de uma operação no sistema. Para o caso da visão ser recuperada a partir dos servidores, o tempo apresentado reflete a execução de duas operações: uma para atualizar a visão (a resposta da primeira tentativa de execução é a visão atualizada – Algorithm 2) e outra para efetivamente executar a operação. Já para o terceiro caso, o tempo apresentado contempla: o envio da requisição para os servidores; a espera pelo *timeout*; a leitura e atualização da visão a partir do repositório; e a execução da operação. A Figura 3 (direita) apresenta uma tabela com os tempos gastos para atualizar a visão e para efetivamente executar a operação após a visão estar atualizada. Como podemos perceber, no caso *(b)* a latência é aproximadamente o dobro do caso *(a)*, enquanto que no caso *(c)* a latência sofre grande impacto do *timeout* para aguardo das respostas.

[4]Utilizamos apenas esta operação porque o objetivo destes experimentos não é analizar a aplicação lista em si, mas o comportamento de uma RME dinâmica em diversos cenários.

Throughput. A Figura 4 mostra o *throughput* obtido pelo sistema, configurado como CFT, considerando eventos de entradas e saídas de réplicas em uma execução de 300 segundos. O *throughput* foi medido em um dos servidores (o líder do protocolo de ordenação [Bessani et al. 2014]). Inicialmente, o sistema é suportado por 3 réplicas, i.e., $v_0 = \{0, 1, 2\}$. As réplicas 3 e 4 são inseridas no sistema após 60 e 120 segundos, respectivamente, permanecendo no sistema por 120 segundos, i.e., a réplica 3 deixou o sistema no segundo 180 e a réplica 4 no segundo 240. Desta forma, o sistema apresentou as seguintes visões de acordo com o intervalo de tempo: $0 - 60$ ($v_0 = \{0, 1, 2\}$, $v_0.f \leq 1$); $60-120$ ($v_1 = \{0, 1, 2, 3\}$, $v_1.f \leq 1$); $120-180$ ($v_2 = \{0, 1, 2, 3, 4\}$, $v_2.f \leq 2$); $180-240$ ($v_3 = \{0, 1, 2, 4\}$, $v_3.f \leq 1$); e $240 - 300$ ($v_4 = \{0, 1, 2\} = v_0$, $v_4.f \leq 1$).

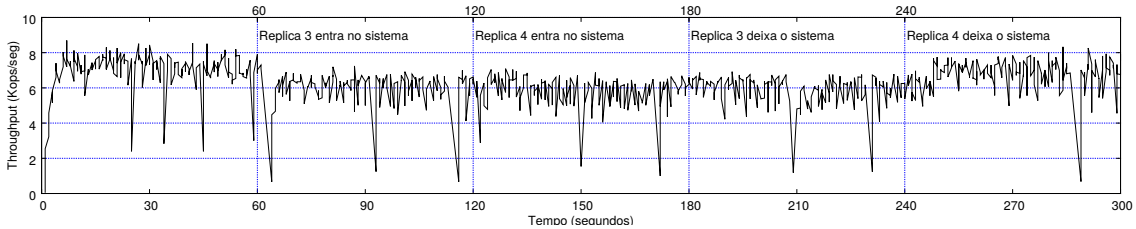


Figura 4. Throughput.

Conforme os clientes iniciam suas execuções, o *throughput* vai aumentando até que todos estejam em execução, o que acontece por volta dos 5 segundos. A partir do segundo 60, o desempenho do sistema sofreu uma pequena queda devido ao aumento do número de réplicas e do tamanho dos quóruns de escrita (de 2 para 3 réplicas). Após isso, o *throughput* não teve variação significativa com a entrada da réplica 4 ou com a saída da réplica 3, pois o tamanho dos quóruns se manteve constante em 3. Finalmente, com a saída da réplica 4 o sistema voltou a ter apenas 3 réplicas (e quóruns de escrita de tamanho 2), apresentando um desempenho semelhante ao inicial. Outro ponto a destacar é que de tempos em tempos o sistema apresentou uma queda brusca de desempenho devido à execução do protocolo de *checkpoints* [Bessani et al. 2013, Bessani et al. 2014].

7. Conclusões

Este artigo apresentou uma proposta para especificação de uma RME dinâmica, onde o conjunto de réplicas que suporta o sistema pode sofrer alterações em tempo de execução. Definimos três fases para o processo de reconfiguração e identificamos as quantidades de réplicas que devem permanecer no sistema em cada fase. Além disso, definimos as propriedades para uma RME dinâmica e apresentamos um protocolo para satisfazê-las. O protocolo proposto foi implementado e os resultados dos experimentos mostram o comportamento prático de uma RME com suporte a 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., Bessani, A. N., and da Silva Fraga, J. (2013). Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas*.

Alchieri, E. A. P., Bessani, A. N., Greve, F., and da Silva Fraga, J. (2014). Reconfiguração modular de sistemas de quóruns. In *Anais do XXXII Simpósio Brasileiro de Redes de Computadores*.

Bessani, A., Santos, M., Felix, J. a., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proc. of the International Conference on Dependable Systems and Networks*.

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

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riché, T. (2009). UpRight cluster services. In *Proc. of the ACM SOSP'09*.

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

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programing Languages and Systems*, 12(3):463–492.

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

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41:63–73.

Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. (2006). The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*.

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

Santos, N. and Schiper, A. (2013). Achieving high-throughput State Machine Replication in multi-core systems. In *Proc. of the IEEE ICDCS'13*.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

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 Symp. on Operating Systems Design and Implementations*.