Replicação Máquina de Estados Paralelas com Escalonamento Híbrido*

Aldênio Burgos[1] **, Eduardo Alchieri**[1] **, Fernando Dotti**[2] **, Fernando Pedone**[3]

[1]CIC, Universidade de Brasília, Brasília - Brasil
[2]PUCRS - Escola Politécnica, Porto Alegre - Brazil
[3]Università della Svitzzera italiana - Suíça

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 across replicas. To improve its performance, parallel SMR uses a scheduler to allow parallel execution of some requests. The late scheduler schedules requests for execution after they are ordered, while in the early scheduling part of the scheduling decisions are made before requests are ordered and must be respected during execution. This work proposes a protocol for hybrid scheduling that leverage from the advantages of each of the previous approaches. Experiments show that hybrid scheduler outperforms previous approaches in many scenarios.*

Resumo. *Replicação Máquina de Estados (RME) é uma abordagem muito utilizada na implementação de sistemas tolerantes a falhas e consiste em replicar os servidores e fazer com que os mesmos executem deterministicamente, e na mesma ordem, o mesmo conjunto de requisições. Para melhorar o desempenho, RMEs paralelas utilizam escalonadores que tiram proveito da semântica das requisições e permitem a execução paralela de algumas delas. No escalonamento tardio as requisições são escalonadas para execução após serem ordenadas, enquanto que no escalonamento antecipado parte das decisões de escalonamento são realizadas antes da ordenação e respeitadas durante a execução. Este trabalho propõe um protocolo de escalonamento híbrido que tira proveito das vantagens de cada uma das abordagens anteriores. Experimentos mostram que o escalonador híbrido supera as outras abordagens em diversos cenários.*

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 conjunto de operações requisitadas por clientes, fornecendo um serviço de replicação com consistência forte (linearizabilidade) [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. Esta abordagem 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.

Recentemente surgiram abordagens que, tirando proveito da semântica das operações, empregam protocolos que suportam a execução paralela de parte das operações

*Este trabalho é parcialmente suportado pelo CNPq através do projeto Universal 420092/2018-8.

(por exemplo, [Kotla and Dahlin 2004, Marandi et al. 2014, Mendizabal et al. 2017, Alchieri et al. 2017, Alchieri et al. 2018, Escobar et al. 2019]). Estas abordagens, chamadas de RME paralelas, classificam as requisições (ou operações) em dependentes (ou conflitantes) e independentes (ou não conflitantes), de modo que as requisições independentes são executadas em paralelo nas réplicas enquanto que as requisições dependentes devem ser executadas sequencialmente. 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 variável em comum e ao menos uma das requisições altera o valor desta variável.

Um aspecto importante no contexto de RME paralelas é como escalonar as requisições para serem executadas por um conjunto de *threads* executoras (ou executores) seguindo as restrições acima descritas. Os escalonadores existentes podem ser divididos em dois grupos principais: no *escalonamento tardio* [Kotla and Dahlin 2004, Escobar et al. 2019] as requisições são escalonadas para execução após serem ordenadas pelas réplicas, enquanto que no *escalonamento antecipado* [Alchieri et al. 2017, Alchieri et al. 2018] parte das decisões de escalonamento são realizadas antes da ordenação e precisam ser respeitadas durante a execução. O escalonador tardio é baseado em um grafo de dependências usado para rastrear os conflitos, sendo que a solução com melhor desempenho emprega um algoritmo livre de bloqueios (*lock-free*) para acesso ao grafo [Escobar et al. 2019]. Apesar desta solução distribuir melhor o trabalho entre os executores e apresentar um desempenho superior à execução sequencial até para índices de conflitos considerados altos (exemplo, até 25% de conflitos [Escobar et al. 2019]), um paralelizador único fica responsável por inserir e remover requisições do grafo, limitando o desempenho desta solução. Por outro lado, o escalonamento antecipado tira proveito de decisões previamente realizadas, como por exemplo a divisão do estado da aplicação em partições e a definição de qual executor executará determinadas requisições, e exige pouco processamento no escalonamento, executado pelo classificador. Porém, são necessárias sincronizações adicionais nos executores para execução de requisições conflitantes, o que diminui drasticamente seu desempenho na presença de conflitos [Alchieri et al. 2018].

Neste trabalho propomos uma abordagem *híbrida* para escalonamento, que se aproveita das vantagens de cada uma das soluções anteriores: a simplicidade e rapidez do classificador do modelo antecipado, bem como a distribuição de carga e a sincronização dos executores através de um grafo do modelo tardio. A ideia principal é particionar o estado da aplicação e atribuir um subgrafo para cada partição, com seu paralelizador correspondente. Cada requisição é recebida no classificador e encaminhada para os paralelizadores de acordo com a(s) partição(ões) que acessa. Quando uma requisição endereçada a mais de uma partição é recebida nos paralelizadores, cada um a insere em seu subgrafo, junto com suas dependências, de forma que esta requisição passa a conectar estes subgrafos. Por fim, um conjunto de executores atrelados a cada subgrafo executa as requisições que estiverem prontas, i.e., cujas dependências já foram resolvidas. Experimentos mostram que o escalonador híbrido possui as vantagens do escalonamento tardio e ao mesmo tempo não possui o desempenho limitado por aquilo que um único paralelizador consegue processar, superando as demais abordagens em diversos cenários.

O restante deste texto é organizado da seguinte forma. A Seção 2 apresenta o modelo de sistema assumido e os escalonadores antecipado e tardio. A Seção 3 apresenta a nossa proposta de escalonador híbrido para RMEs paralelas. A Seção 4 discute os

experimentos realizados. Finalmente, a Seção 5 conclui este trabalho.

2. Definições Preliminares

Esta seção detalha o modelo de sistema adotado e os escalonadores tardio e antecipado.

2.1. Modelo de Sistema

Assumimos um sistema distribuído composto de processos interconectados. Há um conjunto ilimitado de processos cliente e um conjunto de n processos servidor (réplicas). O sistema é assíncrono, i.e., não há limite para atrasos de mensagens e velocidades relativas de processo. Assumimos o modelo de falha por parada e excluímos o comportamento malicioso e arbitrário. Um processo é *correto* se não falha, ou *faltoso* caso contrário. Existem no máximo f réplicas faltosas, de $n = 2f + 1$ réplicas.

Os processos se comunicam por envio de mensagens, usando comunicação ponto-a-ponto ou difusão atômica. A comunicação ponto-a-ponto usa as primitivas $send(m)$ e $receive(m)$, onde m é uma mensagem. Se um remetente enviar uma mensagem vezes suficientes, um destinatário correto eventualmente receberá a mensagem. A difusão atômica é definida pelas primitivas $broadcast(m)$ e $deliver(m)$, onde m é uma mensagem, e garante as seguintes propriedades [Défago et al. 2004, Hadzilacos and Toueg 1993] [1]:

- *Validade*: Se um processo correto difunde a mensagem m, então algum processo correto eventualmente entrega m.
- *Acordo uniforme*: Se um processo correto entrega a mensagem m, então todos os processos corretos eventualmente entregarão m.
- *Integridade uniforme*: Para qualquer mensagem m, se m foi transmitida por um processo, todo processo correto entregará m uma única vez.
- *Ordem total uniforme*: Se dois processos p e q entregam as mensagens m e m', então p entrega m antes de m', se e somente se, q entrega m antes de m'.

RME provê consistência forte ou *linearizabilidade* [Herlihy and Wing 1990]. Uma execução é linearizável se houver uma maneira de ordenar totalmente as operações de tal forma que (a) respeite a semântica dos objetos acessados pelas operações, expressa em suas especificações sequenciais; e (b) respeite a ordenação no tempo das operações. Existe uma ordem no tempo entre duas operações se uma operação termina em um cliente antes que a outra operação comece em outro cliente.

Os algoritmos de escalonamento que discutimos neste artigo exploram a concorrência entre comandos. Assumindo que C é o conjunto de comandos possíveis, então $\#_C \subseteq C \times C$ é a relação de conflito entre os comandos. Se $\{c_i, c_j\} \in \#_C$, então os comandos c_i e c_j conflitam e suas execuções devem ser serializadas; caso contrário, c_i e c_j podem ser executados simultaneamente.

2.2. Escalonamento Tardio

Nesta categoria de protocolos, as réplicas entregam as requisições em ordem total e, em seguida, um paralelizador em cada réplica atribui as requisições aos executores. O paralelizador deve respeitar as dependências entre as requisições: se as requisições r_i e r_j conflitam e r_i é entregue antes de r_j, então r_i deve ser executada antes de r_j; se r_i e r_j são independentes, não há restrições (por exemplo, o paralelizador pode atribuir cada requisição a um executor diferente).

[1] Difusão atômica exige suposições adicionais de sincronia para implementação [Fischer et al. 1985], as quais não são explicitamente usadas pelos protocolos propostos neste artigo.

CBASE [Kotla and Dahlin 2004] é um protocolo nesta categoria. O paralelizador em cada réplica entrega as requisições em ordem total e as inclui em um grafo de dependências onde os vértices representam as requisições e as arestas direcionadas representam as dependências entre elas. A requisição r_i depende de outra requisição r_j (ou seja, $r_i \rightarrow r_j$ é uma aresta no grafo) se r_i foi entregue após r_j e há um conflito entre elas. O grafo de dependências é compartilhado com um conjunto de executores que buscam requisições para executar, sempre respeitando suas interdependências: uma requisição só pode ser executada se ainda não estiver executando e não depender de outra no grafo. Após executá-la, o executor remove-a do grafo e escolhe outra.

O escalonamento tardio adota o seguinte modelo de execução nas réplicas:

1. $m + 1$ *threads*: um paralelizador e m executores.
2. Todas as *threads* acessam um grafo de dependências comum para inserir (paralelizador) e obter/remover (executores) requisições.
3. O paralelizador recebe as requisições em ordem total e insere cada requisição no grafo, inserindo também as dependências de acordo com os conflitos encontrados.
4. Cada executor seleciona uma requisição do grafo que esteja pronta para execução, marcando-a como "em execução" e, após seu processamento, remove-a do grafo.

Recentemente, uma estrutura de dados abstrata, chamada *Conflict-Ordered Set* (COS), foi proposta para capturar os requisitos de uma RME paralela [Escobar et al. 2019]. O COS generaliza as técnicas baseadas em grafos de dependências: o paralelizador inclui novas requisições no COS, respeitando sua ordem de entrega, enquanto que executores obtêm requisições do COS para execução de acordo com suas dependências e, após executadas, removem-nas do COS. Neste mesmo artigo foi proposta uma implementação livre de bloqueios (*lock-free*) para o COS e foi demonstrado que o rastreamento de dependências pode se tornar o gargalo de desempenho, particularmente quando o COS usa bloqueios de granularidade grossa em modo exclusivo como no CBASE. A abordagem baseada em um COS livre de bloqueios apresenta melhor desempenho. Porém, nesta abordagem os executores apenas marcam as requisições como logicamente removidas e o paralelizador único é responsável por inserir e remover fisicamente as requisições do grafo, tornando-se o gargalo da solução.

2.3. Escalonamento Antecipado

A principal ideia do escalonamento antecipado [Alchieri et al. 2017, Alchieri et al. 2018] é realizar parte das decisões de escalonamento antes das requisições serem ordenadas, reduzindo o trabalho realizado nas replicas durante o escalonamento. Para isso, as requisições são agrupadas em classes que são mapeadas para um conjunto de executores. Assim, os clientes adicionam o identificador da classe de cada requisição antes de seu envio, indicando como a mesma deve ser processada. Um classificador entrega as requisições em ordem total e encaminha, com base na classe informada, cada requisição a um ou mais executores, conforme detalhado mais adiante. Esta associação de uma requisição a uma classe requer conhecimento da aplicação. Por exemplo, pode-se estabelecer que todas as requisições que acessam o objeto x são executadas pelo executor t_0 e todas as requisições que acessam o objeto y são executadas pelo executor t_1. Neste caso, para a execução de uma requisição r que acessa ambos os objetos é necessária a coordenação entre os executores t_0 e t_1. A vantagem do escalonamento antecipado é que as decisões de escalonamento são simples nas réplicas, fazendo com que o classificador tenha menos probabilidade de se tornar um gargalo. Por outro lado, a necessidade

de coordenação entre os executores para execução de requisições conflitantes (como a requisição r) diminui drasticamente o desempenho do sistema.

Em [Alchieri et al. 2017], a noção de classes de requisições é introduzida para denotar as dependências, codificando assim o conhecimento da aplicação. Dado um serviço com um conjunto R de requisições possíveis, um conjunto C de classes é definido e, para cada classe de C: (i) um subconjunto não vazio e não sobreposto de requisições de R está associado; e (ii) um subconjunto de classes conflitantes está associado. Um conflito entre as classes acontece quando quaisquer duas requisições dessas classes entram em conflito. As requisições que pertencem a classes conflitantes devem ser serializadas de acordo com a ordem total induzida pela difusão atômica, enquanto requisições de classes independentes podem ser executadas simultaneamente. Observe que uma classe pode ser auto-conflitante, o que significa que suas requisições devem ser serializadas sempre.

O escalonamento antecipado adota o seguinte modelo de execução nas réplicas:

1. $m + 1$ *threads*: um classificador e m executores.
2. Cada executor tem uma fila de entrada separada e remove requisições desta fila em ordem FIFO.
3. O classificador entrega as requisições na ordem total e encaminha cada requisição r para uma ou mais filas de entrada:
    a. Caso seja encaminhada para apenas um executor, r depende das requisições anteriores atribuídas a esse executor, mas pode ser processada paralelamente com outras requisições de outros executores.
    b. Caso seja encaminhada para mais de um executor, r depende das requisições anteriores atribuídas a esses executores. Neste caso, todos os executores envolvidos em r devem sincronizar entre si, para que apenas um deles execute r, enquanto os outros aguardam.

Nesse modelo de execução, as seguintes regras de mapeamento de classes para executores devem ser aplicadas para garantir execuções linearizáveis:

1. *Cada classe está associada a pelo menos um executor:* garantia de que as requisições sejam eventualmente executadas.
2. *Caso uma classe seja auto-conflitante, ela é sequencial:* cada requisição é encaminhada para todos os executores da classe e processada conforme descrito anteriormente.
3. *Se houver conflito de duas classes, pelo menos uma delas deve ser sequencial:* o requisito anterior pode ajudar a decidir qual classe escolher como sequencial.
4. *Para classes conflitantes c_1 (sequencial) e c_2 (concorrente), o conjunto de executores associados a c_2 deve ser incluído no conjunto de executores associados a c_1:* esse requisito garante que as requisições em c_2 sejam serializadas em relação a c_1.
5. *Para classes sequenciais conflitantes c_1 e c_2, basta que c_1 e c_2 tenham pelo menos um executor em comum:* o executor comum garante que as requisições destas classes sejam serializadas.

Essas regras resultam em vários mapeamentos possíveis de classes para executores e foi modelado como um problema complexo de otimização [Alchieri et al. 2018]. Um mapeamento é definido como $CtoT = C \rightarrow \{Seq, Conc\} \times \mathcal{P}(T)$, onde C é o conjunto de nomes de classes; $\{Seq, Conc\}$ é o modo de sincronização sequencial ou concorrente de uma classe; e $\mathcal{P}(T)$ são os subconjuntos possíveis dos n executores $T = \{t_0, .., t_{n-1}\}$.

3. Escalonamento Híbrido

Esta seção apresenta a abordagem híbrida de escalonamento. Após a visão geral da proposta, os protocolos são apresentados em detalhes.

3.1. Visão Geral

O escalonamento híbrido busca aproveitar as vantagens de cada uma das soluções anteriores, i.e., a simplicidade e rapidez do classificador do modelo antecipado, bem como a distribuição de carga e a sincronização dos executores através de uma implementação do COS baseada em grafo livre de bloqueios do modelo tardio. A ideia geral é permitir que vários paralelizadores insiram paralelamente requisições no COS, removendo o gargalo do modelo tardio. Para isso, o estado da aplicação é particionado e o COS é formado por vários subgrafos (Figura 1), um para cada partição, sendo que requisições endereçadas a mais de uma partição criam um nó conectando os subgrafos correspondentes.

Uma classe sequencial é associada a cada partição e mapeada para um paralelizador diferente. Uma vez que todas as inserções em uma mesma partição devem ser sequenciais, não faz sentido usar mais de um paralelizador por partição. Além disso, como uma requisição pode acessar quaisquer partições da aplicação, são criadas classes para todas as combinações possíveis de partições e os respectivos paralelizador são atrelados a cada classe, permitindo a sincronização na inserção destas requisições. Deste modo, o cliente deve incluir o identificador da classe em cada requisição e o classificador recebe as requisições e as distribui entre os paralelizadores, de acordo com suas classes. Os paralelizadores inserem as requisições paralelamente no COS, i.e., nos respectivos subgrafos. Por fim, um conjunto de executores acessa o COS para execução das requisições.

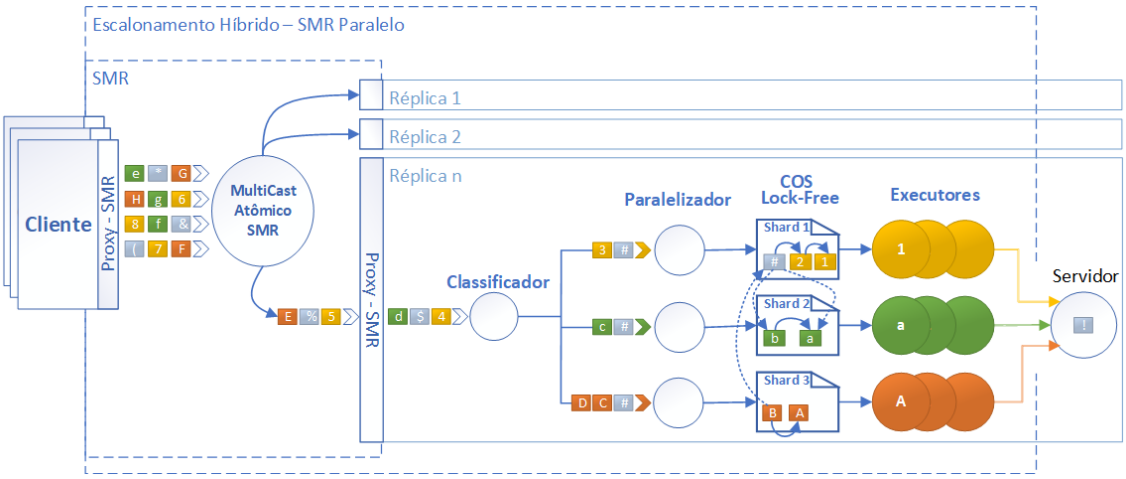


Figura 1. Escalonamento Híbrido: esquema de uma réplica

O escalonamento híbrido adota o seguinte modelo de execução nas réplicas:

- $m + p + 1$ *threads*: um classificador, p paralelizadores e m executores.
- Cada paralelizador tem uma fila de escalonamento separada e remove requisições desta fila em ordem FIFO.
- Cada partição possui um conjunto de executores e um paralelizador associado, os quais acessam o subgrafo de dependências da partição para inserir (paralelizador) e obter/remover (executores) requisições.

- O classificador entrega as requisições na ordem total e encaminha cada requisição r para uma ou mais filas de escalonamento:
    a. Caso r acesse apenas uma partição s, r depende das requisições anteriores endereçadas a s e é incluída na fila do paralelizador correspondente a s.
    b. Caso r acesse mais de uma partição, r depende das requisições anteriores endereçadas a estas partições e é incluída nas filas correspondentes. Neste caso, todos os paralelizadores envolvidos em r devem incluir as dependências de r em suas respectivas partições, sendo que um deles também insere a requisição em seu subgrafo. Uma requisição só é considerada inserida no COS quando todos os paralelizadores envolvidos computarem as dependências.
- Cada executor seleciona uma requisição que esteja pronta para execução no subgrafo correspondente, marcando-a como "em execução" e, após seu processamento, remove-a (logicamente) do subgrafo.

3.2. Protocolo

No Algoritmo 1 podemos ver os principais tipos de dados utilizados e o classificador. O nó possui a requisição (ou comando) do cliente, o identificador da partição (ou *shard*) responsável por seu processamento e um contador de partições que representa os paralelizadores que ainda restam processar este nó. Possui também seu status e a referência atômica para o próximo nó da lista de nós do subgrafo a que pertence, além de um vetor de conjuntos de nós dos quais depende e um vetor de conjuntos dos nós que dependem deste nó. Cada paralelizador modifica apenas os conjuntos de sua posição no vetor, não havendo problema de concorrência no acesso a estes conjuntos. O COS (linhas 12-15) representa uma partição do COS global, sendo composto pela lista N de nós do subgrafo e um conjunto R de nós cujas requisições foram endereçadas a várias partições e que foram inseridos em alguma outra partição, i.e., nós cujas dependências também devem ser verificadas durante uma inserção. Apenas o paralelizador da partição correspondente acessa este conjunto, também não apresentando problemas de concorrência de acesso. Cada $Shard$ (linhas 16-21) possui uma fila de escalonamento, um COS, um semáforo para controlar o seu tamanho e outro para controlar o número de nós prontos para execução.

O classificador da réplica é o primeiro estágio do escalonamento híbrido. Quando uma requisição r é entregue pelo protocolo de difusão atômica, o classificador recupera a lista de partições vinculadas à classe da requisição e armazena seu tamanho na variável rem_s. Uma dessas partições é selecionada como responsável pela requisição. Então, um novo nó é incluído nas filas de escalonamento das partições envolvidas (linhas 29-35).

No Algoritmo 2 podemos ver que cada paralelizador (linhas 1-8) retira as requisições da suas filas de escalonamento e as insere no COS, indicando o subgrafo correspondente. Caso a partição relacionada ao paralelizador seja a responsável pela requisição, então um espaço é adquirido, as dependências são criadas e o nó é inserido em N. Caso contrário, as dependências ainda são criadas, mas o nó é inserido em R, para ser testado quando a conflitos nas inserções futuras. Além do paralelizador, cada partição possui vários executores (linhas 9-15). Cada executor aguarda por um nó pronto, recupera o nó da sua partição do COS e executa a requisição correspondente. Por fim, marca o nó como logicamente removido e disponibiliza um espaço na partição.

Os algoritmos 3 e 4 apresentam a implementação das funções do COS. Estes algoritmos foram estendidos da implementação do COS livre de blo-

Algorithm 1 Tipos de dados e Classificador (1 por réplica)

1: **Tipos de dados:**
2: $Node : \{$
3: $c : Command,$ {*o comando*}
4: $s_{id} : int,$ {*id da partição responsável*}
5: $rem_s : int,$ {*número de partições remanescentes*}
6: $st : \{wtg, rdy, exe, rmd\}$ {*nó pode estar aguardando, pronto, executando ou removido*}
7: $depOn[]$: array of set of $NodeRef$ {*nós dos quais este nó depende, um conjunto por partição*}
8: $depMe[]$: array of set of $NodeRef$ {*nós que dependem desse nó, um conjunto por partição*}
9: $nxt : NodeRef$ {*próximo nó em ordem de chegada*}
10: $\}$
11: $NodeRef$: **atomic reference** to $Node$
12: $COS : \{$
13: $N : NodeRef, initially \perp$ {*lista com os nós do subgrafo da partição*}
14: $R : set\ of\ NodeRef, initially \emptyset$ {*conjunto de nós relacionados, de outras partições*}
15: $\}$
16: $Shard : \{$
17: $queue \leftarrow \emptyset,$ {*fila de escalonamento da partição, inicialmente vazia*}
18: $cos \leftarrow \langle N : \perp, R : \emptyset \rangle,$ {*COS da partição - Algoritmo 3*}
19: $space \leftarrow new\ Semaphore(maxSize/|shards|),$ {*semáforo de espaço da partição*}
20: $ready \leftarrow new\ Semaphore(0)$ {*semáforo de pronto da partição*}
21: $\}$
22: $Request : \{c : Command,$ {*uma requisição tem o seu comando*}
23: $classId : C_{id}\}$ {*e o identificador de sua classe*}
24: **Variáveis:**
25: S {*conjunto com os identificadores das partições/shards*}
26: $shards[S] : array\ of\ Shard$ {*vetor de partições, uma estrutura para cada partição*}
27: $C_{id} : set\ of\ int$ {*identificadores das classes de comandos*}
28: **Código do Classificador da réplica:**
29: **onDeliver**$(req : Request)$
30: $shards_{ids} \leftarrow CtoS(req.classId)$ {*recupera as partições envolvidas na requisição*}
31: $rem_s \leftarrow |shards_{ids}|$ {*conta o número de partições envolvidas*}
32: $ss_{id} \leftarrow selectShard(req.classId)$ {*seleciona a partição responsável pela requisição*}
33: $node \leftarrow createNode(req.c, ss_{id}, rem_s)$ {*cria o nó do grafo COS*}
34: **for all** $s \in shards_{ids}$ **do** {*para cada partição envolvida...*}
35: $shards[s].queue.fifoPut(node)$ {*...põe o nó em sua fila de escalonamento*}
36: **CtoS**$(class_{id} : C_{id}) : set\ of\ Shard$ identifiers {*mapeamento de **C**lasses para ids de **S**hards*}
37: **return** $\mathcal{P}(shards)$ {*o subconjunto não vazio de identificadores de partições da classe*}
38: **selectShard**$(class_{id} : C_{id}) : int$
39: **return** posição da partição escolhida em round-robin dentre $CtoS(class_{id})$
40: **createNode**$(c : Command, s_{id}, rem_s : int) : NodeRef \leftarrow$ **new** $Node\{c, s_{id}, rem_s, \perp, \emptyset, \emptyset, \perp\}$

queios [Escobar et al. 2019] para lidar com as várias partições do modelo híbrido. No Algoritmo 3 encontramos a interface pública, que é composta por três funções:

insert: insere o nó e suas dependências, removendo nós marcados como logicamente removidos. Primeiramente são criadas as relações de dependência com os nós de R e N. O parâmetro $insert_{node}$ indica se o nó deve ser inserido em N ou R, dependendo da partição responsável pelo nó. Por fim, é verificado se o paralelizador atual é o último a tratar desse nó. Quando isto acontece, o nó é atualizado para aguardando e testado quanto a sua prontidão, i.e., se ainda possui conflitos não resolvidos com outras requisições.

remove: marca o nó n como logicamente removido, em seguida todos os nós que dependem de n em todas as partições são testados quando a sua prontidão.

get: percorre os nós da partição até encontrar um nó com status de pronto, modificando-o atomicamente para executando.

Algorithm 2 Paralelizador (1 por partição) e Executor (n por partição)

```
1:  Código do Paralelizador da partição s_id:
2:      while true do                                          {loop infinito}
3:          node ← shards[s_id].queue.fifoPull()               {próximo nó para escalonar}
4:          if node.s_id = s_id then                           {este é o paralelizador responsável?}
5:              shards[s_id].space.down()                      {garante o espaço para inserir o nó}
6:              shards[s_id].cos.insert(node, s_id, true)      {insere o nó e suas dependências...}
7:          else
8:              shards[s_id].cos.insert(node, s_id, false)     {...ou só as dependências}

9:  Código do Executor do shard s_id:
10:     while true do                                          {loop infinito}
11:         shards[s_id].ready.down()                          {garante um nó pronto para execução}
12:         node ← shards[s_id].cos.get()                      {recupera o nó livre}
13:         execute(node.c)                                    {executa o comando do nó}
14:         shards[s_id].cos.remove(node)                      {marca o nó como removido}
15:         shards[s_id].space.up()                            {libera o espaço para novos nós}
```

Algorithm 3 Versão estendida do COS Livre de bloqueios - Funções públicas

```
1:  insert(n_n : NodeRef, s_id : int, insert_node : boolean)
2:      insertRelatedDependencies(n_n, s_id)     {constrói as dependências com os nós relacionados}
3:      n ← insertDependencies(n_n, s_id)        {constrói as dependências com os nós da partição}
4:      if insert_node then                                    {insere o nó no grafo...}
5:          if n = ⊥ then N ← n_n else n.nxt ← n_n
6:      else                                                   {...ou no conjunto de nós relacionados}
7:          R ← R ∪ {n_n}
8:      if decrementAndGet(n_n.rem_s) = 0 then                 {se esta é a última partição...}
9:          n_n.st ← wtg                                       {...o nó ficará aguardando e...}
10:         testReady(n_n)                                     {...será testado para definir se está pronto}

11: remove(n : NodeRef)                                        {assume que existe um n e que n.st = exe}
12:     n.st ← rmd                                             {remoção lógica}
13:     for all s_id ∈ S do                                    {para todas as partições}
14:         for all n_i ∈ n.depMe[s_id] do                     {para todos os nós que dependem de n}
15:             testReady(n_i)                                 {verifica se n_i está pronto}

16: get(): NodeRef                                             {assume que existe um nó pronto}
17:     n ← N                                                  {inicia a procura pelo nó pronto}
18:     while n ≠ ⊥ do                                         {considere cada nó, por ordem de chegada}
19:         if compareAndSet(n.st, rdy, exe) then              {se n estava pronto, agora está executando}
20:             return n                                       {...devolva n}
21:         n ← n.nxt                                          {...ou vai para o próximo}
```

No Algoritmo 4 são apresentadas as funções auxiliares do COS. As funções $insertDependencies$ e $insertRelatedDependencies$ inserem as dependências de um nó com os nós de N e R, respectivamente. Além disso, os nós que foram logicamente removidos são fisicamente removidos das estruturas. A função $removeDependencies$ evita vazamentos de memória, removendo as dependências para um nó logicamente removido, na partição especificada. Esta função é chamada durante a exclusão física do nó

(linhas 5 e 14). A função auxiliar $bind$ é responsável pela criação de uma relação de dependência entre dois nós. Já a função $testReady$ verifica se o nó em questão está pronto para execução, i.e., se o conjunto das suas dependências não removidas está vazio e seu status ainda é aguardando. Neste caso, seu status muda atomicamente para pronto e o semáforo de prontos é utilizado para liberar um executor. As funções $compareAndSet$ e $decrementAndGet$ realizam operações simples de forma atômica.

Algorithm 4 COS Livre de bloqueios - Funções auxiliares

```
1: insertDependencies(n_n : NodeRef, s_id : int) : NodeRef
2:     n' ← n ← N                                              {seta n' e n para N}
3:     while n' ≠ ⊥ do                                         {considere cada nó, em ordem}
4:         if n'.st = rmd then                                 {n' está logicamente removido}
5:             removeDependencies(n', s_id)                    {remove as dependência de n' na partição s_id}
6:             if !compareAndSet(N, n, n'.nxt) then            {remove n' de N caso seja o primeiro nó de N}
7:                 atomic { n.nxt ← n'.nxt }                   {remove n' nos outros casos}
8:         else if conflict(n_n, n') then bind(n_n, n', s_id)  {se conflito, cria uma dependência}
9:         n ← n'; n' ← n'.nxt                                 {vai para o próximo}
10:    return n

11: insertRelatedDependencies(n_n : NodeRef, s_id : int)
12:    for all n ∈ R do                                        {considere cada nó relacionado, em R}
13:        if n.st = rmd then                                  {n está logicamente removido?}
14:            removeDependencies(n, s_id)                     {remove as dependência de n na respectiva partição}
15:            R ← R \ n                                       {remove n de R}
16:        else if conflict(n_n, n) then bind(n_n, n, s_id)    {se conflito, cria uma dependência}

17: removeDependencies(n : NodeRef, s_id : int)
18:    for all n_i ∈ n.depMe[s_id] do                          {para cada nó que depende de n}
19:        n_i.depOn[s_id] ← n_i.depOn[s_id] \ {n}             {remove n das dependências}

20: bind(n_new, n_old : NodeRef, s_id : int)
21:    n_old.depMe[s_id] ← n_old.depMe[s_id] ∪ {n_new}         {n_old é uma dependência de n_new}
22:    n_new.depOn[s_id] ← n_new.depOn[s_id] ∪ {n_old}         {n_new depende de n_old}

23: testReady(n : NodeRef)
24:    if {n_i ∈ {∃s_id ∈ S : n_i ∈ n.depOn[s_id] ∧ n_i.st ≠ rmd}} = ∅ then   {não possui dependências}
25:        if compareAndSet(n.st, wtg, rdy) then               {muda n para pronto}
26:            shards[n.s_id].ready.up()                       {informa que existe um nó pronto em s_id}

27: compareAndSet(a, b, c): boolean ← atomic { if a = b then a ← c; true else false }

28: decrementAndGet(value : int) : int ← atomic {value ← value − 1; value}

29: conflict(n_i, n_j : Node): boolean ← (n_i.c, n_j.c) ∈ #_c    {estes comandos conflitam?}
```

4. Experimentos

Realizamos uma avaliação experimental dos escalonadores com os objetivos de (1) mostrar as vantagens e desvantagens das abordagens tardia e antecipada, (2) apresentar como o escalonador híbrido contorna esses problemas de desempenho em um sistema com estado particionado, e (3) comparar seus desempenhos quando usados para o escalonamento de requisições em uma RME paralela com cargas de trabalho desbalanceadas.

Setup. Os escalonadores foram implementados usando a biblioteca de RME BFT-SMART [Bessani et al. 2014]. A biblioteca foi configurada para tolerar apenas falhas por

parada em todos os experimentos. O BFT-SMART foi desenvolvido em Java e seu protocolo de difusão atômica executa uma sequência de instâncias de consenso, onde cada instância ordena um lote de mensagens. Para melhorar ainda mais o seu desempenho, implementamos interfaces que permitem que os clientes enviem um lote de requisições dentro de cada mensagem. O ambiente experimental foi configurado com 7 máquinas conectadas em uma rede de 1Gbps. O BFT-SMART foi configurado com 3 réplicas hospedadas em máquinas Dell[2] separadas para tolerar até 1 falha de réplica, enquanto 800 clientes foram distribuídos uniformemente em outras 4 máquinas HP[3].

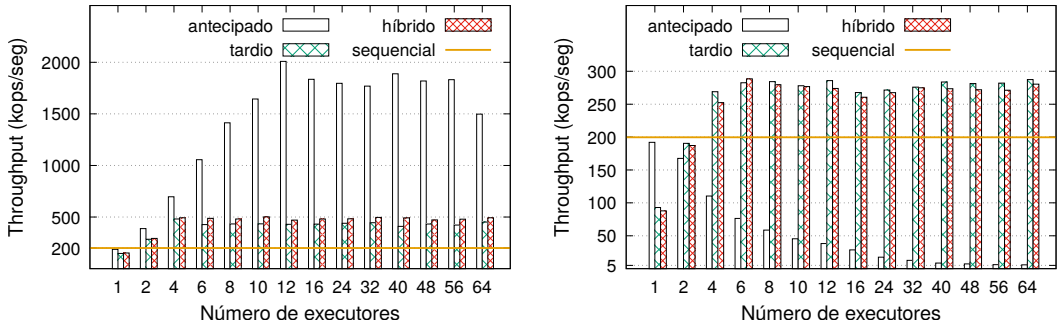
Aplicação. Dois serviços foram implementados, sendo um com apenas uma partição e outro multi-particionado. A aplicação escolhida para o serviço com uma partição foi uma lista encadeada de inteiros com operações de leitura e escrita, a primeira $contains(i)$ verifica se i está na lista e a segunda $add(i)$ inclui i na lista. Observe que no modelo de concorrência desta aplicação, as requisições $contains$ não conflitam umas com as outras, mas conflitam com as requisições add, que conflitam com todas as requisições. Para o serviço multi-particionado, usamos como aplicação um conjunto dessas listas (uma por partição) acrescentando novas operações de leitura e escrita em várias partições simultâneas: $contains_S(i)$ que executa $contains(i)$ nas listas associadas a cada partição $k \in S$ e $add_S(i)$ que executa $add(i)$ nas listas associadas a cada partição $k \in S$. No modelo de concorrência para esta aplicação, $add_{S'}$ conflitam com cada $contains_{S''} : (S' \cap S'') \neq \emptyset$ e com cada $add_{S''} : (S' \cap S'') \neq \emptyset$. Cada lista foi inicializada com mil entradas em cada réplica (variando de 0 a 999). O parâmetro inteiro i usado nas operações é sempre uma posição escolhida aleatoriamente na lista. Foi utilizada uma implementação eficiente de filas sem bloqueio do tipo consumidor único e produtor único [Maffione et al. 2018] para os escalonamentos antecipado e híbrido. Também configuramos o tamanho máximo do grafo de dependências com 150 entradas para as abordagens tardia e híbrida. Nos experimentos, medimos a vazão (ou *throughput*) do sistema nos servidores e a latência de cada requisição nos clientes. Uma fase de aquecimento precedeu cada experimento.

Resultados. O primeiro conjunto de experimentos considera leituras e escritas em uma única partição. A Figura 2 mostra o *throughput* obtido por cada escalonador para diferentes números de executores, considerando duas cargas de trabalho: uma com apenas requisições de leitura e outra composta por 10% de escritas e 90% de leituras, ambas uniformemente distribuídas entre os clientes. Em geral, o escalonamento antecipado se destaca em cargas de trabalho sem conflitos, mas o desempenho diminui abruptamente quando há conflitos. Isso acontece porque o classificador é muito rápido, mas os executores precisam sincronizar entre si para executar requisições conflitantes. Além disso, um aumento no número de executores reduz o desempenho nas cargas com conflitos. Vale ressaltar que o mapeamento de classes de requisições para executores é um problema de otimização complexo [Alchieri et al. 2018]. Por outro lado, o escalonamento tardio usa um grafo livre de bloqueios para rastrear as dependências, sendo que o paralelizador é responsável por manter o grafo, inserindo e removendo fisicamente as requisições. Embora essa abordagem equilibre melhor a carga de trabalho entre os executores, em determinado ponto, o paralelizador se torna o gargalo e o desempenho não aumenta mesmo com a adição de novos executores. Observe que o escalonamento híbrido tem um desempenho

[2]Dell PowerEdge R815 com 4 processadores AMD Opteron 6366HE de 16 núcleos de 1,8 GHz e 128 GB de RAM.

[3]HP SE1102 com 2 processadores Intel Xeon L5420 quad-core de 2,5 GHz e 8 GB de RAM.

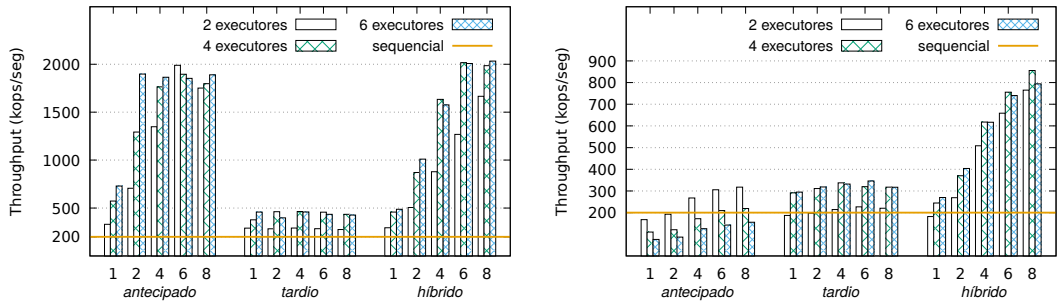
semelhante ao do escalonamento tardio, pois, para sistemas com apenas uma partição, ambas as abordagens funcionam de maneira semelhante.



(a) 0% de escritas (b) 10% de escritas

Figura 2. Throughput para diferentes quantidades de escritas e de executores.

O primeiro conjunto de experimentos mostrou que os escalonamentos híbrido e tardio apresentaram desempenhos semelhantes em um sistema com uma única partição, e o escalonamento antecipado teve um desempenho melhor em cargas de trabalho sem conflitos. Uma questão natural é verificar como estas abordagens se comportam em um sistema multi-particionado, uma vez que os escalonadores antecipado e híbrido tentam se aproveitar das partições do sistema, enquanto que o escalonador tardio não leva este fator em consideração. Para isso, a Figura 3 apresenta os resultados de um sistema multi-particionado considerando diferentes números de partições e executores, além de duas cargas de trabalho: uma carga apenas com requisições de leitura em partições individuais e outra composta por 10% de escritas e 90% de leituras, sendo que 5% são requisições em multi-partições e 95% em partições individuais. As requisições de partição individual são distribuídas uniformemente entre as partições e todas as requisições foram distribuídas uniformemente entre os clientes. Além disso, 10% das requisições multi-partição envolveram todas as partições, enquanto que o restante foi endereçado a duas partições aleatoriamente escolhidas.



(a) 0% de escritas, 0% global (b) 10% de escritas, 5% global

Figura 3. *Throughput* para diferentes quantidades de escritas e requisições globais, variando as quantidades de partições e de executores. Para o escalonador antecipado e híbrido, o número de executores se refere a quantidade por partição, enquanto que no escalonador tardio este é o número total.

Em geral, o escalonamento híbrido superou o desempenho das outras abordagens. Como esta abordagem usa um paralelizador por partição, para inserir e remover as requisições no subgrafo relacionado, seu desempenho escala com o número de partições. O escalonador híbrido também atinge o pico de *throughput* do escalonamento antecipado, para a carga de trabalho apenas com leituras, quando configurado com 6 ou mais partições. Por outro lado, o escalonamento antecipado não escala para mais do que 2 executores por partição com requisições conflitantes e o desempenho do escalonamento tardio é novamente limitado pelo paralelizador único que mantém o grafo. Observe que o escalonamento tardio foi executado com menos executores do que os outros, mas o fato é que é inútil utilizar mais executores, pois o gargalo está no paralelizador.

A Figura 4 mostra os resultados de latência média versus *throughput* para as configurações com melhor desempenho para 8 partições e uma carga de trabalho com 5% global e 10% de escritas. Nas abordagens para RME paralela, todas as requisições têm latência semelhante porque possuem custos de execução semelhantes e a sincronização de escritas impacta o desempenho das leituras ordenadas após uma escrita. Obviamente, o mesmo comportamento ocorre em um RME sequencial. É possível observar que todas as abordagens apresentaram latência semelhante até próximo à saturação do sistema e a partir deste ponto a latência aumenta abruptamente. Como o mesmo comportamento ocorre para as demais configurações e cargas de trabalho, optamos por apresentar apenas estes casos como exemplos. O mesmo comportamento é relatado em trabalhos anteriores sobre RME (por exemplo, [Bessani et al. 2014, Alchieri et al. 2018, Escobar et al. 2019]).

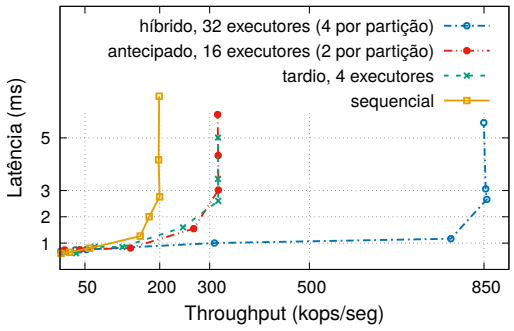


Figura 4. Latência considerando 8 partições e uma carga de trabalho com 10% de escritas e 5% de requisições globais.

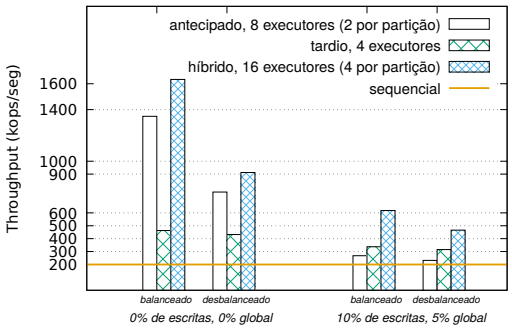


Figura 5. *Throughput* considerando 4 partições com cargas de trabalho balanceadas e desbalanceadas.

Finalmente, a Figura 5 apresenta os valores de *throughput* para um sistema com 4 partições considerando as mesmas cargas de trabalho apresentadas na Figura 3 (balanceado) e também para os casos em que uma partição recebe 50% das requisições e o restante é uniformemente distribuído entre as outras três partições (desbalanceado). Para cada escalonador, usamos a configuração que em geral apresentou melhor desempenho para cargas de trabalho balanceadas. O escalonamento tardio apresentou desempenho semelhante para cargas de trabalho balanceadas e desbalanceadas, uma vez que não tem a noção de particionamento. Por outro lado, o desempenho dos escalonamentos híbrido e antecipado diminui em cargas de trabalho desbalanceadas, pois enquanto algumas partições estão sobrecarregadas, outras têm menos trabalho para processar. No entanto, o escalonamento híbrido novamente supera as outras abordagens.

5. Conclusões

RMEs paralelas permitem que algumas requisições sejam executadas em paralelo nas réplicas, sendo que para manter a consistência entre as réplicas, as dependências entre as requisições precisam ser rastreadas e respeitadas em cada réplica. Esta tarefa é executada por um protocolo de escalonamento. Este trabalho apresentou nossos esforços para aumentar o desempenho de uma RME paralela através de uma abordagem híbrida de escalonamento que tira proveito das vantagens das abordagens anteriores. Um conjunto de experimentos mostrou que o escalonador híbrido supera os anteriores em diversos cenários.

Referências

Alchieri, E., Dotti, F., Mendizabal, O. M., and Pedone, F. (2017). Reconfiguring parallel state machine replication. In *SRDS*.

Alchieri, E., Dotti, F., and Pedone, F. (2018). Early scheduling in parallel state machine replica. In *ACM SoCC*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with bft-smart. In *DSN*.

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

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

Escobar, I. A., Alchieri, E., Dotti, F. L., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. In *Proc. 20th International Middleware Conference*.

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.

Hadzilacos, V. and Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In Mullender, S., editor, *Distributed Systems*, pages 97–145. ACM Press/Addison-Wesley, New York, NY, USA.

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

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *IEEE/IFIP Int. Conference on Dependable Systems and Networks*.

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

Maffione, V., Lettieri, G., and Rizzo, L. (2018). Cache-aware design of general-purpose single-producer-single-consumer queues. *Software: Practice and Experience*, 49.

Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state machine replication for parallelism. In *ICDCS*.

Mendizabal, O. M., Moura, R. T. S., Dotti, F. L., and Pedone, F. (2017). Efficient and deterministic scheduling for parallel state machine replication. In *IPDPS*.

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