Compartilhamento de Recursos em Sistemas Distribuídos para Sistemas de Controle de Versão

Mario João Jr.[1], **Matheus S. D. Alves**[1], **João Silva**[1], **Meirylene Avelino**[1] **e Lúcia Drummond**[1]

[1] Instituto de Computação, UFF – Niterói – RJ – Brasil

{mariojoao,mad,joaomfs,meirylenerea}@id.uff.br, lucia@ic.uff.br

Abstract. *Version control systems, such as Git, have revolutionized the way teams are programmed. Version control allows developer teams to share their source code and make it easier to organize. However, allowing each user to gain exclusive access to shared resources (source code, for example) is an open problem in these systems. The purpose of this work is to present some classic distributed algorithms of token resource sharing that can be applied, with some adaptation, to the solution of this problem.*

Resumo. *Sistemas de controle de versão, tal como Git, revolucionaram a forma de programação em equipes. O controle de versão permite que os times de desenvolvedores compartilhem seus códigos fonte, além de facilitar a organização dos mesmos. Porém, permitir que cada usuário consiga acesso exclusivo a recursos compartilhados (códigos fonte, por exemplo) é um problema em aberto nesses sistemas. O objetivo desse trabalho é apresentar alguns algoritmos distribuídos clássicos de compartilhamento de recursos baseados em* token *que podem ser aplicados, com alguma adaptação, para a solução deste problema.*

1. Introdução

O desenvolvimento de software para variados fins vem sendo aprimorado com o passar do tempo em quesitos que envolvem tempo de execução, lógica, design, entre outros. Um desses quesitos que sempre está em pauta é a eficiência do algoritmo e a agilidade em compartilhar o código que está sendo desenvolvido entre as equipes.

O *git* é uma ferramenta de controle de versões onde diversas pessoas podem trabalhar no código-fonte, individualmente. Ele mantém um histórico com todas as alterações realizadas, o que permite ter uma rastreabilidade muito grande do trabalho que está sendo desenvolvido. Uma vez que há o acesso de diversas pessoas a um único recurso (código-fonte), é possível inferir que, caso a organização e o gerenciamento desse recurso não seja feita de uma forma cuidadosa, erros podem acontecer. Um desses erros (*racing conditions*) é um problema fundamental da Ciência da Computação e vem sendo estudado desde meados do século XX, tendo como mais renomado estudo a criação dos semáforos por Edsger Wybe Dijkstra em 1965 [Dijkstra 1968].

Os sistemas de controle de versão existentes no mercado atualmente, não cuidam da exclusão mútua do recurso que está sendo compartilhado. Caso um usuário (user 1) realize alterações em um recurso compartilhado, não é garantida a exclusividade daquele acesso, podendo um outro usuário (user 2) trabalhar no mesmo recurso sem o conhecimento sobre quais alterações estão sendo realizadas pelo user 1 até que um *commit* seja realizado. Um algoritmo básico para resolver esse problema é o algoritmo Naimi-Trehel

[Naimi et al. 1996]. Tal algoritmo se baseia em utilizar um *token* como representação do recurso compartilhado, eleger um nó inicial para possuí-lo e uma estrutura em árvore para o gerenciamento do mesmo. Esse algoritmo não possui alguns detalhes necessários a um sistema de versionamento distribuído. Um deles é o isolamento de áreas do código de acordo com a especialização de cada membro dentro de uma equipe. Essa limitação é superada pelo algoritmo proposto por Bertier [Bertier et al. 2006] onde cada *cluster* corresponde aos membros de uma mesma especialidade de uma equipe. Porém, apesar de solucionar o acesso a um recurso, em sistemas de versionamento reais, geralmente, a utilização de mais de um recurso simultaneamente é necessário. Para solucionar esse problema é proposto o algoritmo de Bouabdallah [Bouabdallah and Laforest 2000] que se utiliza de um *control token* para controlar a posse de cada recurso separadamente. Nas próximas seções, esses algoritmos e suas implementações serão apresentados para tratar a exclusão mútua de recursos em sistemas distribuídos e, com alguma adaptação, podem ser aplicados em sistemas de controle de versão tal como o git.

2. Abordagens

No contexto de sistemas distribuídos, também faz-se necessária a exclusão mútua no acesso a recursos [Kakugawa 1995], uma vez que somente um processo pode acessar a seção crítica em um determinado instante. Os algoritmos para exclusão mútua existentes na literatura, podem ser divididos em duas classes: baseados em *token* e baseados em permissão [Raynal 1991]. Na primeira classe, uma entidade virtual (*token*) define quem poderá acessar a seção crítica e os nós solicitam entre si essa entidade para poder fazer o acesso. Na segunda abordagem, a cada necessidade de acesso, os nós promovem uma eleição e o nó escolhido terá o direito de acessar a seção crítica.

Neste trabalho, duas possíveis abordagens da classe dos algoritmos baseados em *token* foram utilizadas. A primeira, utilizando um tipo especial de *token* chamado *control token* é apresentada na subseção 2.1. A segunda, organizada de forma hierárquica, é apresentada na subseção 2.2.

2.1. Algoritmo baseado em Control Token e implementação

Em Bouabdallah [Bouabdallah and Laforest 2000] é apresentado um algoritmo de alocação de recurso baseado em *control token* que controla cada recurso separadamente, permitindo assim que vários processos possam entrar em suas seções críticas de forma simultânea. Para isso, são utilizados dois tipos de *tokens*: os *tokens* únicos, que permitem o acesso a cada um dos recursos separadamente, e o *control token*, que controla as solicitações de *tokens* e o acesso à seção crítica.

Nesse algoritmo, cada nó i possui três variáveis: *next* que indica o nó para o qual ele deve enviar o *control token*; *Parent* que indica o nó que possui (ou vai possuir) o *control token* e portanto, o nó para quem ele deve enviar a solicitação quando desejar entrar em uma seção crítica; e a variável *tokens_present* que indica os *tokens* que ele possui. O *control token* consiste de duas listas: a lista A que possui os *tokens* que estão livres e a lista B que indica quais nós possuem (ou vão possuir) cada *token*.

Ao receber o *control token*, um nó i pode se encontrar em dois casos. No primeiro caso, o nó já possui todos os *tokens* necessários para acessar a seção crítica que ele quer. Então, ele trava os *tokens* que ele vai utilizar, envia o *control token* para seu *next* e entra

na seção crítica. Caso contrário, ele não possui todos os *tokens* necessários para acessar a seção crítica. Logo, faz-se necessário enviar uma solicitação de *token*, por meio de uma mensagem *INQUIRE*, para os nós que possuem os *tokens* que ele precisa, utilizando a lista B. Após a solicitação dos *tokens*, o nó atualiza a lista e espera uma mensagem INQ_ACK1 desses nós.

Quando um nó s recebe uma mensagem *INQUIRE* de um nó i, ele responde com uma mensagem *INQ_ACK1* com os *tokens* solicitados que estão disponíveis. Caso existam *tokens* solicitados que não estejam disponíveis para serem enviados, esses *tokens* são armazenados na tupla *(i,Q)*. Quando o nó s sair da sua seção crítica e, assim, tiver os *tokens* de *(i,Q)* disponíveis para serem enviados, estes são enviados dentro de uma mensagem *INQ_ACK2*, satisfazendo completamente a mensagem de *INQUIRE* do nó i. Quando o nó i recebe o último *INQ_ACK1*, ele então trava os tokens recebidos e envia o *control token* para o *next*, se existir. Se agora ele possuir todos os *tokens* necessários, ele entra na seção crítica, caso contrário, ele espera o *INQ_ACK2* até receber todos os *tokens*.

Para a implementação do Algoritmo de Bouabdallah [Bouabdallah and Laforest 2000] foi utilizado o MPI para a paralelização, onde cada processo representa um nó. As mensagens descritas acima foram todas enviadas e recebidas por meio de mensagens bloqueantes (*MPI_Send* e *MPI_Recv*) e por isso, para cada processo foram criadas outras três *threads*, utilizando *Pthreads*, responsáveis por receber e interpretar as mensagens. Uma dessas *threads* é a responsável por receber a mensagem com o pedido de *control token*, por receber as mensagens de pedido e recebimento de *tokens* de recurso, e a última pelas mensagens de *finalize*. O *control token* é um vetor de *n_rec* posições, sendo *n_rec* o número de recursos (*tokens*), onde cada posição do vetor guarda o nó que possui o *token* naquele momento. Assumindo o valor -1 caso o *token* esteja livre. A lista Q dos nós é uma matriz de dimensão *n_proc* por *n_rec*, onde *n_proc* representa o número de processos.

2.2. Algoritmo hierárquico e implementação

Bertier, em [Bertier et al. 2006], apresenta uma versão hierárquica do algoritmo de Naimi–Trehel [Naimi et al. 1996]. Na abordagem apresentada nesse trabalho, o primeiro algoritmo hierárquico de Bertier foi utilizado, onde a requisição do *token* representa a solicitação de algum desenvolvedor para atualizar o repositório. Sua estrutura hierárquica modela a forma distribuída de desenvolvimento em times de desenvolvedores geograficamente distribuídos. A ideia principal do algoritmo, e que facilmente é associada ao modelo de desenvolvimento distribuído de software, é a separação dos nós (desenvolvedores) em *clusters* (times), onde existe uma priorização para requisições internas ao *cluster*, controladas por um *proxy* (líder do time).

Os nós são organizados em uma árvore, onde cada nó aponta para seu pai, indicando a direção que a solicitação do *token* deve seguir até a raiz (nó que detém o *token* no momento) — Figura 1a. Apenas o *proxy* de cada *cluster* aponta para nós de fora do *cluster*. Os nós internos apontam para o *proxy*, caso a raiz seja fora do *cluster*.

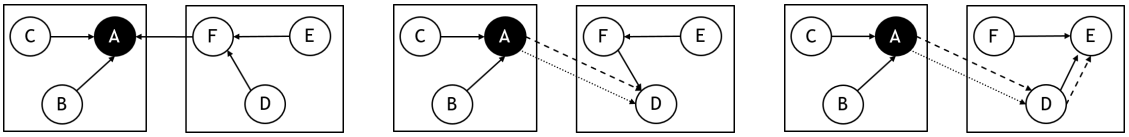
Cada nó N_i possui as seguintes variáveis: *self*, sua identificação; *owner*, representando o nó que possui o *token*; *next*, o nó que receberá o *token* quando N_i liberar a seção crítica; *token*, indicando se N_i possui o *token*; *requesting*, se N_i está requisitando o *token*; *LocalCluster*, um vetor conhecido por todos os nós, onde cada posição i indica a

que *cluster* N_i pertence; *nb_preempt* número sucessivo de requisições locais (usado para não causar *starvation*. O *proxy* de cada *cluster* ainda possui a variável *remote_owner* que indica o nó remoto que receberá o *token* quando esse for liberado pelos membros do *cluster*. Além dessas variáveis, uma constante *Elected_node* indica o nó que possui o *token* inicialmente.

Os nós se comunicam por meio de 3 mensagens: *Request* que tem como parâmetro a identificação do nó e é enviada por um nó que não detém o *token* e quer acessar a seção crítica; *Token* que representa a transmissão do *token*; e *Preempt* para enviar o valor de *nb_preempt* entre os nós do mesmo *cluster* para que eles saibam quando deixar outro *cluster* possui o *token* e, assim, evitar *starvation*.

Inicialmente, cada nó configura o seu *owner*. O nós que não pertencem ao *cluster* do *Elected_node* configuram seu *owner* para o *proxy* do *cluster*. A Figura 1a mostra a inicialização de dois *clusters* cada um com 3 nós. Os *proxies* são, respectivamente, os nós A e F. O nó A também é o *Elected_node* e neste momento está acessando a seção crítica. Na Figura 1b, o nó D, que não pertence ao *cluster* do *Elected_node* solicita o acesso à seção crítica por meio de uma mensagem de *Request* para F, o *proxy* do *cluster* de D. F então redireciona a mensagem de *Request* para A. F atualiza seu *owner* para D e A, ao receber a mensagem, atualiza seus *next* e *remote_owner* para D.

Na Figura 1c, o nó E solicita a F o acesso à seção crítica. F redireciona a mensagem para D, que atualiza seus *next* e *owner* para E. F também atuliza seu *owner* para E. Neste cenário, a requisição de E não é enviada para fora do seu *cluster*, sendo tratada internamente.



(a) Configuração Inicial **(b) D solicita o *token*** **(c) E solicita o *token***

Figura 1. Exemplos da Árvore do Algoritmo de Bertier

Para implementar o algoritmo proposto por Bertier e simular sua utilização para exclusão mútua distribuída, foi criada uma aplicação onde cada processo representa um nó e é dividido em duas partes. A primeira parte simula a utilização da seção crítica e a segunda parte é responsável pela troca de mensagens do algoritmo. Ambas as partes compõem o mesmo processo, tendo sido criada uma *thread* para cada uma, utilizando Pthreads. As *threads* são sincronizadas por meio de semáforos e variáveis condicionais nativos do Pthreads. A *thread* responsável pela comunicação entre processos utiliza MPI para implementar as mensagens do algoritmo (*Request*, *Token* e *Preempt*). Devido à natureza assíncrona dessas mensagens, todo o recebimento foi implementado utilizando MPI_Test e MPI_Irecv, o que faz com que essa *thread* fique em *loop de status* até o final da aplicação, tratando cada mensagem recebida de acordo com o algoritmo de Bertier.

3. Discussão e considerações finais

Ao observar-se um sistema de controle de versão distribuído, alguns requisitos podem ser especulados. Consistência, confiabilidade e replicação são alguns deles. Porém, o requisito que esse trabalho propõe resolver é bem mais específico e não ocorre em todos os

casos, o problema de exclusão de alteração simultânea a um mesmo objeto. Diferentemente de outros sistemas distribuídos os limitantes de um algoritmo para a solução são diferentes. Ao invés de latência de rede, por exemplo, o tempo de um desenvolvedor liberar um objeto para modificação é o tempo mais longo do processo.

Nesse trabalho foi mostrado como alguns algoritmos de sistemas distribuídos podem ser aplicados a requisitos atuais de alguns sistemas de versionamento distribuídos. Porém, ainda resta um requisito importante a ser abordado. Em um sistema de versionamento distribuído é presumido que nem todos os usuários do sistema estão ativos ao mesmo tempo, além de que, o sistema precisa ser tolerante a falhas. Todos os algoritmos apresentados neste trabalho não são tolerantes a falhas e, portanto, não atendem em totalidade às necessidades desses sistemas. Para tal, como trabalhos futuros, algoritmos como o proposto em [Sopena et al. 2005] podem vir a ser adaptados e resolver esse problema.

Referências

Bertier, M., Arantes, L., and Sens, P. (2006). Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. *Journal of Parallel and Distributed Computing*, 66(1):128 – 144.

Bouabdallah, A. and Laforest, C. (2000). A distributed token-based algorithm for the dynamic resource allocation problem. *SIGOPS Oper. Syst. Rev.*, 34(3):60–68.

Dijkstra, E. W. (1968). *Cooperating Sequential Processes*. Programming Languages. Academic Press, New York. First Published as EWD 123, Math Dept., Technological U., Eindhoven, The Netherlands (1965).

Kakugawa, H. (1995). *A Study on Distributed k-Mutual Exclusion Algorithms*. Tese de Mestrado. Hiroshima University.

Naimi, M., Trehel, M., and Arnold, A. (1996). A log (n) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1 – 13.

Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. *Operating Systems Review*, 25(2):47–50.

Sopena, J., Arantes, L., Bertier, M., and Sens, P. (2005). A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *European Conference on Parallel Processing*, pages 654–663. Springer.