Replicação Máquina de Estados Paralelas com Escalonamento Híbrido

Aldênio Burgos[1], **Eduardo Alchieri**[1]

[1]CIC, Universidade de Brasília, Brasília - Brasil

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. Parallel SMR uses a scheduler to allow parallel execution of some requests, improving performance. 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. Moreover, through the implementation of a payment system, this work shows that the proposed approach for parallel SMRs circumvents previously described performance limitations in the implementation of blockchains that uses an SMR as an underline building block.*

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. RMEs paralelas utilizam escalonadores que tiram proveito da semântica das requisições e permitem a execução paralela de algumas delas, melhorando assim o desempenho das aplicações. 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. Além disso, através da implementação de um sistema de pagamento, este trabalho mostra que a abordagem proposta minimiza limitações de desempenho anteriormente estudadas na implementação de blockchains que utilizam uma RME como bloco subjacente de construção.*

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]. A técnica consiste em replicar o servidor e coordenar a comunicação entre clientes e réplicas, a fim de que, partindo de um mesmo estado inicial, executem o mesmo conjunto de requisições, 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 os servidores atuais que possuem processadores com múltiplos núcleos, pois apenas um deles seria utilizado para a execução das operações.

Algumas abordagens tiram proveito da semântica das operações e empregam protocolos que suportam a execução paralela de parte das operações [Kotla and Dahlin 2004a, Marandi et al. 2014a, Mendizabal et al. 2017, Alchieri et al. 2017, Alchieri et al. 2018, Escobar et al. 2019]. Estas abordagens, chamadas de RMEs 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 RMEs 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 2004a, 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 atenderá cada partição. Seu classificador demanda pouco processamento para o escalonamento na réplica, porém, são necessárias sincronizações adicionais nos executores para tratar requisições conflitantes, o que diminui drasticamente seu desempenho na presença de conflitos [Alchieri et al. 2018].

Nesta dissertação 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.

Além disso, esta dissertação apresenta um estudo de caso sobre o uso de RMEs paralelas na implementação de *blockchains*, através da implementação de um sistema de pagamentos. De fato, RMEs tolerantes a falhas bizantinas (BFT) [Castro and Liskov 1999, Castro and Liskov 2002] são particularmente relevantes no contexto de *blockchains* permissionadas [Cachin and Vukolic 2017], por suportarem que uma fração das réplicas sejam controladas por um adversário. Estes protocolos também são usados em *blockchains* não permissionadas, onde um conjunto de réplicas é escolhido para formar um comitê que executa os protocolos de replicação [Pass and Shi 2017, Abraham et al. 2017]. Um trabalho recente [Bessani et al. 2020] demonstrou que utilizar RMEs como um bloco de construção na implementação de *blockchains* resulta em um sistema com limitação de desempenho, principalmente devido a execução sequencial das transações. Neste contexto, esta dissertação buscou estudar como a abordagem proposta para RMEs paralelas poderia minimizar a limitação de desempenho anteriormente discutida.

2. Trabalhos Relacionados

Trabalhos anteriores observaram que requisições independentes podem ser executadas concorrentemente [Schneider 1990], sendo que estas requisições predominam em muitas cargas de trabalho [Kotla and Dahlin 2004b, Marandi et al. 2014b, Marandi and Pedone 2014]. As propostas existentes para RMEs paralelas diferem na estratégia e na arquitetura utilizada para detectar e gerenciar conflitos. Estas técnicas podem ser agrupadas em três grupos.

Técnicas baseadas no conhecimento da semântica das operações. Um escalonar é utilizado para serializar requisições dependentes e encaminhar requisições independentes para execução em um *pool* de *threads* [Kotla and Dahlin 2004b, Marandi et al. 2014b, Alchieri et al. 2017, Alchieri et al. 2018]. Os escalonadores estudados nesta dissertação fazem parte deste grupo e serão melhores explorados na próxima seção.

Técnicas alheias a semântica das operações. Algumas soluções utilizam arquiteturas mais complexas para coordenar as réplicas, mas não empregam conhecimento da semântica das operações. Os sistemas Rex [Guo et al. 2014] e CRANE [Cui et al. 2015] são exemplos de soluções neste grupo. Além disso, escalonadores determinísticos, que buscam ordenar a aquisição de *locks*, também foram estudados no contexto de RMEs paralelas [Habiger et al. 2020].

Técnicas otimistas. Algumas soluções empregam estratégias otimistas para paralelizar a execução de requisições [Kapritsos et al. 2012, Marandi and Pedone 2014]. Nestas soluções, as réplicas executam os comandos conforme são entregues e verificam se chegaram a um estado consistente após a execução.

3. RMEs Paralelas com Escalonamento Híbrido

3.1. Modelo de Sistema

Assumimos o modelo de sistema usual para RMEs. Sendo assim, o sistema distribuído é composto por processos interconectados. Há um conjunto ilimitado de processos cliente

e um conjunto de n processos servidor (réplicas). Consideramos um sistema parcialmente síncrono para garantir terminação [Dwork et al. 1988]. O sistema pode ser configurado para tolerar falha por parada ou bizantina. Um processo é *correto* se não falha, ou *faltoso* caso contrário. Existem no máximo f réplicas faltosas, de $n = 2f + 1$ réplicas (falhas por parada) ou $n = 3f + 1$ réplicas (falhas bizantinas).

A RME provê consistência forte ou *linearizabilidade* [Herlihy and Wing 1990]. Os algoritmos de escalonamento que discutimos neste trabalho 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.

3.2. 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. CBASE [Kotla and Dahlin 2004a] 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.

3.3. 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.

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, regras de mapeamento de classes para executores devem ser aplicadas para garantir execuções linearizáveis [Alchieri et al. 2018]. Essas regras resultam em vários mapeamentos possíveis de classes para executores e foi modelado como um problema complexo de otimização. 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 $T = \{t_0, .., t_{n-1}\}$.

3.4. Escalonamento Híbrido

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.

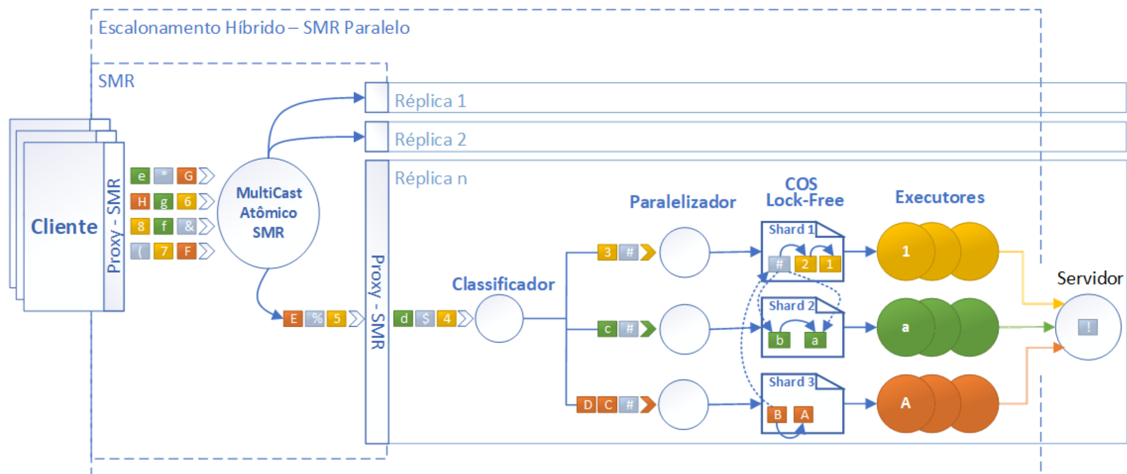


Figura 1. Escalonamento Híbrido: esquema de uma réplica

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 suas 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.

Uma descrição detalhada do protocolo que implementa este modelo de execução, juntamente com uma discussão sobre sua corretude, pode ser encontrada em [Burgos et al. 2022].

3.5. Experimentos

Realizamos uma avaliação experimental dos escalonadores com os objetivos de (1) mostrar as vantagens e desvantagens das abordagens tardia e antecipada, e (2) apresentar como o escalonador híbrido contorna esses problemas de desempenho em um sistema com estado particionado. Todas as implementações estão disponíveis no $GitHub$ em: `https://github.com/aldenioburgos/library`.

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 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[1] separadas para tolerar até 1 falha de réplica, enquanto 800 clientes foram distribuídos uniformemente em outras 4 máquinas HP[2].

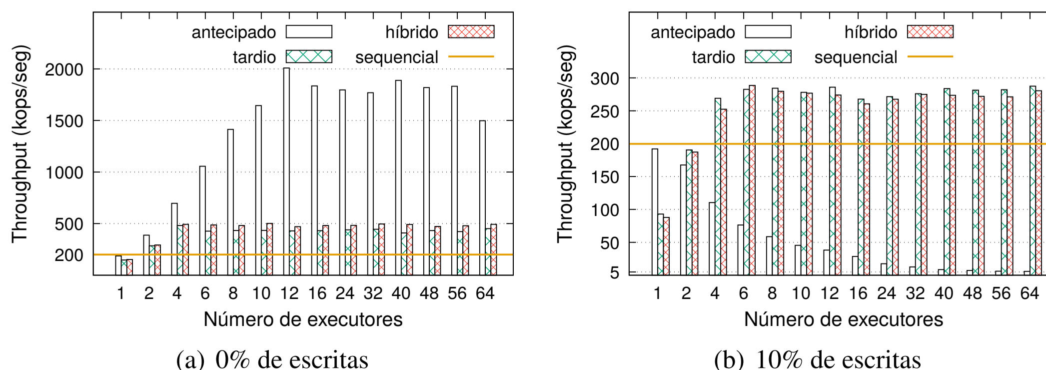
Aplicação. Nesta seção apresentaremos os resultados para dois serviços implementados, sendo um com apenas uma partição e outro multi-particionado. Resultados para outras aplicações são encontrados em [Burgos et al. 2022]. A aplicação escolhida para o serviço com uma partição foi uma lista encadeada de 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. Também configuramos o tamanho máximo do grafo de dependências com 150 entradas para as abordagens tardia e híbrida. Apresentamos apenas resultados para vazão (ou *throughput*) do sistema nos servidores. Resultados para latência e outras cargas de trabalho são encontrados em [Burgos et al. 2022]. 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. Além disso, um aumento no número de executores reduz o desempenho nas cargas com conflitos. Por outro lado, embora o escalonamento tardio 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 semelhante ao do escalonamento tardio, pois, para sistemas com apenas uma partição, ambas as abordagens

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

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

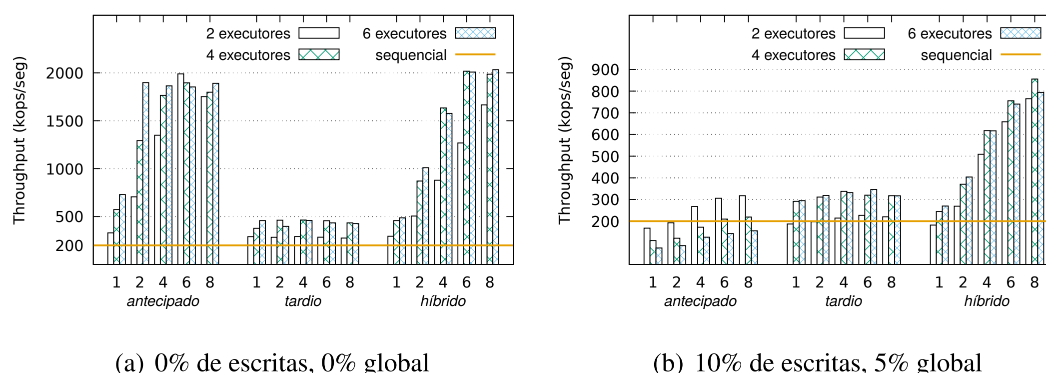
funcionam de maneira semelhante.



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

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

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.

4. Estudo de caso: uso de RMEs paralelas em *blockchains*

Através de uma aplicação de moeda digital, recentemente foi demonstrado que o modelo de execução sequencial de RMEs tradicionais afeta significativamente o desempenho do sistema, quando usamos este mecanismo como um bloco de construção na implementação de *blockchains* [Bessani et al. 2020]. O estudo aqui apresentado também utiliza uma aplicação de moeda digital, chamada de *ParallelCoin*, para demostrar que RMEs paralelas conseguem minimizar as limitações de desempenho anteriormente identificadas, pois permitem que uma parte das transações seja executada em paralelo nas réplicas, seguindo um modelo de concorrência elaborado para as operações definidas para a aplicação.

Experimentos realizados com as implementações desenvolvidas mostram que o *ParallelCoin* possui um desempenho de até $13.75\times$ o da solução sequencial, quando consideramos cargas de trabalho com operações que acessam apenas uma partição/moeda. Já para cargas de trabalho com operações endereçadas a mais de uma partição/moeda, o desempenho do *ParallelCoin* é até $7\times$ maior do que a solução com apenas execuções sequenciais. Detalhes sobre a aplicação desenvolvida para este estudo de caso, o modelo de concorrência adotado para as operações, e os resultados dos experimentos realizados com o *ParallelCoin* podem ser encontrados em [Burgos et al. 2021].

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

Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Spiegelman, A. (2017). Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In *Proceedings of the 21st International Conference on Principles of Distributed Systems*.

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., Alchieri, E., Sousa, J., Oliveira, A., and Pedone, F. (2020). From byzantine replication to blockchain: Consensus is only the beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

Bessani, A., Sousa, J. a., and Alchieri, E. E. P. (2014). State machine replication for the masses with bft-smart. In *International Conf. on Dependable Systems and Networks*.

Burgos, A., Alchieri, E., and Dotti, F. (2021). On the performance of using parallel state machine replication to implement blockchains. In *2021 10th Latin-American Symposium on Dependable Computing (LADC)*, pages 1–6.

Burgos, A., Alchieri, E., Dotti, F., and Pedone, F. (2022). Exploiting concurrency in sharded parallel state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, 33(9):2133–2147.

Cachin, C. and Vukolic, M. (2017). Blockchain consensus protocol in the wild (invited paper). In *Proceedings of the 31th International Symposium on Distributed Computing*.

Castro, M. and Liskov, B. (1999). Practical Byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation*. USENIX Association.

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

Cui, H., Gu, R., Liu, C., Chen, T., and Yang, J. (2015). Paxos made transparent. In *25th Symposium on Operating Systems Principles*.

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

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*.

Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L., and Zhuang, L. (2014). Rex: Replication at the speed of multi-core. In *European Conference on Computer Systems*.

Habiger, G., Hauck, F. J., Reiser, H. P., and Köstler, J. (2020). Self-optimising application-agnostic multithreading for replicated state machines. In *SRDS*.

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.

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about eve: execute-verify replication for multi-core servers. In *OSDI*.

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

Kotla, R. and Dahlin, M. (2004b). High throughput byzantine fault tolerance. In *DSN*.

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

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

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

Marandi, P. J. and Pedone, F. (2014). Optimistic parallel state-machine replication. In *SRDS*.

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*.

Pass, R. and Shi, E. (2017). Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *Proceedings of the 31st International Symposium on Distributed Computing*.

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