Proposta de implementação de uma interface de alto nível para Memórias Transacionais Distribuídas usando RMI em Java

Letícia Pegoraro Garcez[1], **André R. Du Bois**[1]

[1]Laboratory of Ubiquitous and Parallel Systems (LUPS)
Universidade Federal de Pelotas (UFPel)
Pelotas - RS - Brasil

{lpgarcez,dubois}@inf.ufpel.com.br

Resumo. *O Laboratory of Ubiquitous and Parallel Systems da UFPel, está desenvolvendo um projeto de Memória Transacional Distribuída (DMT) usando RMI que precisa utilizar diversos trechos de código similares e repetitivos relacionados às transações e à arquitetura distribuída. O presente trabalho propõe a implementação de uma interface de alto nível para esse sistema, de modo a abstrair os trechos de código citados acima, melhorando a interface.*

1. Introdução

O surgimento de computadores com múltiplos cores, marcou o início de uma nova era computacional, possibilitando um grande aumento tanto na velocidade quanto na capacidade de processamento, por meio da divisão de tarefas de uma aplicação entre os cores do processador. No entanto, é necessária a otimização do código a ser executado para este tipo de processamento, ou não há ganho de desempenho, já que as tarefas de aplicações sequenciais não podem ser divididas entre os cores.

A paralelização de um algoritmo se dá principalmente pelo uso de threads bem sincronizadas. Porém uma sincronização mal feita, além de reduzir o desempenho, pode causar diversos problemas como starvation e deadlocks, além é claro da possibilidade de gerar dados inconsistentes. A sincronização, no entanto, se torna mais difícil quando se aumenta a quantidade de threads, locks e recursos compartilhados, tornando o processo complicado e desgastante para o programador.

Neste contexto, surgem as Memórias Transacionais, uma abstração que transforma uma seção crítica de código em uma transação, similar as transações usadas em bancos de dados, retirando do programador a responsabilidade referente a sincronização das threads. Outro importante paradigma de programação distribuída é o Remote Method Invocation, ou RMI, uma abstração de comunicação entre diversas máquinas ligadas em rede, que permite que uma determinada máquina acesse objetos remotos disponíveis em outras máquinas da rede através de uma interface de comunicação.

No Laboratory of Ubiquitous and Parallel Systems da Universidade Federal de Pelotas (LUPS/UFPel), está sendo desenvolvido um projeto que une os conceitos de RMI e Memórias Transacionais [da Cunha Ramos et al. 2016, da Cunha Ramos 2018], fornecendo ao programador uma Memória Transacional Distribuida, do inglês Distributed Transacional Memory (ou DTM) onde as definições de memórias transacionais são aplicadas a arquiteturas distribuídas. Em seu estado atual, o projeto de DTM do LUPS

requer a utilização de diversos trechos de código repetitivos, que poderiam ser abstraídos, buscando retirar do programador a responsabilidade sobre alguns trechos de código semelhantes e recorrentes, requeridos por este tipo de sistema.

O objetivo do trabalho em mãos é propôr a implementação de uma interface de alto nível para Memórias Transacionais Distribuidas usando RMI em Java. Para alcançar este objetivo, será utilizada a ferramenta ANTLR4, apresentada na seção 3, para fazer alterações na gramática do Java e tornar possível a utilização das abstrações propostas neste trabalho.

A segunda seção deste trabalho aborda rapidamente alguns conceitos e funcionalidades das Memórias Transacionais, já na terceira seção, a ferramenta ANTLR4 será apresentada com mais clareza. O objetivo do presente trabalho é melhor abordado na quarta seção, e finalmente, na quinta seção, se encontram as conclusões e os passos futuros para a concretização do objetivo apresentado apresentado na quarta seção.

2. Memórias Transacionais

Memórias Transacionais, como já foi exposto anteriormente, é uma abstração que permite transformar seções críticas de código em transações, definidas como uma sequência finita de instruções, executadas por um único processo, satisfazendo as propriedades de atomicidade e isolamento [Herlihy and Moss 1993]. De maneira geral, estas duas propriedades garantem que uma transação será executada serialmente, sem a interferência de nenhuma outra transação, e que ao final de sua execução, as alterações feitas na memória serão de fato aplicadas (commit) ou descartadas (abort).

Uma transação deve ser capaz de garantir a consistência dos dados manipulados por ela. Para isso o modelo de implementação da Memória Transacional deve implementar sistemas de versionamento de dados, detecção e resolução de conflitos, para assegurar as propriedades anteriormente citadas, caso contrário, mudanças que não deveriam acontecer são aplicadas, o que gera uma inconsistência nos dados já existentes, prejudicando assim, o funcionamento do algoritmo desenvolvido.

Os tipos de implementação de memórias transacionais [de Leão Bandeira 2013] são as memórias transacionais em software (STM), hardware (HTM) e híbridas (HyTM), nomeadas a partir do componente que implementa o sistema transacional. O foco deste trabalho, se dá sobre Memórias Transacionais Distribuídas implementadas em software (ou DSTM), ou seja, a aplicação do conceito de Memória Transacional determinado anteriormente, ao processamento distribuído, facilitando a sincronização entre tarefas executadas por objetos distribuídos [Siek and Wojciechowski 2016].

Várias características de Memórias Transacionais podem variar de acordo com decisões de implementação, como tipo de detecção de conflitos, versionamento de dados e escalonamento de transações. Em DSTMs, essas variações de implementação são acrescidas ainda de decisões de projeto relacionadas a mobilidade dos objetos e transações, verificação de conflitos distribuídos, replicação, entre outros, e é possível citar diversas variantes de DSMT como por exemplo: Cluster-STM [Bocchino et al. 2008], D2STM [Couceiro et al. 2009], Cloud-TM[Romano et al. 2010], entre outras. É importante, portanto compreender que o conceito de DSTM não é limitado a um estilo de implementação, e sim a um conceito mais abstrato.

3. ANTLR4

ANTLR, acrônimo para Another Tool for Language Recognition [Parr 2012] é um poderoso gerador de parsers que pode ser usado para ler, processar, executar ou traduzir texto estruturado ou arquivos binários. Seu funcionamento se baseia na geração e percorrimento de uma *Parse Tree*, também conhecida como árvore sintática, gerada a partir da definição formal da gramática de uma determinada linguagem.

Após a definição da gramática e compilação da mesma, arquivos auxiliares são gerados tanto para a criação da *Parse Tree* quanto para possibilitar a implementação de ferramentas e aplicações como tradutores e interpretadores. Nestas implementações, é necessário percorrer a árvore sintática usando as *Tree Walkers* providas pela ferramenta, que a partir da visitação dos nós da árvore, realizam ações para gerar a aplicação necessária.

Existem duas possibilidades de *Tree Walkers*, o *Listener* e o *Visitor*. O código para a implementação de ambos é bastante similar e a escolha pelo uso de um ou outro, recai em características funcionais. Além disso, o ANTLR conta com diversas classes para aumentar as funcionalidades dos *Tree Walkers*. No presente trabalho, a ferramenta será usada traduzir alguns trechos de código de nível mais alto para um nível mais baixo, como será explicado na seção seguinte.

4. Objetivo do Trabalho

Como já foi referido em seções anteriores, a implementação de DTMs proposta inicialmente pelo projeto do LUPS da UFPel, exige do programador a inclusão de diversos trechos de código similares e recorrentes. O código especificado na seção esquerda da Figura 1, mostra o código necessário para a implementação de uma transação distribuída. Um código nos mesmos moldes deve ser replicado para cada transação proposta, o que torna um algoritmo com diversas transações confuso e com muito espaço para erros.

Já na seção direita da mesma Figura, se encontra a abstração proposta por este trabalho, onde todos os trechos repetitivos relativos às transações serão abstraídos, cabendo ao programador a especificação apenas da transação, e não da implementação da mesma.

Para realizar essa implementação, pretende-se utilizar a ferramenta ANTLR4 já descrita para reconhecer o código da seção direita da Figura 1, e a partir da tradução do mesmo, gerar todos os trechos relativos a implementação transacional demonstrados na seção esquerda da mesma Figura, aumentando assim o nível de abstração da interface e deixando o processo de implementação das transações transparente para o programador.

```
Account account1 = (...);
Account account2 = (...);
Trans t = new Trans();
t.start();
  while(t.state != COMMITED){

  account1.withdraw(t, 20);
  account2.deposit(t, 20);

  switch(t.state){
  case RETRY:
        t.retry();
        break;
        case ACTIVE:
        t.commit();
        break;
        case ABORTED:
        t.rollback();
        break;
        default:
        break;
     }
  }
```

```
Account account1 = (...);//Referência
//remota
Account account2 = (...);
        atomic{
           account1.withdraw(20);
           account2.deposit(20);
        }
```

Figura 1. Comparação entre a implementação atual (esquerda) e a interface proposta (direita)

5. Conclusões e Passos Futuros

Até o presente momento, buscou-se, além de realizar pesquisas bibliográficas sobre os conceitos de Memórias Transacionais, Remote Mehod Invocation e Distributed Transacional Memories, compreender o funcionamento da ferramenta ANTLR4 em busca da melhor estratégia de implementação da interface proposta, bem como começar sua implementação.

O seguimento do trabalho se dará pela continuação da fase de implementação e testes do mecanismo de tradução da interface de alto nível, e aperfeiçoamento do mesmo de modo a tornar a interface proposta mais concisa e apta para utilização.

Referências

Bocchino, R. L., Adve, V. S., and Chamberlain, B. L. (2008). Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 247–258, New York, NY, USA. Association for Computing Machinery.

Couceiro, M., Romano, P., Carvalho, N., and Rodrigues, L. (2009). D2stm: Dependable distributed software transactional memory. pages 307–313.

da Cunha Ramos, J. (2018). Proposta de um sistema transacional distribuido, control flow, opaco e com versionamento baseado em relógio lógico. Proposta de Tese - PPGC / UFPEL.

da Cunha Ramos, J., Bois, A. R. D., and Pilla, M. L. (2016). Um estudo sobre stm para arquiteturas distribuídas. *Escola Regional de Alto Desempenho*, pages 205–206.

de Leão Bandeira, R. (2013). Um sistema de detecção de conflitos com invalidação mista para a linguagem ctm java. Dissertação de Mestrado — PPGC/UFPEL.

Herlihy, M. and Moss, J. E. B. (1993). Transacional memory: Architectural support for lock-free data structures. *ISCA '93 Proceedings of the 20th annual international symposium on computer architecture*, 21(2):289–300.

Parr, T. (2012). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 1st edition.

Romano, P., Rodrigues, L., Carvalho, N., and Cachopo, J. (2010). Cloud-tm: Harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, page 1–6.

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