# Explorando Concorrência em Aplicações Chave-Valor com Replicação de Máquinas de Estado

**Erick Pintor**[1], **Fernando Luís Dotti**[1]

[1]Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

***Abstract.*** *In the article "Boosting concurrency in Parallel State Machine Replication", the authors present various parallel approaches to increase throughput in SMR systems. In this article we aim to expand on their work by exploring a workload suitable for a high degree of parallelism. We also propose an analogous solution of lower implementation complexity by taking advantage of Java standard library's concurrent data-structures. We observe the system's throughput with this new workload, and the experiments suggest the equivalence of both implementations.*

***Resumo.*** *No artigo "Boosting concurrency in Parallel State Machine Replication" os autores apresentam diferentes abordagens paralelas para aumentar a vazão de sistemas do tipo RME. Neste artigo, buscamos expandir o trabalho destes autores explorando uma carga de trabalho propícia a alto grau de paralelismo. Sugerimos ainda uma solução análoga com menor complexidade de implementação por utilizar estruturas de dados concorrentes disponíveis na biblioteca padrão da linguagem Java. Observamos a vazão do sistema com esta nova carga de trabalho e os experimentos executados sugerem a equivalência de ambas implementações.*

## 1. Introdução

Replicação de Máquina de Estados (RME) [Lamport 1978, Schneider 1990], é uma abordagem conhecida para a implementação de sistemas tolerantes a falhas. Em um sistema RME, o estado do sistema é replicado em várias máquinas, nas quais alterações no mesmo ocorrem através de comandos que são executados em ordem preestabelecida.

No artigo "Boosting concurrency in Parallel State Machine Replication" [Escobar et al. 2019], os autores apresentam abordagens paralelas que buscam aumentar a vazão de sistemas RME através da sobreposição temporal na execução de comandos chamados "não-conflitantes". Neste artigo, nos baseamos em uma das abordagens proposta pelos autores e buscamos experimentar com uma carga de trabalho mais propícia a um alto grau de paralelismo. Ainda, propomos uma solução análoga que utiliza estruturas de dados para programação concorrente disponíveis na biblioteca padrão da linguagem Java, sendo esta de menor complexidade de implementação. Por fim, testamos ambas implementações a fim de comparar seu desempenho.

Na seção 2 é introduzida a fundamentação teórica que dá suporte à pesquisa. Na seção 3 apresentamos a motivação do trabalho. Na seção 4 discorremos sobre as implementações desenvolvidas. Na seção 5 relatamos os experimentos executados. A seção 6 apresenta os resultados obtidos. Seção 7 propõe trabalhos futuros relacionados ao tema de pesquisa e concluímos o trabalho na seção 8.

## 2. Fundamentação Teórica

### 2.1. Boosting concurrency in Parallel State Machine Replication

Em "Boosting concurrency in Parallel State Machine Replication", os autores introduzem uma estrutura de dados para detecção de conflitos baseada em um grafo dirigido acíclico (DAG). Nesta estrutura, comandos são representados na forma de nodos e conflitos são representados na forma de arestas que ligam comandos conflitantes. A direção das arestas determina sua ordem de execução, esta que respeita a ordenação total de comandos conflitantes. Por exemplo, considere a sequência de comandos A, B, C, D, e E onde apenas C e D conflitam com B. A Figura 1 demonstra o DAG para detecção de conflitos correspondente. Neste exemplo, podemos observar que comandos A, B e E podem ser executados em paralelo, no entanto, C e D devem ser executados apenas após a execução de B. Apesar disto, após a execução de B, C e D podem ser executados em paralelo.

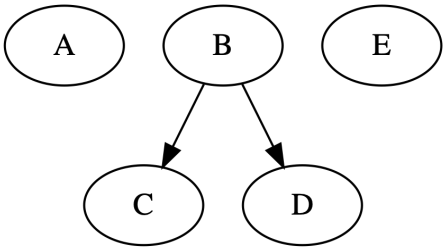


**Figura 1. DAG com comandos C e D conflitantes com B, demais não-conflitantes**

Conforme os dados apresentados no artigo, dentre as abordagens apresentadas, a implementação dita *lock-free* demonstrou prover maior vazão tanto em relação ao aumento do número de *threads* quanto ao grau de conflito entre comandos.

### 2.2. Estruturas de Dados para Concorrência em Java

A linguagem Java introduziu na sua versão 5 o pacote *java.util.concurrent* contendo diversas estruturas de dados para programação concorrente. Dentre elas, a interface *ExecutorService*, que desacopla operações concorrentes de seu contexto de execução [Urma et al. 2018, 2.4.3]. De forma sucinta, uma tarefa é definida por um conjunto de instruções agnósticas ao modelo de paralelismo adotado. O contexto de execução, por sua vez, não tem conhecimento sobre o conjunto de instruções que irá executar, ficando responsável apenas pela alocação justa dos recursos disponíveis para a execução das tarefas submetidas a ele. Ao desacoplar tarefas do seu contexto de execução, esta abordagem possibilita que tarefas oriundas de diferentes algoritmos compartilhem os mesmos recursos computacionais ao serem submetidas ao mesmo contexto.

Diversas implementações da interface *ExecutorService* estão disponíveis, cada uma com suas próprias características de escalonamento e desempenho. Destaca-se dentre elas a classe *ForkJoinPool*, que implementa um contexto de execução baseado em um conjunto de *threads* com furto de trabalho. Neste contexto, em alto nível, *threads* executam tarefas de sua própria responsabilidade, ou furtam tarefas de outras *threads* para executar, visando assim maximizar a utilização de seu tempo de *CPU*.

Na versão 8 da linguagem Java foi introduzida a classe *CompletableFuture*, utilizada para compor tarefas concorrentes. Sua semântica permite criar tarefas a serem exe-

cutadas em um determinado contexto de execução a partir de uma interface que permite combinar tarefas assíncronas sem necessidade de sincronização no nível da aplicação.

Ambas estruturadas de dados são implementadas utilizando técnicas de programação não-bloqueantes. A utilização de tais estruturas reduz significativamente a complexidade de implementação de algoritmos concorrentes, reduzindo assim espaço de codificação onde problemas possam ser introduzidos e facilitando a adoção ou adaptação de tais algoritmos para aplicações de uso específico.

## 3. Motivação

### 3.1. Carga de Trabalho

No artigo "Boosting concurrency in Parallel State Machine Replication", os autores realizaram experimentos com uma carga de trabalho baseada em uma lista encadeada. Nesta carga de trabalho, ao enviar um comando de inserção à lista, todos os comandos concorrentes são linearizados após a inserção. Esta necessidade se dá devido ao fato de que: inserções sempre ocorrem no mesmo ponto da lista, sua cauda, o que requer a sincronização entre comandos concorrentes a fim de garantir a consistência dos dados inseridos; e, uma inserção na lista pode incluir o elemento que um comando de leitura busca, logo, a execução paralela de ambos poderia resultar em respostas diferentes dependendo da ordem de execução dos comandos.

Ao caracterizar quaisquer comandos de escrita como conflitantes com todos os outros comandos concorrentes, se reduz a possibilidade de paralelismo. A aplicação passa por períodos de alto paralelismo quando executa apenas leituras, e baixo paralelismo ao executar comandos de escrita.

Sistemas SGBD distribuídos baseados em estruturas de dados chave-valor tendem a reduzir o escopo dos comandos conflitantes a comandos que operam sobre uma mesma chave, ou conjunto de chaves [Thomson et al. 2012]. Neste cenário, comandos que operam sobre chaves distintas podem ser executados em paralelo, enquanto apenas comandos que operam sobre as mesmas chaves são passíveis de linearização. Tal cenário é, portanto, mais propício a um alto grau de paralelismo.

### 3.2. Estruturas de Dados

A implementação proposta pelos autores faz uso de primitivas da linguagem Java para implementar um modelo produtor-consumidor com um conjunto configurável de *threads* para a execução de comandos em paralelo. Sistemas altamente concorrentes são compostos de diversos algoritmos que necessitam de *threads* para sua execução. Plataformas de *hardware* possuem um limite de desempenho em relação à sobrecarga de seu escalonador ao aumentar o número de *threads* no sistema [Urma et al. 2018, 15.1.2]. Logo, dispor de um número fixo de *threads* por algoritmo pode ser um limitador para a adoção de tais implementações. É desejável, portanto, que algoritmos de mesma característica computacional possam compartilhar um conjunto de *threads* a fim de maximizar o uso da *CPU* e minimizar a sobrecarga do escalonador.

Ainda, a implementação do grafo de dependências sugerido é uma implementação relativamente complexa em termos de reprodução uma vez que cada operação sobre a estrutura de dados é cuidadosamente estudada para evitar problemas de concorrência e

prover alto desempenho. Tais técnicas podem ser de difícil adoção dado que quaisquer adaptações necessárias para adequar a implementação às necessidades de uma aplicação específica podem invalidar invariantes do algoritmo. A implementação de um algoritmo análogo que utiliza estruturas de dados concorrentes da biblioteca padrão Java pode oferecer um modelo mental mais simples de ser adaptado tanto em Java quanto em linguagens semelhantes.

## 4. Implementação

### 4.1. Carga de Trabalho Chave-Valor

A fim de testar a implementação paralela sugerida pelos autores em uma estrutura de dados propícia a alto grau de paralelismo, foi desenvolvida uma carga de trabalho baseada em dicionário chave-valor, ambos números inteiros positivos. A aplicação desenvolvida é capaz de executar dois tipos de comandos:

- Incremento: operação de escrita que incrementa o valor atual dada uma chave;
- Leitura: operação que retorna o valor atual dada uma chave.

Nesta carga de trabalho, dois comandos são ditos conflitantes se, e somente se, ambos operam sobre a mesma chave e pelo menos um deles é um comando de escrita. Sendo assim, apenas operações sobre a mesma chave são passíveis de linearização.

A carga de trabalho inicializa um dicionário chave-valor com um número configurável de chaves, todas com valor inicial zero. No artigo original, os autores estabelecem uma relação de tamanho da carga de trabalho com o custo da execução de cada comando, sendo este custo vinculado à complexidade de localizar elementos em uma lista encadeada conforme a lista cresce em tamanho ($\mathcal{O}(n)$). Visto que acessos a um dicionário chave-valor tem complexidade $\mathcal{O}(1)$, variar o tamanho da carga de trabalho em número de chaves não afeta o custo da execução de comandos. Neste artigo, a carga de trabalho desenvolvida possui uma configuração de custo de execução por comando em nanosegundos, possibilitando assim variar o tempo de serviço a fim de simular diferentes cargas de trabalho.

Um gerador aleatório de comandos foi desenvolvido no qual a distribuição aleatória de chaves é determinada pela configuração do desvio padrão em uma curva aleatória Gaussiana [Gau 2020], possibilitando assim variar a dispersão de chaves utilizadas nos experimentos. A figura 2 demonstra a estimativa de densidade kernel [KDE 2020] das chaves geradas dada a taxa de dispersão configurada. Quanto menor a taxa de dispersão, maior a densidade de comandos gerados para um subconjunto de chaves e, por consequência, gera maior linearização. Ao aumentar a taxa de dispersão, a densidade relativa de cada chave diminui e a abrangência do subconjunto de chaves geradas aumenta, possibilitando maior paralelismo.

### 4.2. Implementação com Estruturas de Dados Concorrentes em Java

Sugerimos uma implementação análoga à implementação *lock-free* proposta anteriormente, porém utilizando estruturas de dados concorrentes disponíveis na biblioteca padrão da linguagem Java, sendo elas *CompletableFuture* e *ForkJoinPool*.

Ao optar pela adoção da estrutura *CompletableFuture*, passamos a representar o grafo de dependências como uma estrutura lógica, ou seja, não há necessidade de codificar o grafo, seus nodos, arestas, e a gerência da estrutura de dados como parte da

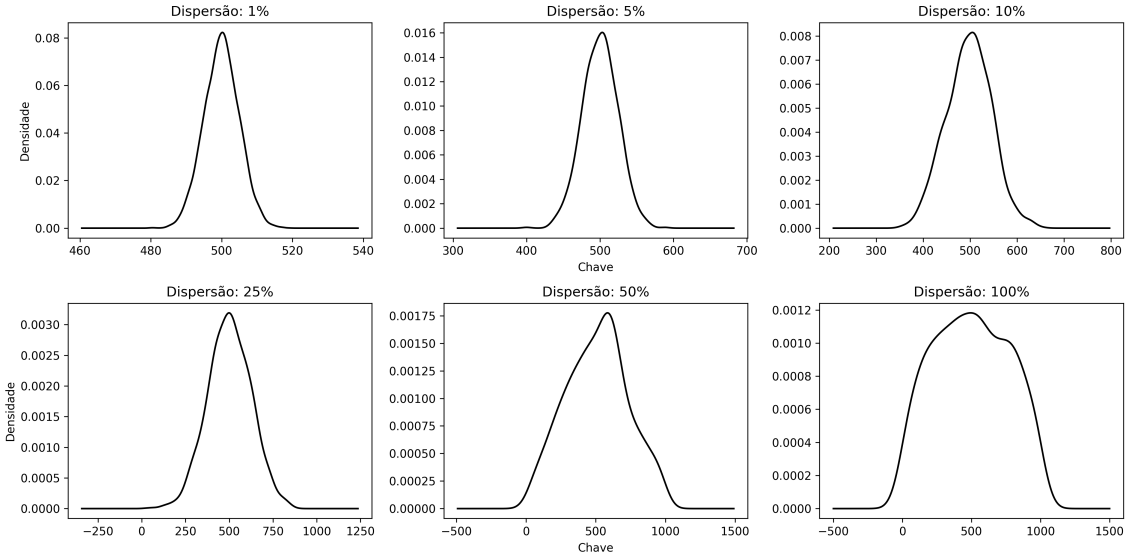


**Figura 2. Densidade de chaves dada taxa de dispersão (chaves entre [0,1000])**

aplicação. Tais responsabilidades ficam a cargo da estrutura de dados uma vez respeitada sua semântica, o que reduz significativamente a complexidade de implementação.

Neste algoritmo, representamos nodos do grafo de dependência com instâncias de *CompletableFuture*. Arestas que conectam nodos dependentes são representadas pela composição lógica das dependências com seus comandos dependentes. Desta forma, tarefas dependentes são executadas apenas após suas dependências, garantindo a linearização de comandos conflitantes.

## 5. Experimentos

Foram executados dois experimentos: (a) vazão por implementação e numero de *threads*; e (b) vazão por implementação e taxa de dispersão de chaves. Em (a), buscamos comparar a vazão da implementação *lock-free* com sua implementação análoga em relação ao número de *threads* disponíveis. Neste experimento utilizamos 100% de dispersão entre chaves e 0% de comandos de escrita. Em (b), buscamos avaliar a vazão de cada implementação ao variar a taxa de conflito entre comandos. Para tal fim, fixamos em 16 o número de *threads*, utilizamos 100% de comandos de escrita e variamos a taxa de dispersão de chaves.

Os experimentos foram executados em *hardware* proprietário, ao qual apenas os pesquisadores têm acesso. O perfil das máquinas utilizadas foi:

- Servidores: Dell PowerEdge R815, 16 cores AMD Opteron 6366HE de 1.8GHz e 128GB de memória RAM;
- Clientes: HPSE1102, 4 cores Intel Xeon L5420 de 2.5GHz e 8GB de memória RAM.

Todas as máquinas utilizadas foram configuradas com Ubuntu Linux na versão 18.04, máquina virtual Java de 64 bits na versão 11.0.7 e conexão de rede de até 1Gb/s. Apesar do uso de *hardware* proprietário, não há dependências que impeçam a reprodução

dos experimentos em outro *hardware*, desde que mantida a versão mínima da máquina virtual Java utilizada.

Todos os experimentos utilizam três máquinas-servidoras e três máquinas clientes. Cada experimento foi executado com 3 cargas de trabalho: leve, moderada, e pesada, variando o custo de execução por comando (em microsegundos) de $5\mu$, $50\mu$ e $500\mu$, respectivamente. Cada máquina cliente gera 50 comandos para chaves aleatórias, respeitando a configuração de taxa de dispersão de chaves e a porcentagem de comandos de escrita. Cada cliente envia os comandos gerados para uma máquina-servidora e aguarda o término da execução antes de enviar novos comandos.

## 6. Resultados

A figura 3 apresenta a vazão, em número de requisições por segundo, por implementação e número de *threads*. A vazão obtida segue aproximadamente a mesma curva de desempenho, com exceção da carga de trabalho pesada com 64 *threads*. No entanto, a implementação análoga se mostra equivalente à implementação *lock-free* na maior parte dos casos observados.

Com relação à carga de trabalho, é possível observar que há um limite de aproximadamente 500 mil comandos por segundo alcançado pela carga leve de trabalho. Nas demais cargas de trabalho, é possível observar melhor vazão com o aumento do número de *threads*.

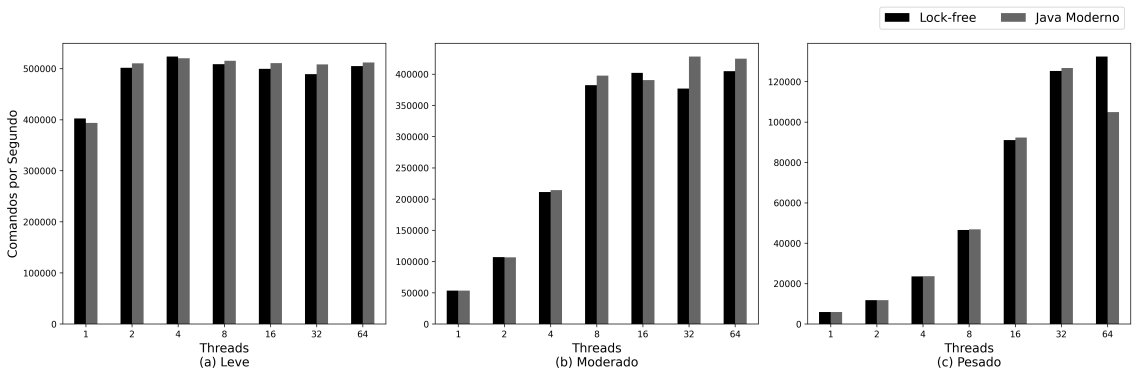


**Figura 3. Vazão por implementação e número de *threads***

A figura 4 apresenta a vazão por implementação e porcentagem de dispersão de chaves. Com exceção da carga pesada ao executar com 1% de dispersão de chaves, nos demais casos a curva de desempenho segue aproximadamente a mesma para ambas implementações. A figura 5 apresenta a vazão obtida pela por implementação proposta no artigo original. Ao observarmos a vazão da implementação *lock-free* com 100% de conflito, experimento análogo ao apresentado na figura 4, podemos observar que o maior grau de paralelismo obtido com a carga de trabalho chave-valor aumenta a vazão do sistema.

A variação na taxa de dispersão de chaves tem pouco impacto na maior parte dos casos observados. Há fortes indícios, portanto, de que esta é uma abordagem interessante para aplicações do tipo chave-valor de propósito geral, visto que é capaz de prover alta vazão independentemente da densidade de acesso às chaves do sistema.

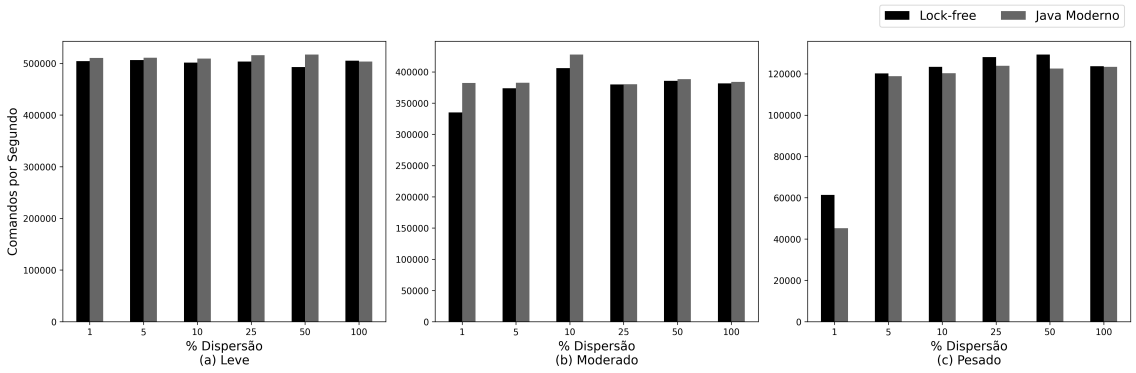


**Figura 4. Vazão por implementação e taxa de dispersão de chaves**

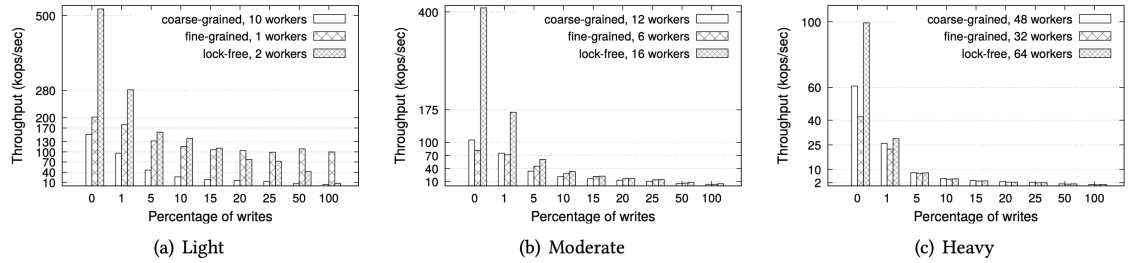


**Figura 5. Vazão por implementação e taxa de conflito (artigo original)**

Com exceção dos pontos discrepantes mencionados, dos quais novas análises são necessárias para verificar o motivo de tal discrepância, na maior parte dos casos observados ambas implementações se mostram equivalentes.

## 7. Contribuições

Durante a pesquisa, identificamos e corrigimos [Pintor 2020b] dois problemas relacionados a acesso concorrente, não sincronizado, a endereços compartilhados de memória. Um dos problemas ocorria no próprio algoritmo *lock-free* e o outro na plataforma na qual o algoritmo foi implementado. Ambos impediam o progresso do sistema e foram encontrados ao aumentar o grau de paralelismo do mesmo.

Contribuições futuras relacionadas ao tema de pesquisa podem incluir a análise dos pontos discrepantes relacionados à vazão do sistema, mencionados na seção 6, além de repetir os experimentos buscando encontrar medidas de intervalo de confiança dos dados coletados.

Ainda, a carga de trabalho desenvolvida neste artigo pode servir de base para a caracterização de uma carga de trabalho real, como por exemplo, sistemas de vendas *online*, contagem de votos, enquetes, entre outros. Tal caracterização pode fornecer dados concretos sobre o desempenho destes algoritmos para casos de uso conhecidos e documentados na indústria, visando identificar pontos de melhoria ou solidificar tais algoritmos como o estado da arte na replicação de estado paralelo.

## 8. Conclusão

Neste artigo buscamos expandir o trabalho desenvolvido pelos autores de “Boosting concurrency in Parallel State Machine Replication”, avaliando o desempenho de sua

implementação *lock-free* em uma carga de trabalho propícia a alto grau de paralelismo. Buscamos ainda oferecer uma implementação alternativa utilizando estruturas de dados para programação concorrente em Java a fim de reduzir sua complexidade de implementação, adaptação e adoção por aplicações específicas que podem se beneficiar de tais técnicas.

Testamos ambas implementações com a nova carga de trabalho desenvolvida, demonstrando sua equivalência para a maior parte dos casos observados e sua alta vazão para diferentes cargas de trabalho. Implementações desenvolvidas para este artigo, assim como os dados coletados durante os experimentos, estão disponíveis em plataforma de código aberto, com o nome "parallel-SMR" [Pintor 2020a].

## Referências


(2020). Gaussian Function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Gaussian_function&oldid=964162137`. Acessado em: 23/06/2020.

(2020). Kernel Density Estimation — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Kernel_density_estimation&oldid=956274500`. Acessado em 24/06/2020.

Escobar, I. A., Alchieri, E., Dotti, F. L., and Pedone, F. (2019). Boosting concurrency in parallel state machine replication. In *Proceedings of the 20th International Middleware Conference*, pages 228–240.

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

Pintor, E. (2020a). Parallel SMR library. `https://github.com/erickpintor/parallel-SMR`. Acessado em: 23/06/2020.

Pintor, E. (2020b). Patch - Parallel SMR library — fix dead-locks and timeouts. `https://github.com/erickpintor/parallel-SMR/commit/f4f4c1ab7328fc0970da851fd6eae67f5c8991d3`. Acessado em: 25/06/2020.

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

Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12.

Urma, R.-G., Mycroft, A., and Fusco, M. (2018). *Modern Java in Action*. Manning Publications.