Investigando Gerenciamento de Contenção em um Sistema de Memória Transacional Distribuída

Rafael Rutz dos Santos[1], **Gerson Cavalheiro,** [1] **André Rauber Du Bois**[1]

[1]Programa de Pós-Graduação em Computação – Universidade Federal de Pelotas

rddsantos, gerson.cavalheiro, dubois@if.ufpel.edu.br

Abstract. *Transactional Memory (TM) is an abstraction for the synchronization of concurrent systems, where shared data is accessed through transactions. To guarantee the consistency of shared data, a transaction only commits if there are no conflicts during its execution, which are resolved by the Contention Manager (CM), who can implement different policies. Distributed Transactional Memory (DTM), in turn, is the adaptation of TM for distributed systems. In this paper, contention management is investigated in DTM systems through eight different policies that were adapted and integrated into Transactional RMI (TRMI), a DTM system based on RMI. Tests were run on two applications, and the results show the impact of the chosen CM policy on their overall performance.*

Resumo. *Memória Transacional (MT) é uma abstração para a sincronização de sistemas concorrentes, onde dados compartilhados são acessados por meio de transações. Para garantir a consistência destes dados, uma transação só é concluída se não houver conflitos durante sua execução, os quais são resolvidos pelo Gerenciador de Contenção (GC), que pode usar diferentes políticas. A Memória Transacional Distribuída (DTM), por sua vez, é a adaptação da MT para sistemas distribuídos. Neste artigo, o gerenciamento de contenção é investigado em sistemas DTM por meio de oito políticas diferentes, que foram adaptadas e integradas ao Transactional RMI (TRMI), um sistema DTM baseado em RMI. Testes foram realizados em duas aplicações, e os resultados mostram o impacto da política de GC escolhida no desempenho geral dos sistemas.*

1. Introdução

Memórias Transacionais (MTs) são uma abstração de programação desenvolvida inicialmente para facilitar a programação de sistemas concorrentes. Usando MTs, o programador acessa seções criticas por meio de transações, parecidas com transações de bancos de dados. Apesar de realizar diversas operações, as transações de memória têm comportamento aparentemente atômico, dando a impressão de execução em um momento único no tempo [Harris et al. 2010]. Nesse sentido, para garantir a consistência dos dados, as ações tomadas por transações são efetivadas apenas se executam sem nenhum conflito, caso contrário, são abortadas e executadas novamente. Memórias Transacionais tem sido utilizada não somente em CPUs, mas também em outras arquiteturas como GPUs [Shen et al. 2020] e Sistemas Distribuídos [Zhang and Ravindran 2009, Siek and Wojciechowski 2016, Busch et al. 2022].

Este trabalho foca em MTs para arquiteturas distribuídas, i.e., *Distributed Transactional Memory* (DTM), onde a abstração de MT é usada para a sincronização de tarefas

que estejam sendo executadas em redes de computadores, cada qual com seu espaço de memória [Guerraoui and Romano 2015]. Em um sistema de MT, incluindo os sistemas de DTM, as diversas transações são executadas concorrentemente e faz-se necessário o uso de mecanismos de gerenciamento de conflitos, os quais devem decidir o que fazer quando duas transações concorrentes conflitam no acesso de um mesmo dado. Os mecanismos responsáveis por gerenciar conflitos em uma MT são chamados de Gerenciadores de Contenção (GCs) (ou *Contention Managers* - CMs). Os GCs são responsáveis por decidir qual transação deve prosseguir, e qual deve ser abortada (ou atrasada), de forma a garantir a consistência dos dados e a progressão do sistema de Memória Transacional. Apesar da importância dos GCs para as MTs, existem poucos trabalhos que investigam diferentes politicas de gerenciamento de contenção em ambientes distribuídos.

Dessa forma, o objetivo deste trabalho é adaptar e implementar, para o contexto distribuído, um conjunto de políticas de GC originalmente desenvolvidas para MTs que rodam em ambientes *multicore*. As contribuições deste artigo são:

- A investigação e adaptação de diferentes GCs clássicos da literatura, comumente aplicados e desenvolvidos para sistemas concorrentes/multiprocessados, para um modelo de DTM. No caso, foi utilizada uma DTM denominada Transactional RMI (TRMI) [Ramos et al. 2023], cujo objetivo é estender o RMI (Remote Method Invocation) para a execução de transações que envolvam objetos distribuídos. O sistema desenvolvido para este trabalho permite o carregamento e configuração de forma dinâmica dos GCs implementados.
- Um conjunto de experimentos usando duas aplicações, uma aplicação sintética[Ramos et al. 2023], que permite simular diferentes cenários de transações, e uma nova aplicação implementada para este trabalho, uma Tabela Hash Distribuída (DHT). As duas aplicações foram executadas com diferentes parâmetros e em diferentes cenários, demonstrando que a escolha da política de GC tem forte impacto no desempenho das aplicações.

As aplicações e o sistema TRMI modificado com os novos GCs desenvolvidos para este trabalho encontram-se no GitHub do Projeto.[1]

O texto está organizado da seguinte forma: na Seção 2 são apresentados conceitos de MT e DTM. Na Seção 3 são discutidos detalhes sobre o TRMI, os GCs implementados e suas adaptações para integração ao TRMI, possibilitando seu carregamento dinâmico. Já na Seção 4, são apresentadas as aplicações usadas para testes destes GCs, e os resultados obtidos sob diferentes cenários usando diversos GCs, além de considerações sobre estes resultados. Na Seção 5 são apresentados alguns dos trabalhos anteriores sobre DTMs que trataram de GCs. Por fim, a Seção 6, apresenta conclusões quanto ao presente trabalho, e propostas de trabalhos futuros.

2. Memórias Transacionais Distribuídas

As MTs podem ser adaptadas para abstraírem a sincronização de aplicações distribuídas, sendo denominadas, nesse contexto, como Memórias Transacionais Distribuídas (DTMs) [Guerraoui and Romano 2015]. A pesquisa sobre DTMs parte da necessidade por sistemas mais performáticos e com garantias robustas de consistência para grandes aplicações

[1] https://github.com/raf98/DSTM

escaláveis, sobre as quais um grande volume de dados é processado, especialmente num contexto de popularização da computação em nuvem [Guerraoui and Romano 2015]. Sistemas de DTM são classificados pelo seu modelo de execução, podendo variar entre modelos de fluxo de controle, ou *control-flow*, de fluxo de dados, ou *data-flow*, ou que utilizem um híbrido desses dois modelos. No *control-flow*, os objetos são fixos aos nodos da rede e as transações ou são movidas entre os nodos ou acessam os objetos compartilhados por meio de chamadas de procedimento remoto, ou RPCs (*Remote Procedure Calls*)[Ramos et al. 2023]. Já em DTMs *data-flow*, os objetos é que são movidos entre os nodos, os quais devem adquirir posse sobre um objeto antes de poderem realizar operações sobre este. Ainda, quanto aos modelos híbridos, estes trabalham com o movimento ou dos objetos ou das transações a depender da situação [Busch et al. 2018].

3. Gerenciadores de Contenção para Memória Transacional Distribuída

Esta Seção, primeiramente descreve o sistema de Memória Transacional Distribuída utilizado neste trabalho, o Transactional RMI (Seção 3.1). Em seguida, na Seção 3.2, a principal contribuição deste trabalho é apresentada, ou seja, os GCs estudados e adaptados para execução junto ao TRMI.

3.1. TRMI - Transactional RMI

O Transactional RMI (TRMI) [Ramos et al. 2023] é um sistema DTM *control-flow* que estende o RMI do Java permitindo transações envolvendo objetos distribuídos. O sistema adapta algoritmos de Memória Transacional do tipo *obstruction-free* para execução em um contexto distribuído. Não sendo baseado em *locks*, como muitos dos trabalhos apresentados anteriormente sobre DTMs, e.g., [Saad Ibrahim 2011, Saad and Ravindran 2012, Saad and Ravindran 2011b], o TRMI provê uma garantia de progresso *obstruction-free*, que previne a ocorrência de *deadlocks*.

No TRMI, objetos são encapsulados em objetos transacionais `TMObjServer` nos servidores, sendo publicados na rede dessa forma. Clientes, por sua vez, buscam esses objetos na rede por meio de uma chamada de *lookup*. Tal chamada retorna uma referência remota ao objeto transacional presente em um servidor por meio do registro do RMI [Ramos et al. 2023]. Na execução das transações, tais objetos remotos são abertos para leitura ou para escrita. No TRMI, quando uma transação ativa, considerada atacante, tenta obter posse de um objeto transacional para escrita, verifica-se o status da última transação que o modificou. Caso a transação anterior tenha tido suas modificações descartadas (ABORTED) ou efetivadas (COMMITTED), a última versão efetivada do objeto é disponibilizada para a transação atacante. Caso, entretanto, o status seja ACTIVE, isso indica que outra transação ativa, considerada inimiga, já possui posse de escrita sobre esse objeto, ou seja, há um conflito entre as duas transações. Nesse caso, a transação atacante chama seu GC, passando como parâmetro a transação inimiga. O GC, por sua vez, deve decidir, de acordo com alguma política de contenção, qual das transações deverá prosseguir com sua execução e qual deverá ser abortada e/ou atrasada.

3.2. Gerenciadores de Contenção Implementados

Os mecanismos responsáveis por gerenciar conflitos em uma MT são chamados de Gerenciadores de Contenção (GCs), no inglês, *Contention Managers* (CMs). Os GCs, em caso de conflito, decidem qual transação deve prosseguir (a ganhadora), e qual deve ser

abortada ou atrasada (a perdedora) [Guerraoui and Romano 2015], de forma a garantir a consistência dos dados em um sistema de Memória Transacional e, também, a progressão das transações [Herlihy 2012]. Diferentes tipos de abordagem podem ser utilizados na resolução de conflitos entre transações utilizando Gerenciadores de Contenção, como o uso de mecanismos de prioridade ou de *delays* [Harris et al. 2010]. Vale também apontar que uma métrica de desempenho importante para GCs é seu *makespan*, isto é, o tempo necessário para concluir um conjunto finito de transações quando utilizando tal GC [Zhang et al. 2014]. Sendo assim, um dos objetivos de um GC é minimizar o *makespan* para maximizar o *throughput* [Zhang and Ravindran 2009], isto é, aumentar o número de transações concluídas num determinado período de tempo.

No TRMI, todo Gerenciador de Contenção deve estender da classe `ContentionManager`. Contudo, os GCs eram fixos no sistema, sendo escolhidos de forma *hardcoded*. Para contornar isso, o TRMI foi modificado para poder receber, em tempo de carregamento do sistema, um parâmetro que indique qual GC deve ser instanciado pelas transações.

Os GCs implementados para este trabalho foram:

Passive - Aborta a transação atacante, isto é, a que chama o GC para obter posse do objeto, realizando um auto *abort*. Para manter a garantia *obstruction-free* do algoritmo de TM, uma transação não pode abortar a si própria sempre, em algum momento ela deve passar a abortar a inimiga de forma a garantir o seu progresso[Scherer III and Scott 2005]. Para isso, foi adicionado um novo campo, chamado transactionAborts, à classe `Transaction`. Este campo guarda o número de vezes que uma determinada transação auto abortou em tentativas anteriores de sua execução.

Polite - Aplica recuos exponenciais à transação atacante até chegar a um *delay* máximo, quando aborta a transação inimiga. Já presente no trabalho de [Ramos et al. 2023], foi adicionado a este GC a capacidade de definir seus *delays* mínimo e máximo dinamicamente, assim como para os demais GCs.

Karma - A cada abertura de um objeto transacional a prioridade da transação é incrementada, priorizando transações que realizaram ”mais trabalho”. Caso uma transação inimiga tenha posse daquele mesmo objeto sobre o qual a transação atacante queira escrever, o GC é chamado. Caso a transação atacante tenha prioridade maior que a inimiga, ou a soma da prioridade da transação atacante e suas tentativas de obter um objeto for maior que a prioridade da transação inimiga, esta é abortada. Caso contrário, é aplicada um *delay* fixo na transação atacante. Para implementar esse GC foi adicionado um campo de prioridade à classe `Transaction`. Este novo campo é incrementado a cada nova abertura de objeto transacional iniciada por uma transação e, caso seja esta abortada, mantém seu valor numa próxima tentativa de execução da transação, de forma a propiciar que todas as transações sejam eventualmente finalizadas.

Polka - Utiliza a mesma lógica básica do Karma com uma exceção: o *delay* aumenta exponencialmente (até um determinado limite) a cada nova tentativa de obter um objeto, assim como realizado no Polite. A nível de adições no sistema, o Polka se aproveitou do mesmo mecanismo de prioridade já implementado para o Karma.

Timestamp - A cada nova transação iniciada é atribuído um número inteiro, indicando sua ordem de lançamento na rede, de acordo com um relógio global/centralizado,

que mantém um contador atômico. Na ocorrência de um conflito, verifica-se se o *timestamp* da transação corrente/atacante é menor que o da transação inimiga, e neste caso a inimiga é abortada. Caso contrário, se o número de tentativas de executar a transação atacante for maior que o número de vezes em que o *delay* foi aplicado e a transação inimiga estiver inativa (i.e., não abriu objetos transacionais recentemente), a inimiga também é abortada. Para garantir que uma transação já abortada mantenha o mesmo *timestamp* inicial, seu valor anterior é sempre guardado ao ocorrer um *abort*. Quanto ao uso de um relógio global, cabe salientar que ele é um ponto de central de falha, contudo, GCs baseados em *timestamp* já foram usado noutros trabalhos [Saad Ibrahim 2011, Saad and Ravindran 2011b, Zhang and Ravindran 2009].

Kindergarten - Cada transação mantém uma lista de transações inimigas com as quais conflitou. Em caso de conflito, se a transação inimiga não está na lista, é aplicado um *delay* de período fixo à transação atacante. Caso contrário, isto é, caso a transação inimiga já esteja na lista, esta é abortada. Para implementá-lo, a lista de transações inimigas usa um *HashSet*, para prover busca média constante.

Less - Além das políticas citadas, uma nova também foi adicionada, tendo sido denominada de Less, onde aborta-se a transação com mais *aborts*, com o objetivo de priorizar transações que já tenham realizado mais progresso. Contudo, para prevenir que uma transação seja abortada indefinidamente, há um contador de auto *aborts* que é zerado para a transação que ultrapassar um número máximo. Esta nova política pretende prevenir situações onde transações com múltiplas falhas de execução impeçam transações com certo progresso de avançarem e, por consequência, atrasem a progressão de aplicações.

Aggressive - Este apresenta o comportamento oposto ao do Passive original, isto é, sempre aborta a transação inimiga. Sua escolha, inclusive, foi feita para contrastar o seu comportamento com o do *Passive*. Sua simplicidade, porém, torna-o suscetível a *livelocks*, como já relatado na literatura [Guerraoui et al. 2005], e esse impacto limitou seu uso nos testes realizados, como será detalhado na próxima Seção.

Como já mencionado na listagem acima, além de modificar o sistema TRMI para que o mesmo pudesse suportar diferentes GCs, as classes `Transaction` (que representa uma transação) e `TMObjServer` (que representa objetos remotos), foram modificadas para que pudessem guardar os meta-dados necessários para certos GCs. Também foi adicionada a possibilidade de configurar dinamicamente o GC e sua parametrização por meio de linha de comando.

4. Experimentos

Os experimentos foram executados em um servidor Linux Ubuntu 24.04.2 LTS que conta com 4 CPUs AMD Opteron 6276s com 8 cores cada, totalizando 32 cores físicos, e 128GB de memória RAM. Os testes foram realizados usando duas aplicações, um Simulador Sintético (Seção 4.1) e uma Tabela Hash Distribuída (DHT) (Seção 4.2). Para simular um ambiente distribuído, cada cliente e servidor roda em uma JVM (Java Virtual Machine) distinta associada a um dentre os 32 cores físicos da máquina. Em todos os testes realizados, tanto sobre o Simulador Sintético, quanto sobre a DHT, foram fixados 16 servidores onde os objetos transacionais são criados, e foi variado o número de clientes entre 2, 4, 8 e 16 clientes. Além disso, foram também variados o número de objetos transacionais registrados nos servidores, o número de objetos acessados por cada transação,

e a porcentagem de escritas/leituras realizadas. Em cada experimento é apresentado o tempo de execução de 5000 transações em cada configuração, i.e., *makespan*. Os tempos são a média de 10 execuções.

4.1. Simulador Sintético

A primeira aplicação utilizada nos experimentos foi um Simulador Sintético, apresentado em [Ramos et al. 2023], que simula diferentes cenários execução, permitindo configurar aspectos como, número de clientes/servidores, número de objetos em cada servidor, número de objetos acessados em cada transação, tempo de processamento de cada transação e porcentagem de escritas e leituras. Para os testes sobre o simulador sintético, o número de objetos transacionais disponíveis na rede foi variado entre 100 e 500 objetos, para simular cenários de maior e menor contenção, respectivamente. Também, foi variado a duração das transações entre longas, abrindo 20 objetos transacionais cada, e curtas, trabalhando sobre apenas 5 objetos. Ademais, a porcentagem de transações totais de escrita em uma execução foi variada entre 20% e 50%. Dessa forma, foi possível gerar múltiplos cenários de teste de acordo com a mudança dos valores dessas variáveis de controle. Para calibrar cada um dos Gerenciadores de Contenção implementados, foram realizados testes iniciais com diferentes parâmetros no simulador sintético sobre múltiplos cenários, variando diferentes aspectos de execução das transações, conforme descrito anteriormente. A partir dos resultados de desempenho obtidos, em *makespan*, foram definidos os parâmetros a serem utilizados nos GCs nas simulações tanto sobre o simulador sintético, quanto sobre a DHT.

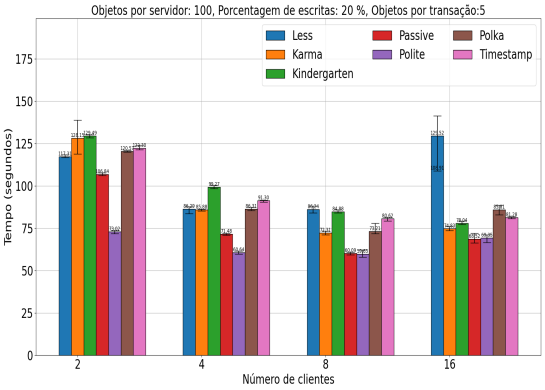
Os resultados dos testes para o Simulador Sintético podem ser vistos na Figura 1.

4.2. Distributed Hash-Table

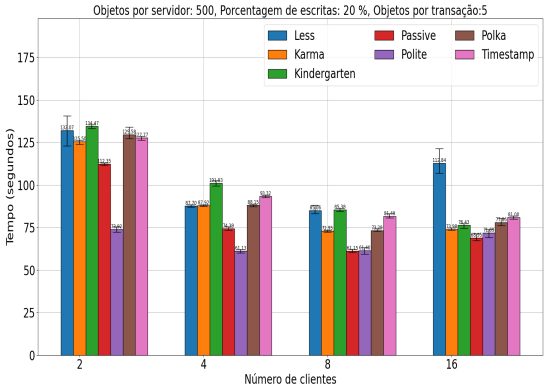
Para este trabalho, também foi concebido um novo *benchmark*, que implementa uma Tabela Hash Distribuída (Distributed Hash Table, ou DHT) [Balakrishnan et al. 2003] simplificada. A Tabela Hash Distribuída é composta por um conjunto de Tabelas Hash (HTs), cada uma localizada em uma máquina diferente do sistema. As Tabelas Hash individuais são representadas por *arrays* de *buckets*, onde cada *bucket* é implementado por uma lista encadeada de objetos transacionais, e o espaço de chaves é distribuído igualmente entre os servidores. Assim, as chaves para operações de inserção ou leitura de pares chave-valor nas Tabelas Hash são geradas pseudo-randomicamente, dentro do escopo delimitado para a máquina/ servidor onde o par chave-valor será inserido e/ou buscado. Para os testes com a DHT foram usados os seguintes parâmetros: a porcentagem de escritas foi variada entre 20% e 50%, o número de operações numa Tabela Hash realizadas por uma transação foi variada entre 20 e 50 (curta e longa), e variou-se também o número de *buckets* em cada servidor entre 128 e 1567 (alta e baixa contenção). Já o espaço de chaves foi fixado em 20000 chaves, variando entre 0 e 19999, geradas aleatoriamente para cada transação. Os resultados dos testes sobre a DHT podem ser vistos na Figura 2.

4.3. Discussão sobre os resultados obtidos

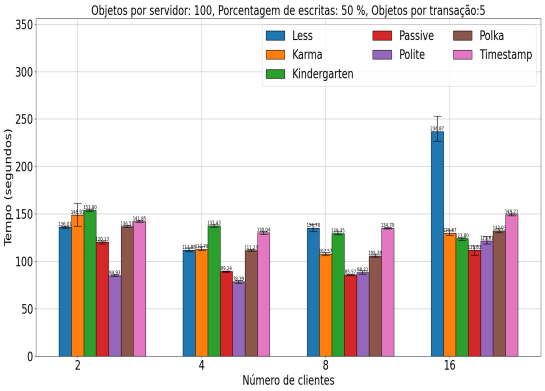
Como pode ser percebido nos gráficos do simulador sintético, na Figura 1, o Polite apresenta os menores *makespan* em todos os cenários com até 4 clientes, especialmente naqueles com transações curtas. Com 8 clientes, o Polite disputa com Passive como o melhor. Para 16 clientes, o Passive tem os menores tempos médios, enquanto o Polite varia



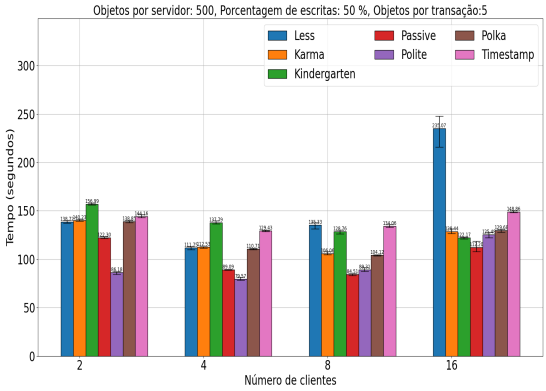
(a) 100 objetos por servidor, transações curtas com 20% de escritas



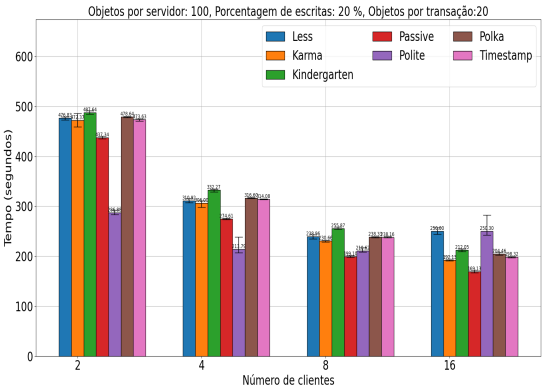
(b) 500 objetos por servidor, transações curtas com 20% de escritas



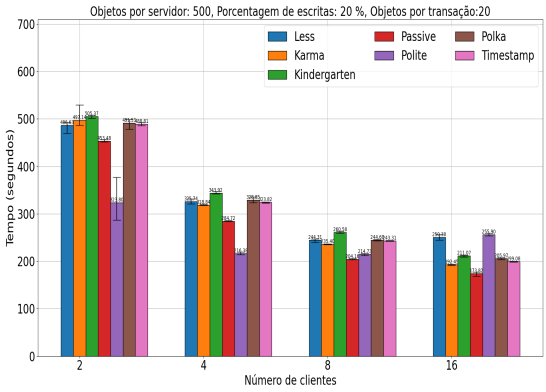
(c) 100 objetos por servidor, transações curtas com 50% de escritas



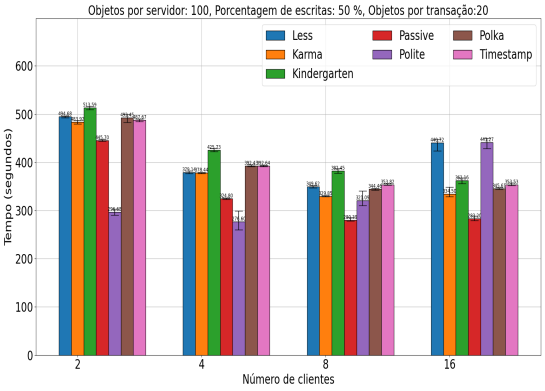
(d) 500 objetos por servidor, transações curtas com 50% de escritas



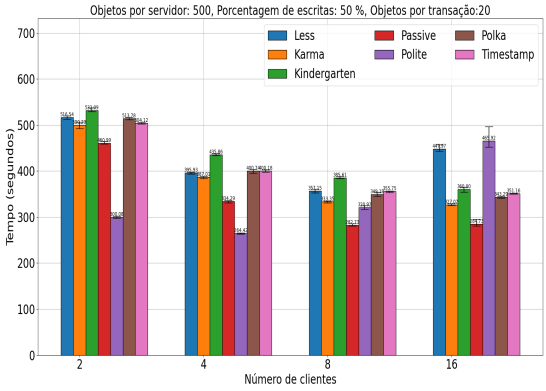
(e) 100 objetos por servidor, transações longas com 20% de escritas



(f) 500 objetos por servidor, transações longas com 20% de escritas

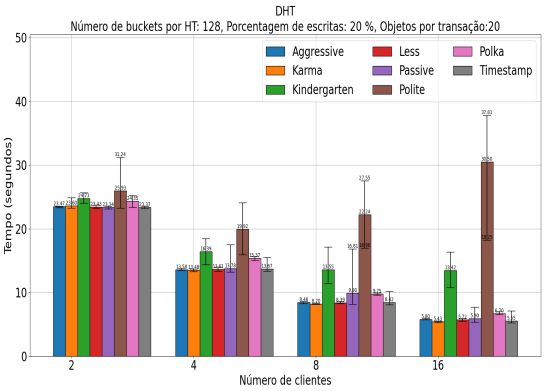


(g) 100 objetos por servidor, transações longas com 50% de escritas

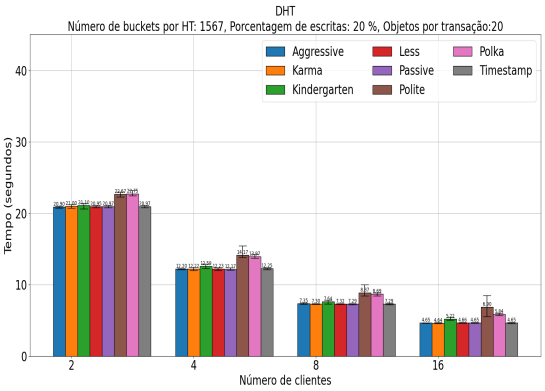


(h) 500 objetos por servidor, transações longas com 50% de escritas

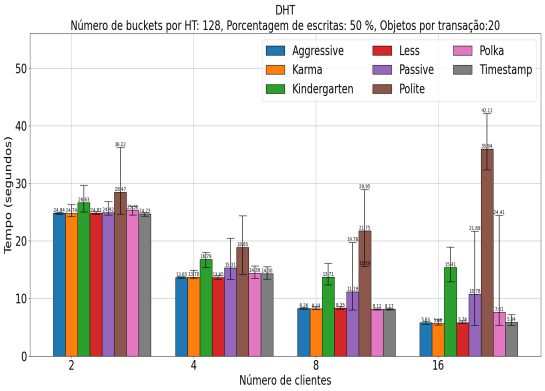
Figura 1. Resultados dos testes finais sobre o simulador sintético



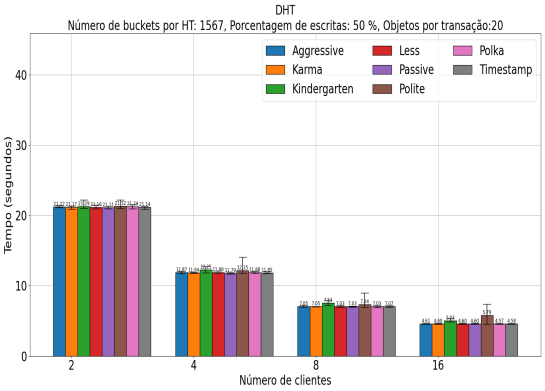
(a) 128 buckets por servidor, transações curtas com 20% de escritas



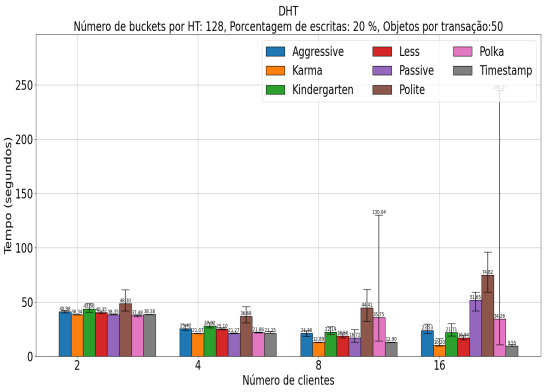
(b) 1567 buckets por servidor, transações curtas com 20% de escritas



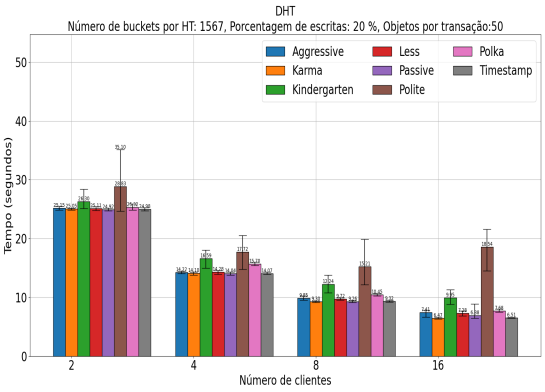
(c) 128 buckets por servidor, transações curtas com 50% de escritas



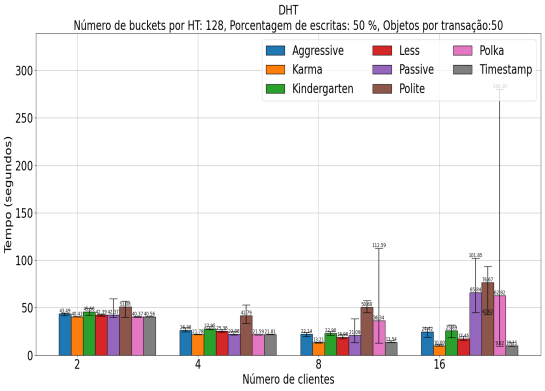
(d) 1567 buckets por servidor, transações curtas com 50% de escritas



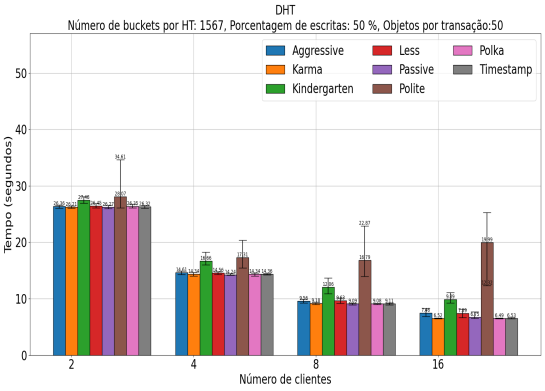
(e) 128 buckets por servidor, transações longas com 20% de escritas



(f) 1567 buckets por servidor, transações longas com 20% de escritas



(g) 128 buckets por servidor, transações longas com 50% de escritas



(h) 1567 buckets por servidor, transações longas com 50% de escritas

Figura 2. Resultados dos testes finais sobre a DHT

entre ser o segundo melhor GC, em cenários com transações curtas, e o pior GC, com transações longas. Nestes cenários, disputa o pior/maior *makespan* com o Less, o pior GC no quesito escalabilidade dentre os GCs testados nesta aplicação, apresentando tempos muito superiores aos dos demais GCs quando em transações curtas com 16 clientes. Quanto aos demais GCs, o Karma costuma apresentar o terceiro menor tempo médio quando usando 8+ clientes, exceto quando metade das transações curtas são de escrita. Neste caso, disputa essa posição com o Kindergarten, cujo desempenho demonstra a melhor escalabilidade dentre os GCs, apresentando seus menores *makespan* sempre com 16 clientes. Já o Polka costuma acompanhar o desempenho do Karma, mas com um certo tempo despendido a mais, especialmente com 16 clientes. Isso pode ser resultado da aplicação de *delays* maiores que o necessário eventualmente. Quanto ao Timestamp, este tende a ter um desempenho pior que o do Polka em cenários com transações curtas, mas muito parecido quando em transações longas. O GC Agressive, que aborta automaticamente a transação inimiga, apresentou *livelock* em vários cenários, conforme observado na literatura [Guerraoui et al. 2005], e por isso seus dados não foram apresentados. Cabe salientar que, apesar de ter apresentado os melhores resultados gerais para o simulador sintético, o Passive necessitou de diversos ajustes no seu parâmetro que conta o número máximo de auto *aborts*, para que não resultasse em *livelocks* devido à eventual aplicação da política Aggressive, quando passa automaticamente a cancelar as transações inimigas.

Na DHT, os tempos de execução médios, isto é, o *makespan* médio para a execução de 5000 transações nos cenários propostos, demonstram a prevalência de uma tendência de diminuição à medida em que o número de clientes aumenta em cenários de baixa contenção, com exceção do Polite. Já em cenários de alta contenção, é notável uma execução média mais demorada em outros GCs, como Kindergarten, Polka e Passive, onde há uma tendência de piora no *makespan* à medida em que mais clientes são usados; ou seja, há uma limitação de escalabilidade. Os picos de execução (*outliers*) atingidos durante algumas execuções do Polka em cenários com alta contenção, 8+ clientes, transação longas e/ou muitas escritas, são especialmente notáveis. Em todos os cenários, o Polite demonstra os piores resultados médios, sendo superado apenas por eventuais *outliers* em execuções usando o Polka ou o Passive. Esses resultados demonstram a limitação do Polite ao necessitar recuar um número fixo de vezes até abortar a transação inimiga, e ao usar uma calibração de parâmetros da simulação sintética. Os GCs Karma e Timestamp, por outro lado, apresentam os melhores resultados em todos os cenários, já Agressive e Less acompanham essa tendência no geral, mas tem piora em cenários com transações longas, mais escritas e mais clientes, apresentando uma limitação ao escalar para uma execução mais distribuída.

Dessa forma, os resultados apresentados na Figura 2 demonstram que, mesmo em aplicações mais simples e com tempo de execução média menor, há uma importância na escolha do meio de gerenciamento de conflitos em DTMs. Comparando os tempos médios de execução obtidos nas duas aplicações (Figuras 1 e 2) para múltiplos GCs, pode ser percebida a diferença que as características da aplicação trazem na escolha do GC mais adequado. Alguns dos GCs com os menores *makespan* no Simulador Sintético, como o Passive e o Polite (em cenários com menos clientes), figuraram entre os maiores *makespan* na DHT. O inverso ocorreu com o Less. Além disso, o uso do GC Aggressive foi possível na DHT, onde apresentou resultados bons consistentemente, demonstrando sua capacidade de uso em aplicações com transações de pouca duração.

5. Trabalhos Relacionados

A literatura traz diferentes formas de implementação de DTMs. Em [Saad Ibrahim 2011], por exemplo, é apresentado o HyFlow, um framework para uso de DTMs escrito em Java. O Hyflow suporta tanto o modelo *data-flow* , quanto o modelo *control-flow* sendo ambos modelos baseados em *locks*. Em [Saad Ibrahim 2011] é proposto um GC Global para o HyFlow para definir a progressão de transações em cenários em que algumas delas seriam *data-flow* e outras *control-flow*. Contudo, a proposta não é concretizada nos trabalhos seguintes sobre o sistema [Saad and Ravindran 2011a, Saad and Ravindran 2011b, Saad and Ravindran 2012]. No sistema Snake [Saad and Ravindran 2011b], que é a DTM *control-flow* do HyFlow, são apresentados alguns resutados de experimentos sobre os GCs clássicos Timestamp, Polite e Priority. No Snake, cada nodo sobre o qual a transação executa possui um GC local. Para realizar o *commit*, todos os nodos devem concordar, votando por meio de seus GCs locais, em efetivar aquela transação. Todos os GCs implementados neste trabalho utilizam apenas informações das transações envolvidas no conflito para tomar uma decisão, sem a necessidade de votação.

Nos primeiros trabalhos sobre DTM, como [Kotselidis et al. 2008b] e [Kotselidis et al. 2008a], foi experimentada a união de TMs *multicore* com uma API para o RMI, para executar transações em um *cluster*. Nestes trabalhos, um nodo mestre mantém o conjunto de objetos atualizados, e as transações executam localmente nos demais nodos sobre cópias desses objetos. A efetivação dos resultados de uma transação no *cluster* depende de validação remota, onde transações que iniciaram sua validação antes são priorizadas, de forma parecida com o Timestamp.

O trabalho teórico de [Zhang and Ravindran 2009] apresenta provas formais da melhora de desempenho em DTMs *data-flow* ao usar o GC Greedy junto de protocolos de coerência de cache *location-aware* (LAC) eficientes, ao invés de protocolos de coerência de cache arbitrários. Seguindo o trabalho anterior, em [Zhang et al. 2014], é apresentado um novo GC para DTMs *data-flow* denominado Cutting. Este trabalho também apresenta apenas provas formais, relacionadas à melhoria de desempenho dada pelo Cutting em cenários onde os objetos a serem acessados pela transação são conhecidos no início de sua execução.

Outros trabalhos, como [Busch et al. 2018, Poudel et al. 2021, Busch et al. 2022, Poudel et al. 2024], focam na otimização de algoritmos de escalonamento, isto é, numa abordagem pessimista, com foco em evitar conflitos de dados, ao invés de tratá-los quando ocorrem. Tais trabalhos tentam otimizar dois aspectos em escalonadores para DTMs: o tempo de execução de todas as transações, e o custo de comunicação necessário para mover (ou buscar) os objetos entre os nodos da rede.

Por fim, em [Ramos et al. 2023] é apresentado o TRMI, isto é, o modelo de DTM utilizado no presente artigo. No trabalho, são realizados testes sob diferentes cenários com o Simulador Sintético, executando-o o TRMI com 2 GCs distintos, Passive e Polite. Como resultado, o modelo de DTM TRMI demonstrou bons resultados em vários cenários quando comparado ao uso de *locks*.

6. Conclusões e Trabalhos Futuros

Este trabalho apresentou a adaptação e integração de diversos Gerenciadores de Contenção clássicos para Memórias Transacionais em um sistema de Memória Transa-

cional Distribuída. Com o objetivo de entender o comportamento desses gerenciadores em diferentes cenários, os mesmos foram executados com 2 aplicações: um Simulador Sintético, que simula diferentes contenções, e uma Tabela Hash Distribuída, permitindo a contrastação dos resultados, e a demonstração da diferença de desempenho entre os GCs a depender do cenário e da aplicação em que são usados.

Existem várias ideias para dar continuidade ao trabalho apresentado neste artigo. Como nem sempre é viável encontrar os melhores parâmetros para os GCs em cada aplicação, trabalhos futuros poderiam explorar a configuração dinâmica desses parâmetros. Isso poderia impedir a eventual ocorrência de *livelocks* ao "dar turnos" de tempo insuficiente para transações concluírem em GCs baseados em *delay* fixo, como Kindergarten e Karma. Já no caso de GCs como Polka e Polite, foi demonstrado o impacto no uso de um *delay* desajustado na DHT, ou de manter-se o mesmo em cenários com muitos clientes na própria simulação sintética. Sendo assim, o ideal seria seu ajuste dinâmico, ajustando-se às necessidades de cada situação de execução. No caso do Passive, uma alternativa seria a volta do GC de uma transação ao estado passivo, caso seu número máximo de auto *aborts* tivesse sido ultrapassado e ela continuasse abortando. Esta abordagem poderia impedir a eventual adoção do comportamento agressivo simultaneamente por várias transações, assim prevenindo a ocorrência de *livelocks* no Passive. Também, como este trabalho possibilitou o carregamento dinâmico de diferentes GCs, trabalhos futuros poderiam explorar a troca dos GCs (ou de seus parâmetros) durante a execução da aplicação, de acordo com alguma heurística pré-estabelecida baseada na análise do cenário presente. Além disso, trabalhos futuros poderiam experimentar a execução do TRMI em clusters. Nestes, outros aspectos como custo de comunicação trariam maior impacto na execução das aplicações, especialmente ao executá-las usando diferentes Gerenciadores de Contenção.

Referências

Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2003). Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48.

Busch, C., Herlihy, M., Popovic, M., and Sharma, G. (2018). Time-communication impossibility results for distributed transactional memory. *Distributed Computing*, 31:471–487.

Busch, C., Herlihy, M., Popovic, M., and Sharma, G. (2022). Dynamic scheduling in distributed transactional memory. *Distributed Computing*, 35(1):19–36.

Guerraoui, R., Herlihy, M., and Pochon, B. (2005). Toward a theory of transactional contention managers. In *24th ACM PODC*, pages 258–264.

Guerraoui, R. and Romano, P. (2015). *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Lecture Notes in Computer Science. Springer.

Harris, T., Larus, J. R., and Rajwar, R. (2010). *Transactional memory*. Morgan & Claypool San Francisco.

Herlihy, M. (2012). The art of multiprocessor programming.

Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., and Watson, I. (2008a). Distm: A software transactional memory framework for clusters. In *2008 37th International Conference on Parallel Processing*, pages 51–58. IEEE.

Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., and Watson, I. (2008b). Investigating software transactional memory on clusters. In *IPDPS 2008*, pages 1–6. IEEE.

Poudel, P., Rai, S., and Guragain, S. (2024). Ordered scheduling in control-flow distributed transactional memory. *Theoretical Computer Science*, 993:114463.

Poudel, P., Rai, S., and Sharma, G. (2021). Processing distributed transactions in a predefined order. In *22nd ICDCN*, pages 215–224.

Ramos, J., Du Bois, A. R., and Cavalheiro, G. (2023). Obstruction-free distributed transactional memory. In *27th SBLP*, pages 33–40.

Saad, M. M. and Ravindran, B. (2011a). Hyflow: A high performance distributed software transactional memory framework. In *20th HPDC*, pages 265–266.

Saad, M. M. and Ravindran, B. (2011b). Snake: control flow distributed software transactional memory. In *Symposium on Self-Stabilizing Systems*, pages 238–252. Springer.

Saad, M. M. and Ravindran, B. (2012). Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems. In *2012 IEEE 24th SBAC-PAD*, pages 219–226. IEEE.

Saad Ibrahim, M. M. (2011). *HyFlow: A High Performance Distributed Software Transactional Memory Framework*. PhD thesis, Virginia Tech.

Scherer III, W. N. and Scott, M. L. (2005). Advanced contention management for dynamic software transactional memory. In *24TH ACM PODC*, pages 240–248.

Shen, Q., Sharp, C., Davison, R., Ushaw, G., Ranjan, R., Zomaya, A. Y., and Morgan, G. (2020). A general purpose contention manager for software transactions on the gpu. *Journal of Parallel and Distributed Computing*, 139:1–17.

Siek, K. and Wojciechowski, P. T. (2016). Atomic RMI: A distributed transactional memory framework. *International Journal of Parallel Programming*, 44(3):598–619.

Zhang, B. and Ravindran, B. (2009). Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *2009 28th IEEE SRDS*, pages 268–277. IEEE.

Zhang, B., Ravindran, B., and Palmieri, R. (2014). Distributed transactional contention management as the traveling salesman problem. In *International Colloquium on Structural Information and Communication Complexity*, pages 54–67. Springer.