Replicação Máquina de Estados Paralela e Reconfigurável

Alex Lobo[1] **, Eduardo Alchieri**[1] **, Fernando Pedone**[2] **,**
Fernando Dotti[3] **, Odorico Mendizabal**[4]

[1] Departamento de Ciência da Computação – Universidade de Brasília
[2] Faculdade de Informática – Universidade de Lugano
[3] Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul
[4] Centro de Ciências Computacionais – Universidade Federal do Rio Grande*

Abstract. *State Machine Replication (SMR) is an approach used to implement fault-tolerant systems. In this approach, servers are replicated and client requests are deterministically executed in the same order by all replicas. Consequently, client requests must be ordered and sequentially executed by every replica. To improve system performance in multicore systems, parallel SMR allows parallel execution of requests, according to the degree of parallelism definied at startup. However, some requests still need sequential execution, impacting the system performance since additional synchronization is needed. This work proposes a protocol to reconfigure the degree of parallelism in parallel SMR. The protocol aims to improve performance by taking into account the workload. Experiments show the gains due to reconfiguration and shed some light on the behaviour of parallel SMR.*

Resumo. *A Replicação Máquina de Estados (RME) é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas. Esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de requisições. Para isso, as requisições devem ser ordenadas e executadas sequencialmente segundo esta ordem em todas as réplicas. Visando melhorar o desempenho do sistema em arquiteturas com múltiplos núcleos, RMEs paralelas tiram proveito da semântica das requisições e permitem a execução paralela de algumas delas, de acordo com um grau de paralelismo pré-definido. Porém, algumas requisições continuam precisando de execução sequencial e impactam negativamente o desempenho do sistema, visto que sincronizações adicionais são necessárias, de acordo com o grau de paralelismo. Este trabalho propõe um protocolo para RME paralela e com grau de paralelismo reconfigurável de acordo com o workload atual, visando tirar proveito em situação favoráveis e impactar o mínimo possível em situações desfavoráveis. Experimentos mostram os ganhos advindos com as reconfigurações e ajudam a elucidar o funcionamento deste tipo de sistema.*

1. Introdução

A Replicação Máquina de Estados (RME) [Schneider 1990] é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas [Lamport 1998, Schneider 1990, Castro and Liskov 2002]. Basicamente, esta técnica consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo

*Este trabalho recebeu apoio da CAPES através do projeto Scalable Dependability (88881.062190/2014-01) e do CNPq através do projeto FreeStore (457272/2014-7).

conjunto de operações requisitadas por clientes, fornecendo um serviço de replicação com consistência forte (*linearizability*) [Herlihy and Wing 1990].

Para manter o determinismo da execução, as operações são ordenadas e executadas sequencialmente seguindo a mesma ordem em todas as réplicas, o que limita o desempenho do sistema principalmente quando consideramos servidores atuais que possuem processadores com múltiplos núcleos, pois apenas um deles seria utilizado para a execução das operações. Com o objetivo de contornar esta limitação, recentemente surgiram abordagens que, tirando proveito da semântica das operações, empregam protocolos que suportam a execução paralela de algumas operações [Kotla and Dahlin 2004, Marandi et al. 2014, Marandi and Pedone 2014, Alchieri 2015, Zbierski 2015].

Estas abordagens, chamadas de RME paralelas, classificam as requisições em dependentes (ou conflitantes) e independentes (ou não conflitantes), de modo que as requisições independentes são executadas em paralelo nas réplicas. Já requisições dependentes devem ser executadas sequencialmente, o que exige alguma sincronização nas réplicas pois nenhuma outra operação pode ser executada paralelamente. Duas requisições são independentes quando acessam diferentes variáveis ou quando apenas leem o valor de uma mesma variável. Por outro lado, duas requisições são dependentes quando acessam pelo menos uma mesma variável e pelo menos uma das requisições altera o valor desta variável.

A quantidade de requisições independentes executadas em paralelo é configurada de acordo com o grau de paralelismo (número de *threads* de execução) definido na inicialização do sistema. Um número elevado aumenta o desempenho em *workloads* com requisições predominantemente independentes, mas impacta negativamente caso a predominância seja de operações dependentes, devido a necessidade de sincronizações adicionais. Por outro lado, um número baixo não tira proveito de *workloads* favoráveis (com predominância de requisições independentes) [Marandi et al. 2014, Alchieri 2015]. Desta forma, apesar de ser uma técnica muito promissora, configurar o grau de paralelismo é um problema complexo mesmo conhecendo-se o *workload a priori*, o que geralmente não é possível. Além disso, o *workload* geralmente sofre variações durante a execução do sistema [Le et al. 2016]. Com o objetivo de contornar estas limitações, as principais contribuições deste trabalho são:

- Proposta de um protocolo para RME paralela que melhora os protocolos anteriormente propostos, notadamente [Marandi et al. 2014, Marandi and Pedone 2014], por permitir a definição de grupos intermediários de dependências, além dos grupos de dependentes e independentes, aumentando o paralelismo na execução.
- Proposta de extensão do protocolo anterior para suportar reconfiguração no grau de paralelismo (número de *threads* de execução), possibilitando a adaptação ao *workload* atual, sendo que a principal vantagem desta técnica é que nenhuma informação preliminar se faz necessária.
- Apresentação e análise de uma série de experimentos realizados com uma implementação dos protocolos propostos, possibilitando uma melhor compreensão a respeito do funcionamento de uma RME paralela, bem como dos ganhos advindos com a sua reconfiguração.

O restante deste artigo está organizado da seguinte forma. A Seção 2 discute os conceitos envolvendo uma RME e as abordagens para melhorar seu desempenho. A Seção 3 apresenta os protocolos para RME paralela e reconfigurável. A Seção 4 analisa

alguns experimentos realizados. As conclusões do trabalho são apresentadas na Seção 5.

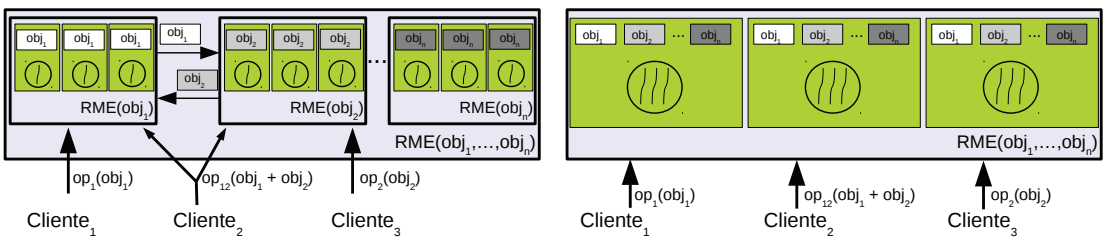
2. Replicação Máquina de Estados

A Replicação Máquina de Estados (RME) [Schneider 1990] é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas [Lamport 1998, Schneider 1990, Castro and Liskov 2002] e consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados. São necessário $2f + 1$ ou $3f + 1$ réplicas para tolerar até f falhas por *crash* ou bizantinas, respectivamente.

Para que as réplicas apresentem a mesma evolução em seus estados, é necessário que: *(i)* partindo de um mesmo estado inicial e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* todas as réplicas cheguem ao mesmo estado final, definindo o determinismo de réplicas. Para prover o item *(i)* 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), procedimento que pode não ser trivial se considerarmos a possibilidade de recuperação de réplicas falhas [Bessani et al. 2014].

Já garantir o item *(ii)* envolve a utilização de um protocolo de difusão atômica [Hadzilacos and Toueg 1994], também conhecido como difusão com ordem total, que possibilita que todas as réplicas corretas entreguem todas as requisições na mesma ordem. Finalmente, para prover o item *(iii)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação (com os mesmos parâmetros) produza a mesma mudança de estado e tenha o mesmo retorno nas diversas réplicas do sistema.

Muito tem se estudado sobre como prover o item *(ii)*, visto que o problema da difusão atômica é equivalente ao do consenso [Hadzilacos and Toueg 1994] (os processos devem entrar em acordo – propriedade fundamental do consenso – acerca da ordem de entrega das mensagens/requisições) sendo geralmente tratado como o gargalo de uma RME por exigir vários passos de comunicação. Em cada passo, uma ou mais réplicas enviam/recebem mensagens para/de uma outra ou mais réplicas. Tanto o número de passos de comunicação quanto a quantidade de mensagens enviadas/recebidas em cada passo variam de acordo com o protocolo utilizado [Castro and Liskov 2002, Abd-El-Malek et al. 2005, Lamport 2006, Marandi et al. 2010, Cowling et al. 2006, Kotla et al. 2009, Guerraoui et al. 2010].



(a) RME Escalável. (b) RME Paralela.

Figura 1. Abordagens para aumentar o desempenho de uma RME.

Um aspecto que recentemente começou a ser explorado é a otimização dos procedimentos necessários para a execução das requisições, modificando a visão sobre a forma

de como o item *(iii)* pode ser atendido. Estas abordagens tiram proveito da semântica das operações para *(1)* dividir o estado da aplicação entre várias RMEs (RME escalável – Figura 1(a)) [Bezerra et al. 2014, Le et al. 2016, da Silva Böge et al. 2016] ou *(2)* executá-las paralelamente nas réplicas (RME Paralela – Figura 1(b)) [Kotla and Dahlin 2004, Marandi et al. 2014, Marandi and Pedone 2014, Alchieri 2015].

RME Escalável: A ideia desta abordagem é dividir o estado da aplicação entre várias RMEs (ou partições) que executam em paralelo (Figura 1(a)). As operações são mapeadas para a partição correspondente e o desempenho é melhorado visto que tanto a ordenação quanto a execução de operações enviadas para RMEs diferentes são executadas em paralelo. No entanto, uma operação que precisa acessar o estado de mais de uma partição deve ser enviada para as várias partições envolvidas e os objetos (estado) devem ser transferidos entre as partições, a fim de que uma delas execute a operação e as outras apenas aguardem esta execução [Bezerra et al. 2014]. Com o objetivo de evitar a execução de operações que envolvem várias partições e a consequente queda de desempenho, o estado da aplicação pode ser redistribuído entre as partições [Le et al. 2016] ou ainda partições podem ser criadas ou removidas [da Silva Böge et al. 2016].

RME Paralela: Em uma RME paralela, cada réplica possui o estado completo da aplicação como em uma RME tradicional, mas operações que acessam partes diferentes do estado (ou partições) são executadas em paralelo através de um conjunto de *threads* de execução (Figura 1(b)). Para isso, duas soluções foram propostas: *(1)* partes do estado da aplicação são associadas a diferentes *threads* e quando uma operação envolve o estado manipulado por várias *threads*, uma delas executa e as outras apenas aguardam [Marandi et al. 2014, Alchieri 2015]; ou *(2)* um grafo de dependências é criado e as *threads* buscam requisições independentes neste grafo para execução [Kotla and Dahlin 2004]. Como podemos perceber, os efeitos negativos causados pela execução de operações que envolvem várias partições tendem a ser menores nesta abordagem visto que não é necessária a transferência de objetos entre partições. Como nas RMEs escaláveis, qualquer técnica utilizada para melhorar o desempenho do protocolo de ordenação pode ser empregada, como a ordenação em lotes [Bessani et al. 2014] ou a execução paralela de várias instâncias do protocolo de ordenação [Marandi et al. 2014].

3. Replicação Máquina de Estados Paralela e Reconfigurável

Esta seção apresenta a nossa proposta para uma RME paralela e reconfigurável. Primeiramente, discutimos o protocolo para execuções paralelas, o qual faz uso de um escalonador (*scheduler*) que atribui as requisições para as *threads* de execução. Posteriormente, apresentamos os protocolos que permitem a reconfiguração do número de *threads* ativas de acordo com políticas de reconfiguração que podem ser definidas pelos usuários.

3.1. Paralelismo

Esta seção apresenta o protocolo para execuções paralelas de requisições em uma réplica, requisito de uma RME paralela (Figura 1(b)). O protocolo aqui apresentado é independente do protocolo de ordenação, o qual também pode executar várias instâncias em paralelo (ordenação em paralelo) [Marandi et al. 2014] e/ou ordenar lotes de requisições em uma única instância (ordenação em lotes) [Bessani et al. 2014].

A ideia deste protocolo é dividir o estado da aplicação entre um conjunto de *threads* de execução, de forma que cada *thread* acesse a sua partição do estado para executar operações em paralelo. Sempre que uma operação envolve o estado de várias *threads*, uma delas acessa todo o estado necessário para executar a operação e as outras *threads* envolvidas aguardam esta execução, garantindo as propriedades de uma RME [Bezerra et al. 2014, Marandi et al. 2014].

Esta divisão do estado define grupos de execução (ou de dependências) e o cliente deve informar para qual grupo de execução sua requisição é endereçada. Em nosso protocolo é possível criar qualquer grupo de execução necessário para a aplicação (basta configurar as réplicas com o identificador do grupo e as *threads* que o compõem), mas os seguintes grupos são definidos por padrão: *(i)* `CONFLICT_ALL` – contém todas as *threads* e deve ser usado para realizar operações que acessam todo o estado da aplicação; *(ii)* `CONFLICT_NONE` – usado para realizar operações que podem executar em paralelo com qualquer outra operação (geralmente operações que não modificam o estado), por qualquer *thread*; *(iii)* cada *thread* ainda possui um grupo com seu próprio identificador (as *threads* são identificadas de forma incremental começando em 0) que deve ser usado para operações que acessam apenas o estado associado à respectiva *thread* (este mapeamento é definido pelo cliente, i.e., operações que acessam uma mesma partição do estado devem sempre ser enviadas para a mesma *thread*).

Algoritmo 1 Algoritmo de distribuição das requisições (*thread scheduler*).

variables: Variables and sets used by the scheduler.
$numThreads \leftarrow$ the number of worker threads
$queues[numThreads] \leftarrow$ the queues used to assign requests to the threads (see Algorithm 2)
$nextThread \leftarrow 0$ // id of the thread that will execute the next `CONFLICT_NONE` request
on initialization:
1) start the barrier for the `CONFLICT_ALL` group with number of parties equal to $numThreads$
2) start the barriers for other created conflict groups
on A-deliver(request):
3) **if** $request.groupId == $ `CONFLICT_NONE` **then**
4) $queues[nextThread].put(request)$ //assigns the request to a thread...
5) $nextThread = (nextThread + 1)\%numThreads$ //...using a roud-robin policy
6) **else if** $request.groupId == $ `CONFLICT_ALL` **then**
7) **for** $i = 0; i < numThreads; i++$ **do** //assigns the request to all threads
8) $queues[i].put(request)$
9) **end for**
10) **else if** $request.groupId < numThreads$ **then** //request directly sent to some thread
11) $queues[request.groupId].put(request)$
12) **else** //request to a created conflict group
13) $group\_queues \leftarrow$ get the queues of the threads in $request.groupId$
14) **for** $each\ q \in group\_queues$ **do** //assigns the request to the threads in the group
15) $q.put(request)$
16) **end for**
17) **end if**

Escalonador. Sempre que uma requisição é entregue pelo protocolo de ordenação, o escalonador é executado (por uma outra *thread*) para atribuir uma determinada requisição a uma ou mais *threads* de execução (Algoritmo 1). A comunicação entre o escalonador e as *threads* de execução ocorre através de uma fila sincronizada, de acordo com o identificador do grupo de execução conforme segue:

- `CONFLICT_NONE` – a requisição é atribuída a uma única *thread* de execução, seguindo a política *round-robin* (linhas 3-5).
- `CONFLICT_ALL` – a requisição é atribuída a todas as *threads* (linhas 6-9).
- Caso a requisição seja endereçada para o grupo de uma *thread* específica, a mesma é atribuída para tal *thread* (linhas 10-12).
- Finalmente, caso a requisição seja endereçada para um grupo formado por mais de uma *thread*, a requisição é encaminhada para as *threads* do grupo (linhas 13-16).

Threads Executoras. Sempre que existir uma requisição disponível para ser executada, cada *thread* procede da seguinte forma, de acordo com o grupo ao qual a requisição foi endereçada (Algoritmo 2):

- `CONFLICT_NONE` ou seu próprio grupo – apenas executa a operação, pois nenhuma sincronização com outras *threads* é necessária (linhas 3-4).
- `CONFLICT_ALL` ou outro grupo – neste caso é necessário que ocorra uma sincronização entre as *threads* do grupo (linhas 5-14), de forma que primeiro é necessário esperar até que todas as *threads* cheguem a este ponto da execução (primeiro acesso à barreira – linhas 7 e 11) para então uma delas executar a requisição (linha 6) enquanto que as outras apenas aguardam por isso (segundo acesso à barreira – linhas 9 e 12). A função $getBarrier(request.groupId).await()$ sinaliza que uma determinada *thread* atingiu a barreira do grupo $request.groupId$.

Algoritmo 2 Algoritmo de execução de requisições (*threads* executoras).

variables: Variables and sets used by each worker thread.
$myId \leftarrow$ id received at initialization // thread id (the ids range from 0 to $maxThreads - 1$)
$queue \leftarrow$ a synchronized/blocking queue that contains the requests to be executed by this thread
on thread run:
1) **while** true **do**
2) $request \leftarrow queue.take()$ //get the next request to be executed, blocks until a request be available
3) **if** $request.groupId ==$ `CONFLICT_NONE` $\vee$ $req.groupId == myId$ **then** // no conflict
4) executes the request against the application state and sends the reply to the client
5) **else** //conflict: $request.groupId ==$ `CONFLICT_ALL` or some other created conflict group
6) **if** $myId == executor\ for\ request.groupId$ **then** // the thread with the smallest id
7) $getBarrier(request.groupId).await()$ // waits the other threads to stop
8) executes the request against the application state and sends the reply to the client
9) $getBarrier(request.groupId).await()$ // resumes the other threads execution
10) **else**
11) $getBarrier(request.groupId).await()$ //signalizes that will wait for the execution
12) $getBarrier(request.groupId).await()$ //waits the execution
13) **end if**
14) **end if**
15) **end while**

3.2. Reconfiguração

Apesar do protocolo da seção anterior implementar uma RME paralela, o número de *threads* de execução, que define o grau de paralelismo, é definido de forma estática na inicialização do sistema. Esta definição é muito importante e afeta diretamente o desempenho do sistema [Marandi et al. 2014, Alchieri 2015]. Um número grande de *threads* tende a aumentar o desempenho em *workloads* com uma grande percentagem de requisições não conflitantes (`CONFLICT_NONE`), mas o desempenho diminui abruptamente com o aumento da quantidade de requisições conflitantes (`CONFLICT_ALL`). Por

outro lado, um número pequeno de *threads* não faz com que o sistema seja capaz de atingir o desempenho máximo para *workloads* com poucas requisições conflitantes. O desempenho máximo do sistema ainda é limitado pelo *hardware* utilizado.

Algoritmo 3 Algoritmo reconfigurável de distribuição das requisições (*thread scheduler*).

variables: Variables and sets used by the scheduler.
$minThreads \leftarrow$ the minimum number of active worker threads
$maxThreads \leftarrow$ the maximum number of active worker threads
$currentThreads \leftarrow$ the current number of active worker threads
$queues[maxThreads] \leftarrow$ the queues used to assign requests to the threads (see Algorithm 4)
$nextThread \leftarrow 0$ // id of the thread that will execute the next `CONFLICT_NONE` request
on initialization:
1) start the barrier for `CONFLICT_ALL` group with number of parties equal to $currentThreads$
2) start the barriers for other created conflict groups
3) start the barrier $reconfig\_barrier$ for `RECONFIG` with number of parties equal to $maxThreads$
on A-deliver(request):
4) $recNum \leftarrow reconfigPolicy(request, minThreads, currentThreads, maxThreads)$
5) **if** $recNum \neq 0 \wedge (minThreads \leq currentThreads + recNum \leq maxThreads)$ **then**
6) $currentThreads \leftarrow currentThreads + recNum$
7) $nextThread \leftarrow 0$
8) $reconfig\_request.groupId \leftarrow$ `RECONFIG`
9) **for** $i = 0; i < maxThreads; i++$ **do** //assigns the request to all threads
10) $queues[i].put(reconfig\_request)$
11) **end for**
12) **end if**
13) **if** $request.groupId ==$ `CONFLICT_NONE` **then**
14) $queues[nextThread].put(request)$ //assigns the request to a thread...
15) $nextThread = (nextThread + 1)\%currentThreads$ //...using a roud-robin policy
16) **else if** $request.groupId ==$ `CONFLICT_ALL` **then**
17) **for** $i = 0; i < currentThreads; i++$ **do** //assigns the request to all active threads
18) $queues[i].put(request)$
19) **end for**
20) **else if** $request.groupId < maxThreads$ **then** //request directly sent to some thread
21) **if** $request.groupId < currentThreads$ **then**
22) $queues[request.groupId].put(request)$
23) **else**
24) $request.groupId \leftarrow$ `CONFLICT_ALL` //handles the request as `CONFLICT_ALL`
25) **for** $i = 0; i < currentThreads; i++$ **do** //assigns the request to all active threads
26) $queues[i].put(request)$
27) **end for**
28) **end if**
29) **else** //request to a created conflict group
30) **if** some thread belonging to $request.groupId$ is not active **then**
31) $request.groupId \leftarrow$ `CONFLICT_ALL` //handles the request as `CONFLICT_ALL`
32) **for** $i = 0; i < currentThreads; i++$ **do** //assigns the request to all active threads
33) $queues[i].put(request)$
34) **end for**
35) **else**
36) $group\_queues \leftarrow$ get the queues of the threads in $request.groupId$
37) **for** $each\ q \in group\_queues$ **do** //assigns the request to the threads in the group
38) $q.put(request)$
39) **end for**
40) **end if**
41) **end if**

Devido ao fato de o *workload* poder mudar durante a execução, e dadas as restrições anteriormente descritas, definir um número ideal para a quantidade de *threads* de execução é uma tarefa bastante difícil, se não impossível, e ao mesmo tempo fundamental para o desempenho do sistema. Neste sentido, esta seção apresenta um protocolo de reconfiguração para uma RME paralela, de forma que o número de *threads* possa sofrer alterações durante a execução da aplicação para tentar se adaptar ao *workload* corrente.

Para isso, este protocolo divide as *threads* em ativas e inativas. Uma *thread* é ativa caso esteja apta a executar requisições, caso contrário é considerada inativa. Somente as *threads* ativas participam dos protocolos da RME paralela e, com isso, afetam o desempenho do sistema. Desta forma, a reconfiguração ocorre através da ativação e desativação de *threads*. Na inicialização do sistema, além de especificar o número inicial de *threads* ativas, o usuário também deve definir os números mínimo e máximo de *threads* que podem estar ativas ao mesmo tempo e fornecer uma política com as regras a serem seguidas para ativação e/ou desativação de *threads*.

Escalonador. Sempre que uma requisição é entregue pelo protocolo de ordenação o escalonador é executado (Algoritmo 3). Primeiramente, a política de reconfiguração é acessada para verificar a necessidade de reconfiguração (linha 4), a qual deve retornar o número de *threads* a serem ativadas (retorno positivo) ou desativadas (retorno negativo). As *threads* ativadas/desativadas sempre serão as de maiores identificadores. Caso seja necessário reconfigurar o sistema (linhas 5-11), uma requisição especial (`RECONFIG`) é adicionada na fila de todas as *threads* (até mesmo as inativas, que também participam da reconfiguração) e o escalonador passa a atribuir requisições para as *threads* ativadas ou a não atribuir mais requisições para as *threads* que serão desativadas. Desta forma, para ativar ou desativar uma *thread* basta começar a adicionar ou parar de adicionar requisições em sua fila, respectivamente. A reconfiguração propriamente dita ocorre quando todas as *threads* atingem o mesmo ponto da execução onde esta requisição especial é executada.

O restante deste protocolo é semelhante ao do escalonador estático (Algoritmo 1). A principal diferença é que apenas as *threads* ativas são consideradas na distribuição das requisições e caso uma requisição seja endereçada para uma *thread* inativa ou para um grupo que a contenha, a mesma é considerada como uma requisição que conflita com todas (`CONFLICT_ALL`) pois de outro modo não seria executada. Requisições endereçadas para grupos que possuem alguma *thread* inativa poderiam ser executadas pelo próprio grupo caso o mesmo fosse reconfigurado (sua barreira reconfigurada para o novo número de *threads* ativas pertencentes ao grupo).

Threads Executoras. A principal diferença para o modelo estático está na execução de reconfigurações (Algoritmo 4), as quais são tratadas de forma semelhante às requisições do grupo `CONFLICT_ALL`, i.e., sem paralelismo. Durante a execução de uma reconfiguração, a barreira do grupo `CONFLICT_ALL` é reconfigurada de acordo com a nova configuração do sistema (linha 6). Todas as *threads* devem participar desta execução pois a barreira utilizada para reconfigurações ($reconfig\_barrier$) deve ser estática (não pode ser reconfigurada em meio a execução da reconfiguração). Como é esperado que a quantidade de reconfigurações seja proporcionalmente muito menor do que a quantidade de requisições de clientes, isso não afeta o desempenho do sistema.

Esta abordagem funciona pelo fato de que até a determinação da reconfiguração pelo escalonador, todas as *threads* ativas na configuração antiga receberam as requisições

e portanto a barreira com a configuração antiga é utilizada até que todas cheguem ao ponto da execução da reconfiguração. A partir da determinação da reconfiguração, o escalonador passa a atribuir requisições apenas para as *threads* ativas na nova configuração e, do ponto da execução da reconfiguração em diante, a barreira estará reconfigurada para refletir este número atual de *threads* ativas.

Algoritmo 4 Algoritmo reconfigurável de execução de requisições (*threads* executoras).

variables: Variables and sets used by each worker thread.
$myId \leftarrow$ id received at initialization // thread id (the ids range from 0 to $maxThreads - 1$)
$queue \leftarrow$ a synchronized/blocking queue that contains the requests to be executed by this thread
on thread run:
1) **while** true **do**
2) $request \leftarrow queue.take()$ //get the next request to be executed, blocks until a request be available
3) **if** $request.groupId ==$ `RECONFIG` **then** // thread reconfiguration
4) **if** $myId == 0$ **then**
5) $reconfig\_barrier.await()$ //waits all the other threads to stop
6) reconfigure the barrier for `CONFLICT_ALL` with number of parties equal to $currentThreads$
7) $reconfig\_barrier.await()$ //resumes all the other threads
8) **else**
9) $reconfig\_barrier.await()$ //signalizes that will wait for the reconfiguration
10) $reconfig\_barrier.await()$ //waits the reconfiguration execution
11) **end if**
12) **end if**
13) lines 3–14 of Algorithm 2
14) **end while**

3.2.1. Políticas de Reconfiguração

A definição da nova configuração do sistema e de quando adotá-la segue uma política de reconfiguração (linha 4 – Algoritmo 3), que pode ser especificada pelo usuário. A política considera a requisição que está sendo escalonada, o número atual de *threads* ativas, além das configurações para o número mínimo e máximo de *threads* ativas.

Algoritmo 5 Política de reconfiguração.

variables: Variables and sets used.
$conflict \leftarrow 0$ // counter for the number of conflict requests
$notConflict \leftarrow 0$ // counter for the number of non conflict requests
$period \leftarrow 10000$ // period (number of requests) to check for reconfigurations
reconfigPolicy(request,minThreads,currentThreads,maxThreads)
1) **if** $request.groupId ==$ `CONFLICT_ALL` **then**
2) $conflict++$
3) **else**
4) $notConflict++$
5) **end if**
6) **if** $(conflict + notConflict) == period$ **then**
7) $percent \leftarrow conflict * 100/period$
8) $conflict \leftarrow 0$; $notConflict \leftarrow 0$
9) **if** $(percent \leq 20) \wedge ((currentThreads + 1) \leq maxThreads)$ **then**
10) **return** 1
11) **else if** $(percent > 20) \wedge ((currentThreads - 1) \geq minThreads)$ **then**
12) **return** -1
13) **end if**
14) **end if**
15) **return** 0

O Algoritmo 5 apresenta um exemplo de política, a qual estabelece que a cada 10.000 requisições é determinado o percentual de requisições conflitantes. Caso o *workload* apresente uma quantidade de até 20% de requisições conflitantes, uma *thread* é ativada até que a quantidade máxima seja atingida. Caso contrário, uma *thread* é desativada até que a quantidade mínima seja atingida. Com um *workload* acima de 20% conflitantes, gasta-se muito tempo sincronizando as *threads* (linhas 5-14 – Algoritmo 2) e geralmente o desempenho é melhor em uma execução sequencial [Marandi et al. 2014, Alchieri 2015].

4. Experimentos

Visando analisar o desempenho das soluções propostas, bem como o comportamento de uma RME com execuções paralelas e reconfigurável, os protocolos propostos foram implementados no BFT-SMART [Bessani et al. 2014] e alguns experimentos foram realizados no Emulab [White et al. 2002]. O BFT-SMART representa a concretização de uma RME, desenvolvida na linguagem de programação Java, onde uma única instância do protocolo de ordenação é executada por vez para ordenar um lote de requisições. O principal objetivo destes experimentos não é determinar os valores máximos de desempenho, mas sim analisar as diferenças entre as abordagens e determinar os ganhos advindos com a possibilidade de reconfiguração.

Aplicação utilizada: Lista Encadeada. Esta aplicação foi implementada nos servidores através de uma lista encadeada (*LinkedList*), que é uma estrutura de dados não sincronizada, i.e., caso duas ou mais *threads* acessem esta estrutura concorrentemente e pelo menos uma delas modifique a sua estrutura, então estes acessos devem ser sincronizados. Neste experimento, a lista foi utilizada para armazenar inteiros (*Integer*) e as seguintes operações foram implementadas para seu acesso: *boolean add (Integer i)* – adiciona i no final da lista e retorna *true* caso i ainda não esteja na lista, retorna *false* caso contrário; *boolean remove(Integer i)* – remove i e returna *true* caso i esteja na lista, retorna *false* caso contrário; *Integer get(int index)* – retorna o elemento da posição indicada; e *boolean contains(Integer i)* – retorna *true* caso i esteja na lista, retorna *false* caso contrário. Note que todas estas operações possuem um custo de $O(n)$, onde n é o tamanho da lista. As seguintes dependências foram definidas para estas operações: add e $remove$ são dependentes de todas as outras operações (`CONFLICT_ALL`), enquanto que get e $contains$ não possuem dependências (`CONFLICT_NONE`), somente as já definidas com add e $remove$.

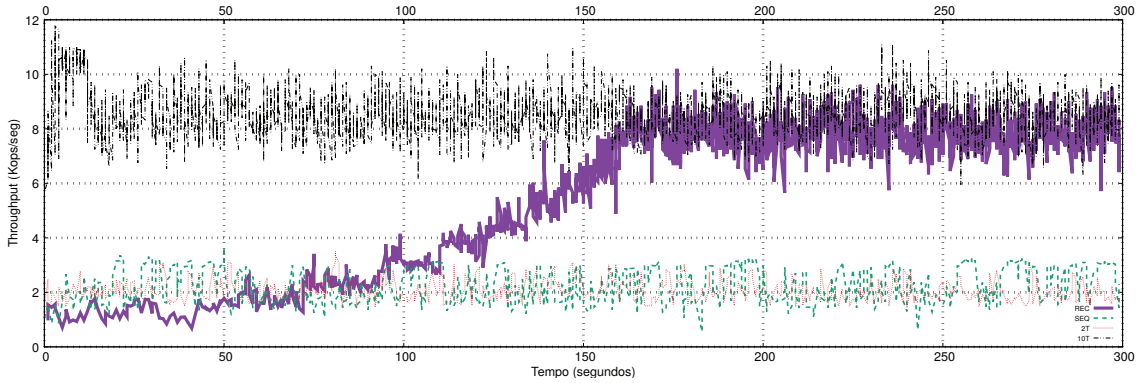
Configuração dos Experimentos. O ambiente para os experimentos foi constituído por 6 máquinas *d430* (2.4 GHz E5-2630v3, com 8 núcleos e 2 *threads* por núcleo, 64GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O BFT-SMART foi configurado com 3 servidores para tolerar até uma falha por parada (*crash*). Cada servidor executou em uma máquina separada, enquanto que 90 clientes foram distribuídos uniformemente nas outras 3 máquinas. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 14 64-*bit* e máquina virtual Java de 64 *bits* versão 1.7.0_75.

Nos experimentos, a lista foi inicializada com $100k$ entradas em cada réplica e utilizamos as operações add e $contains$ para execução de operações `CONFLICT_ALL` e `CONFLICT_NONE`, respectivamente. O parâmetro destas operações sempre foi o último elemento da lista ($100k-1$) de forma que em todos os experimentos foi possível executar o mesmo conjunto de operações com os mesmos parâmetros. Para verificar o desempenho das diferentes abordagens, o *throughput* foi medido em um dos servidores (sempre o mesmo – líder do consenso [Bessani et al. 2014]) a cada 1000 requisições durante um

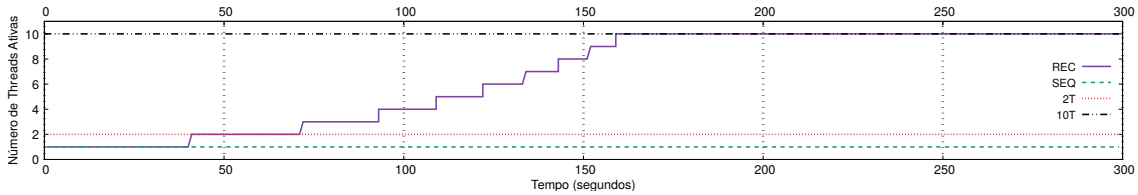
intervalo de 300 segundos. A política de reconfiguração utilizada foi a apresentada no Algoritmo 5, mas utilizamos diferentes períodos para calcular a percentagem de operações conflitantes, conforme descrito nos experimentos.

Resultados e Análises. Três experimentos foram realizados, os quais permitem mostrar que: *(1)* quando o *workload* favorece, o sistema se ajusta para uma configuração com o máximo de *threads* ativas visando aumentar o desempenho; *(2)* quando o *workload* não favorece, o sistema se ajusta para o mínimo de *threads* ativas causando o menor impacto possível no desempenho; e *(3)* quando o workload varia durante a execução, o sistema vai se ajustando para tentar sempre obter o melhor desempenho possível.

No primeiro experimento (Figura 2) o clientes geram um *workload* que favorece o paralelismo (100% `CONFLICT_NONE`) e o sistema foi configurado para iniciar com apenas 1 *thread* ativa, podendo chegar até 10. O período para verificar a necessidade de reconfigurações foi especificado para cada 50.000 requisições. Para efeitos de comparação, os gráficos mostram os valores do *throughput* para o sistema com execuções paralelas e reconfigurações (REC), com execução sequencial que representa uma RME tradicional (SEQ), sem reconfigurações mas paralelo configurado com 2 (2T) e 10 (10T) *threads*. Apesar da melhor configuração para este *workload* ser a com 10 threads, isso só seria possível com um conhecimento prévio a respeito das operações que seriam executadas no sistema (neste caso apenas operações $contains$), o que não é realista.



(a) *Throughput*.

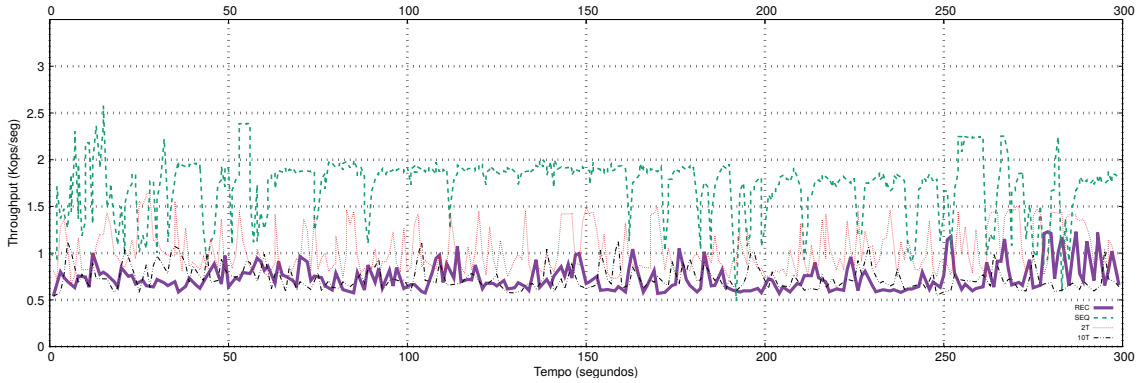


(b) Número de *threads* ativas.

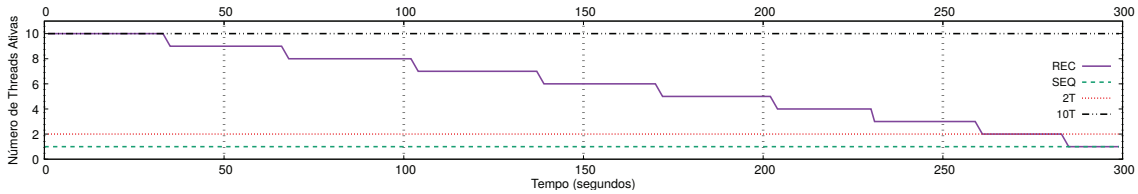
Figura 2. *Throughput* para *workload* de 0% conflitantes.

É possível perceber que com o passar da execução, o sistema com reconfiguração vai ativando as *threads* até atingir o limite máximo de 10, fazendo com que o desempenho seja praticamente o mesmo da configuração 10T. Outro ponto que merece destaque é que com apenas uma *thread* ativa, o desempenho é suavemente inferior ao da execução sequencial devido aos gastos na comunicação entre o escalonador e a *thread* através da fila sincronizada.

O segundo experimento (Figura 3) representa um cenário oposto ao primeiro e mostra que quando o *workload* não favorece (100% `CONFLICT_ALL`), mesmo inicializado com uma configuração inadequada (10 *threads* ativas), o sistema vai desativando *threads* até atingir o limite mínimo de 1 *thread* ativa causando o menor impacto possível no desempenho. Neste experimento, o período para verificar a necessidade de reconfigurações foi configurado para 25.000 requisições. Novamente são apresentados os valores para o sistema configurado como paralelo e reconfigurável (REC), sequencial (SEQ), paralelo com 2 (2T) ou 10 (10T) *threads*. Vale destacar que conforme as *threads* vão sendo desativadas através de reconfigurações, recursos podem ir sendo desalocados.



(a) *Throughput*.



(b) Número de *threads* ativas.

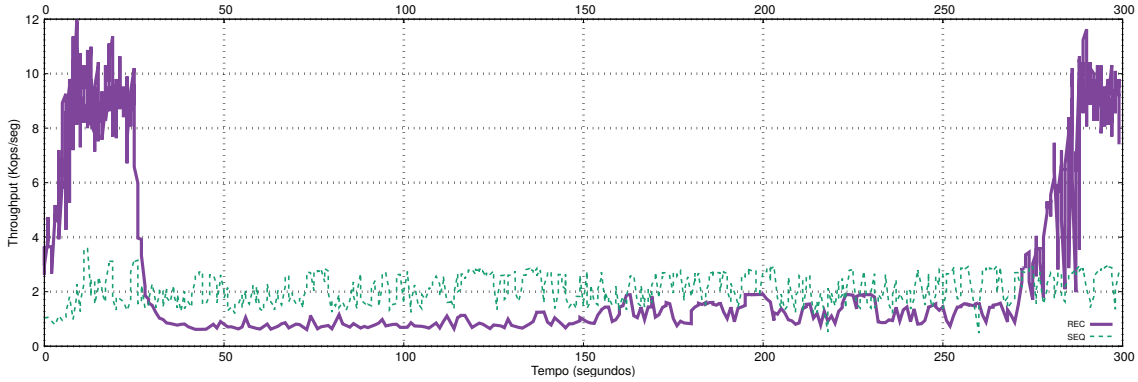
Figura 3. *Throughput* para *workload* de 100% conflitantes.

Finalmente, o terceiro experimento (Figura 4) representa um cenário em que o *workload* sofre variações durante a execução: inicialmente cada cliente executa 5.000 requisições não conflitantes, depois passam a executar 5.000 requisições conflitantes e no final voltam a executar operações não conflitantes. Na execução com reconfiguração, o sistema foi inicializado com 5 *threads* ativas e o período de reconfigurações foi definido como a cada 10.000 requisições.

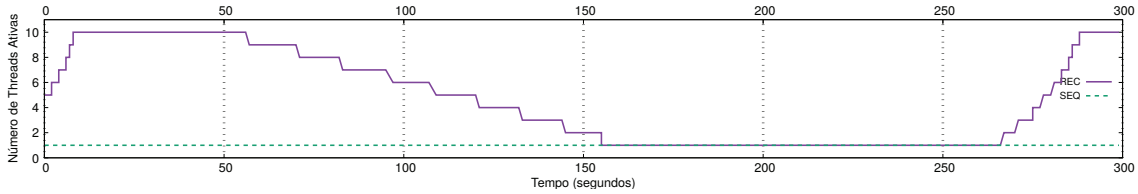
Através deste experimento podemos perceber que o sistema inicialmente ativa *threads* até atingir o limite máximo para obter o melhor desempenho visto que o *workload* favorece. Quando os clientes passam a executar operações conflitantes o sistema passa a desativar as *threads*, as quais são ativadas novamente quando os clientes voltam a executar operações não conflitantes.

Nestes experimentos utilizamos uma política de reconfiguração mais conservadora que ativa ou desativa apenas uma *thread* a cada reconfiguração, o que permitiu mostrar o comportamento dos protocolos propostos. Porém, políticas mais agressivas podem ser utilizadas com o objetivo de se atingir a configuração ideal mais rapidamente. Além disso, analisando sobre outra perspectiva, a taxa de conflitos é algo externo aos servidores e independe do estado interno de alocação e utilização de recursos. Neste sentido, uma

política poderia criar níveis intermediários de paralelismo buscando mapear intervalos de taxas de conflitos para um número de *threads* ativas de forma que o balanço entre desempenho e ociosidade de recursos seja ótimo.



(a) *Throughput*.



(b) Número de *threads* ativas.

Figura 4. *Throughput* variando o *workload* durante a execução.

5. Conclusões

Neste trabalho apresentamos protocolos para implementação de uma RME paralela e reconfigurável, permitindo o ajuste do sistema para o *workload* atual. Não é necessário o conhecimento de nenhuma informação adicional antes da execução do sistema e experimentos mostram os ganhos advindos com a possibilidade de reconfiguração.

Como trabalhos futuros, pretendemos explorar outras formas de escalonar as requisições, como o escalonamento em lotes de forma que o custo da sincronização entre as *threads* seja diluído entre as requisições do lote e, principalmente, analisar políticas de reconfiguração baseadas em outros parâmetros, como o *throughput* e/ou a latência atual apresentada pelo sistema, além do cenário atual de alocação de recursos.

Referências

Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005). Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles*.

Alchieri, E. A. P. (2015). Suportando execuções paralelas no BFT-SMaRt. In *Anais do XVI Workshop de Teste e Tolerância a Falhas- WTF 2015*.

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

Bezerra, C. E., Pedone, F., and Renesse, R. V. (2014). Scalable state-machine replication. In *44th IEEE/IFIP 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.

Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

da Silva Böge, D., da Silva Fraga, J., and Alchieri, E. (2016). Reconfigurable scalable state machine replication. In *Proceedings of the 2016 Latin-American Symposium on Dependable Computing*.

Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010). The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

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.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*.

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

Lamport, L. (2006). Fast paxos. *Distributed Computing*, 19(2):79–103.

Le, L. H., Bezerra, C. E., and Pedone, F. (2016). Dynamic scalable state machine replication. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 13–24.

Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *Proc. of the 34th Int. Conference on Distributed Computing Systems*.

Marandi, P. J. and Pedone, F. (2014). Optimistic parallel state-machine replication. In *Proceedings of the 33rd International Symposium on Reliable Distributed Systems*.

Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*.

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*.

Zbierski, M. (2015). Parallel byzantine fault tolerance. In Wilinski, A., Fray, I. E., and Pejas, J., editors, *Soft Computing in Computer and Information Science*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 321–333. Springer Publishing.