Proposta para Reparticionamento de Estado em Replicação Máquina de Estado Paralela

João Gabriel Trombeta
Universidade Federal de Santal Catarina – UFSC
joao.gabriel.trombeta@grad.ufsc.br

Odorico Machado Mendizabal
Universidade Federal de Santa Catarina – UFSC
odorico.mendizabal@ufsc.br

ABSTRACT

State Machine Replication is a widely used technique to provide fault tolerance and strong consistency. In this approach, all commands are executed sequentially throughout the replicas. Aiming to improve the system's throughput, enhanced versions were proposed, where independent commands can be executed in parallel. One arising challenge, though, is how to balance the workload between threads, while avoiding the need of synchronization between them. This extended abstract proposes a technique to schedule requests in a dynamic and efficient manner, using a workload graph to keep track of access patterns and graph partitioning to decompose and dispatch workload to the worker threads.

KEYWORDS

Computação Paralela, Replicação Máquina de Estados Paralela, Balanceamento de Carga, Corte Mínimo em Grafos, Reparticionamento

1 INTRODUÇÃO

Diversos serviços na Internet possuem rigorosos requisitos de disponibilidade. Nesse contexto, a Replicação Máquina de Estado (RME) [1, 2] é uma técnica bem estabelecida que provê tolerância a falhas e garantia de consistência forte (linearizabilidade). Em RME, réplicas do sistema inciam no mesmo estado inicial e executam os mesmos comandos na mesma ordem, de maneira determinística, fazendo com que todas atinjam os mesmos estados durante a execução.

A implementação clássica de RME possui a limitação de que comandos precisam ser executados de maneira sequencial, não tomando proveito de arquiteturas multiprocessadores. Replicação Máquina de Estado Paralela [3–6] foi então desenvolvida, onde comandos não conflitantes podem ser executados simultaneamente, aumentando a vazão de requisições processadas sem violação de consistência. Dois comandos são ditos não conflitantes caso a escrita de um não ocorra no mesmo dado em que a leitura ou escrita do outro, caso contrário são conflitantes. Para maximizar o ganho de desempenho, faz-se necessário explorar o balanceamento do trabalho entre as threads, assim como minimizar o impacto causado pela necessidade de sincronização que se dá pela existência de comandos conflitantes [7].

O balanceamento de carga em uma réplica individual deve levar em consideração a distribuição do acesso aos dados, que pode sofrer alterações com o tempo, e evitar sincronização entre threads, que degrada a performance do sistema. Trabalhos recentes estudam problemas de otimização semelhantes em contextos distribuídos (e.g., [8–10]). Resultados apresentados em [9, 10] indicam que o uso de grafos de dependência auxilia no reparticionamento do estado entre as réplicas distribuídas. Espera-se que o uso dessa técnica possa ser aplicado localmente em réplicas em RME Paralela, visando reduzir sincronizações e balancear a carga entre threads.

Nesse resumo é proposta uma técnica em que a execução de requisições referentes a cada dado é delegada a uma única thread, de maneira dinâmica e completamente transparente. Durante a execução, de acordo com o número de requisições feitas a um dado e os conflitos por elas causados, a associação entre dados e threads pode ser reestruturada, visando um melhor equilíbrio de trabalho e, consequentemente, aumento na vazão do sistema.

2 SOLUÇÃO PROPOSTA

Seguindo o modelo de Replicação Máquina de Estado Paralela, assume-se que cada réplica possui k threads trabalhadoras. Além disso, o sistema possui uma estrutura que mapeia um dado da aplicação para a thread responsável por executar as requisições que o acessam, assim como um grafo, que é utilizado para recalcular a associação entre os dados e threads buscando rebalanceamento.

2.1 Grafo de trabalho

A réplica armazena, de maneira global, um grafo não direcionado. Cada vértice (v, w) é um elemento do conjunto de vértices V, onde o dado v da aplicação foi acessado w vezes. Cada aresta pode ser descrita como (x, y, p), onde os vértices x e y estão conectados se e somente se x e y executaram comandos conflitantes p vezes, sendo $p \geq 1$. Essa estrutura foi escolhida para que, durante a fase de reparticionamento, algoritmos conhecidos em grafos possam ser usados para otimizar o reparticionamento.

Uma implementação de grafo eficiente é fundamental para que a técnica apresente um bom desempenho, uma vez que ele é acessado frequentemente e por múltiplas threads. Estruturas de dados concorrentes vêm sendo tema de estudos recentes, incluindo grafos, idealmente a estrutura deve ser livre de espera e ter tempo de acesso à vértices e arestas constante. Algumas implementações sendo estudadas foram propostas em [11–13].

2.2 Execução de requisições

Uma requisição r é acessada pela thread associada aos dados manipulados por r, seguido por uma atualização no grafo de trabalho para registrar a ocorrência do acesso e, possivelmente, conflito.

Caso requisições acessem um único dado, elas podem ser executadas de maneira paralela. Uma sincronização é necessária quando uma requisição acessa múltiplos dados, sendo pelo menos dois desses mapeados para threads diferentes. Para demonstrar a razão, suponha como exemplo uma thread t_1, responsável pelas requisições referentes ao dado a, e t_2, responsável pelas requisições em b. Suponha também o comando $write(x, y)$, que escreve o valor y em x, e $swap(w, z)$, que troca os valores de w e z. Ao receber a sequência de comandos $write(a, 2)$, $write(b, 3)$ e $swap(a, b)$, é necessário que o comando $swap(a, b)$ seja executado após t_1 e t_2 processarem suas requisições de escrita, exigindo sincronização entre t_1 e t_2

para garantir consistência. Na ocorrência de uma situação como a demonstrada cria-se uma barreira.

Para que seja feito o processamento da requisição que requer sincronização, é necessário que todas as threads envolvidas tenham processado todas as requisições que antecedem a barreira. Note que neste caso, algumas threads ficam ociosas, aguardando pela execução dessas requisições. Quando todas as threads estiverem prontas, apenas uma executa o comando e atualiza o grafo de trabalho. Em seguida, todas as threads envolvidas na sincronização podem prosseguir.

A consistência garantida pela sincronização entre threads é trivialmente atingida caso todos os dados acessados estejam associados a uma única thread, uma vez que existe a garantia de que todas as requisições envolvendo os múltiplos dados serão executadas de maneira sequencial.

A Fig. 1 mostra um exemplo de estado do sistema, onde a thread $t1$ é responsável por executar requisições sobre os dados x e y, enquanto $t2$ é responsável por operações sobre w e z. É possível observar que r_1, r_2, r_4 e r_5 acessam dados mapeados a uma única partição. A requisição r_3, entretanto, acessa dados das partições de t_1 e t_2, sendo adicionada nas filas de ambas as threads.

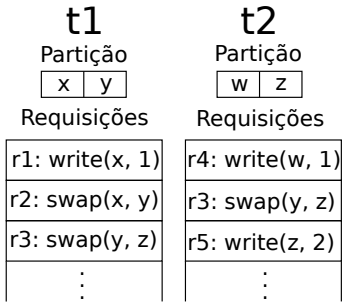


Figura 1: Exemplo de estado do sistema.

As requisições $r1$ e $r4$ podem ser executadas em paralelo, como demonstrado na Fig. 2. $t1$ pode executar $r2$, pois ambos os dados estão em sua partição, mas $t2$ não pode executar $r3$, pois precisa de um dado que está em $t1$. $t2$ então sinaliza para $t1$ que está aguardando, e quando chega a vez de $t1$ executar $r3$ ela a faz, uma vez que já recebeu o aviso de que $t2$ está aguardando. $t1$ executa $r3$ e comunica $t2$, que pode continuar a executar comandos em sua fila. Cada thread, após executar uma requisição, atualiza o grafo de trabalho.

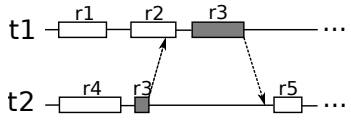


Figura 2: Execução das requisições representadas na Fig. 1.

Decorrente do modelo de execução descrito, não existe concorrência ao acessar um dado, removendo a necessidade de travas no processamento de requisições e tornando-as livres de espera. Requisições que envolvem uma única variável serão executadas apenas pela única thread a que está designada, e requisições que envolvem múltiplas variáveis serão executadas por uma única thread, caso as variáveis estejam todas mapeadas a uma mesma thread, ou todas as threads envolvidas sincronizarão de forma que apenas uma execute a requisição.

O reparticionamento é iniciado depois de disparado um certo gatilho, nele os dados adquiridos durante a execução são utilizados para criar um novo mapeamento que melhor equilibra a carga de execução das threads. Para evitar que durante a transição para o novo mapeamento duas threads executem requisições em um mesmo dado, é feita uma barreira para esperar que todas as threads finalizem requisições pendentes, depois disso a barreira é liberada e o novo mapeanto entra em vigor.

2.3 Reparticionamento dos dados

É necessário diminuir a quantidade de sincronizações entre threads, uma vez que causam ociosidade até que todas as threads atinjam a barreira. Não existindo a necessidade de sincronização, menos threads permanecerão ociosas, acarretando em um melhor aproveitamento dos recursos do sistema. Logo, é ideal que dados frequentemente acessados em conjunto sejam delegados à mesma thread.

A quantidade de sincronizações durante a execução de requisições poderia ser trivialmente resolvida fazendo com que todas as requisições sejam executadas por uma única thread, o que seria o mesmo que uma implementação de RME tradicional, onde o processamento ocorre de maneira puramente sequencial e recursos da máquina são subutilizados. Existe a necessidade de reduzir a quantidade de comandos que requerem sincronização, mas, simultaneamente, é preciso distribuir a execução de forma a não sobrecarregar algumas threads enquanto outras são subutilizadas.

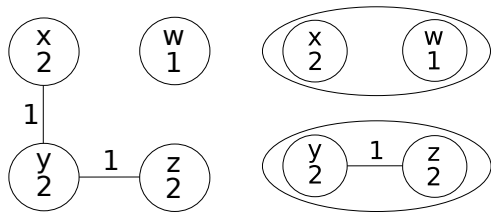
Outro fator que pode desbalancear o trabalho entre threads são aspectos da semântica da aplicação. Em redes sociais, por exemplo, um vídeo que repentinamente tornou-se viral terá um grande pico em seu acesso, sobrecarregando a thread responsável por seus dados. É necessário que o reparticionamento seja feito de maneira dinâmica, para levar em consideração as mudanças causadas por aspectos inerentes à aplicação que não podem ser previstos de maneira estática.

O reparticionamento é, portanto, um problema de otimização, onde os dados devem ser distribuídos de maneira a diminuir a necessidade de sincronizações e, ao mesmo tempo, balancear o trabalho entre threads de maneira homogênea. Tendo em vista que se quer particionar os dados em k threads com a restrição do balanceamento de carga, é possível reduzir esse problema para o problema de particionamento de um grafo em k partições com restrições, onde a restrição é o balanceamento de carga entre as partições. Especificamente, o problema pode ser descrito como o corte mínimo em k partições, que consiste em encontrar uma forma de cortar um grafo em k partições de modo que a soma dos pesos das arestas cortadas (que atravessam partições) seja o menor possível, com a adição da restrição que a soma dos pesos dos vértices em cada partição seja similar.

O grafo de trabalho é modelado de forma que seja possível usá-lo como entrada em um algoritmo de corte mínimo. O corte será feito preferencialmente em arestas de menor peso, separando dados que conflitaram poucas vezes, e manterá arestas de alto peso, isso é,

permanecerão conectados na mesma partição dados que conflitaram muitas vezes. Além disso, a soma dos valores dos vértices em cada componente do grafo devem ser similares, como o valor de um vértice é a quantidade de vezes que ele foi acessado, a quantidade de acesso aos dados em cada partição serão similares.

Depois de aplicado o algoritmo de particionamento no grafo de trabalho, os dados são redistribuídos entre as threads. Ao final do reparticionamento, cada vértice será designado a uma única partição, e cada partição será designada a uma única thread. A Fig. 3(b) mostra um possível resultado do reparticionamento do grafo representado na Fig. 3(a). Nesse exemplo o grafo foi cortado em duas partições, uma de tamanho 3 e outra de tamanho 4, removendo uma aresta de peso 1 no processo. Após o reparticionamento, uma partição composta por x e w é designada a uma das threads, e uma composta por y e z à outra. O corte mínimo admite mais de uma solução, o grafo poderia ser cortado de outras maneiras que resultariam em partições de mesmo tamanho e com a soma dos pesos das arestas removidas com o mesmo valor.



(a) Após execução das requisições. (b) Partições definidas após reparticionamento.

Figura 3: Exemplo de grafo de trabalho.

O reparticionamento representa um aspecto crítico da execução, é necessário que ele não interfira no processamento de requisições e que consiga reagir a mudanças na carga de trabalho em tempo hábil. Técnicas eficientes de particionamento são bastante pesquisadas por sua utilidade em diversas áreas, elas podem ser divididas em algoritmos de particionamento estático, em fluxo, e para bancos de dados de grafos distribuídos [14]. Algoritmos para o particionamento de grafos em fluxo são uma possibilidade a ser considerada, por possuírem menor complexidade e menor consumo de memória, e ainda assim produzirem boas partições. Além disso, algoritmos online [15] também são promissores, por adaptar as partições dinamicamente de acordo com as mudanças feitas no grafo, sem a necessidade de recalcular as partições por inteiro.

3 CONSIDERAÇÕES FINAIS

Por permitir certo grau de paralelismo, Replicação Máquina de Estado Paralela trás grandes ganhos de desempenho em comparação a implementações tradicionais de RME, porém o sucesso desta técnica depende da carga de trabalho das aplicações e das estratégias de balanceamento de carga. Aqui é proposta uma técnica onde dados são divididos em partições e cada thread é responsável por uma partição, para que requisições sejam distribuídas entre as threads de forma a balancear o trabalho e reduzir a necessidade de sincronizações.

Um aspecto da proposta que ainda deve ser explorado é o gatilho que dará início ao reparticionamento, algumas opções são ser disparado por um cronômetro ou ser disparado após um desbalanceamento acima do aceitável ser detectado.

É esperado que ao implementar o método de reparticionamento aqui proposto o sistema experiencie um aumento na vazão de requisições, mesmo que os padrões de acesso a dados da aplicação sofram alterações durante sua execução.

AGRADECIMENTOS

O presente trabalho foi realizado com o apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq - Brasil.

REFERÊNCIAS

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

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[3] Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584. IEEE, 2004.

[4] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. {All about Eve}: execute-verify replication for multi-core servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, pages 237–250, 2012.

[5] Parisa Jalili Marandi, Carlos Eduardo Bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 368–377. IEEE, 2014.

[6] O. M. Mendizabal, R. S. T. D. Moura, F. L. Dotti, and F. Pedone. Efficient and deterministic scheduling for parallel state machine replication. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 748–757, 2017. doi: 10.1109/IPDPS.2017.29.

[7] Eduardo Alchieri, Fernando Dotti, Odorico M Mendizabal, and Fernando Pedone. Reconfiguring parallel state machine replication. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 104–113. IEEE, 2017.

[8] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[9] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. Dynastar: Optimized dynamic partitioning for scalable state machine replication.

[10] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[11] Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. A simple and practical concurrent non-blocking unbounded graph with linearizable reachability queries. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 168–177, 2019.

[12] Nikolaos D Kallimanis and Eleni Kanellou. Wait-free concurrent graph objects with dynamic traversals. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[13] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. Boosting concurrency in parallel state machine replication. In *Proceedings of the 20th International Middleware Conference*, pages 228–240, 2019.

[14] Chayma Sakouhi, Abir Khaldi, and Henda Ben Ghezal. An overview of recent graph partitioning algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 408–414. The Steering Committee of The World Congress in Computer Science, Computer …, 2018.

[15] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2012.