Leonardo Santos Paulucio

Relatório do 1º Trabalho de Processamento Paralelo e Distribuído

Vitória - ES

22 de Maio de 2018

Leonardo Santos Paulucio

Relatório do 1º Trabalho de Processamento Paralelo e Distribuído

Trabalho apresentado à disciplina de Processamento Paralelo e Distribuído do curso Engenharia da Computação da Universidade Federal do Espírito Santo como requisito parcial de avaliação.

Professor: João Paulo A. Almeida

Universidade Federal do Espírito Santo

Engenharia da Computação

Vitória - ES

22 de Maio de 2018

Sumário

1 Introdução

Nas últimas décadas houve uma crescente demanda na área de processamento de dados, o que exigiu o desenvolvimento de máquinas mais poderosas. Porém, apesar da grande capacidade existentes nos novos computadores quando comparados aos primeiros a serem produzidos e até mesmo a computadores de uma década atrás principalmente ao se analisar capacidade de processamento e memória, eles ainda não são capazes de atender essas demandas de processamentos sozinhos. Por esse motivo foram desenvolvidas várias técnicas de concorrência o que contribuiu para a criação dos Sistemas Distribuídos.

Esse trabalho tem por objetivo praticar programação paralela usando o *middleware* JavaRMI e realizar a análise de desempenho em um cluster de computadores. Ele consistirá na implementação de uma arquitetura mestre/escravo para realizar um ataque de dicionário em uma mensagem criptografada.

2 Implementação

2.1 Estrutura de Dados

2.2 Decisões de Projeto

2.3 Cliente

O programa cliente recebe como argumentos: o nome do arquivo criptografado, o trecho conhecido do texto original e opcionalmente um terceiro parâmetro que indica o tamanho do vetor de *bytes* aleatório que será gerado em caso do arquivo criptografado não existir.

Assim, o cliente é responsável por localizar o mestre utilizando o *Registry* e solicitar o serviço de ataque através do método *attack*. Ao solicitar o serviço o cliente passa o arquivo criptografado e o trecho conhecido. Caso o nome do arquivo fornecido como argumento para o programa cliente seja inválido o programa cliente é responsável por gerar um vetor aleatório de bytes, cujo tamanho será igual ao 3º parâmetro fornecido ou, caso esse não exista, um tamanho aleatório na faixa de 1Kb a 100Kb.

O comando utilizado para iniciar o cliente é:

```
java br.inf.ufes.ppd.application.Client <arquivo> <Trecho conhecido> [<Tamanho do
    vetor de bytes>]
```

2.4 Escravo

o q faz e comandos para iniciar

2.5 Mestre

2.6 Problemas Durante a Implementação

Durante o desenvolvimento do trabalho alguns problemas surgiram fazendo com que fosse necessário realizar algumas mudanças na estrutura de dados e na lógica de programação. A maioria deles ocorreu durante o desenvolvimento do mestre, visto que é o componente mais complexo do trabalho já que é responsável por gerenciar todos os escravos e ataques que ocorrem, além de atender as requisições que chegam dos cliente.

2.6.1 Cliente e Escravo

O cliente foi o primeiro a ser implementado. Devido a sua simplicidade não foram encontrados problemas durante sua implementação.

Já na implementação do escravo, que também não era muito complicada, um problema ocorreu, apesar de ter sido de fácil solução.

O problema que surgiu foi durante a implementação do método *startSubAttack*. Nos primeiros testes notou-se que os escravos não realizavam outros sub-ataques em paralelo, assim, ao analisar o código percebeu-se que não estavam sendo criadas *threads* no método para realizar o ataque, dessa forma o escravo ficava bloqueado durante a execução de um ataque não atendendo outras requisições que chegavam. Ao se criar *threads* esse problema foi solucionado.

Outro problema que surgiu durante a implementação do escravo foi o modo de verificar se o trecho conhecido existia no arquivo descriptografado, podendo assim ser um candidato a ser o arquivo original. Essa verificação estava sendo feita da seguinte forma: o array de bytes descriptografado era transformado em uma *String* e a partir dela era usado o método dessa classe chamado *contains*, que verifica se um uma *substring* existe na *string*. Porém ao se converter esse *array* de *bytes* em uma *string* o Java utiliza uma codificação padrão interna que pode não ser a mesma do arquivo que foi criptografado fazendo com que várias ou nenhuma chave candidata possa ser encontrada.

2.6.2 Mestre

O grande problema durante a implementação do mestre foi relacionado a estabelecer uma boa estrutura de dados para que fosse possível realizar o gerenciamento dos escravos e ataques. No início do desenvolvimento a estrutura utilizada não possuía a ideia de subataques, isso fez com que surgisse um problema na hora da divisão de serviço entre os escravos.

Como cada ataque possuía um número único de identificação ao dividir o trabalho entre os escravos eles recebiam esse mesmo número, assim, quando um escravo mandava um checkpoint, duas coisas estavam ocorrendo:

- Quando um escravo que recebia a primeira parte do serviço enviava um checkpoint ele era armazenado na estrutura de controle do ataque, porém quando um escravo que recebeu um intervalo de índices maior ao enviar o primeiro checkpoint já eliminava a informação do primeiro escravo, pois o checkpoint mais atual era salvo no lugar do anterior na estrutura de controle do ataque. Assim não se tinha uma forma de verificar o status de ataque de cada escravo, pois o que recebia o maior índice "dominava"nos checkpoints.

- A verificação para determinar se um trabalho havia terminado era baseada no índice recebido, se ele fosse igual ao último significava que havia terminado. Assim, quando o escravo que recebeu o maior índice terminava seu trabalho o mestre verificava que o serviço tinha acabado e enviava a resposta ao cliente, porém os outros escravos que estavam trabalhando no mesmo ataque podiam não ter terminado a sua parte. Com isso, o mestre enviava uma resposta de um trabalho incompleto para o cliente.

A solução para resolver esse problema foi a criação de subataques para um ataque. Dessa forma, um ataque solicitado pelo cliente é dividido em vários subataques e cada escravo fica responsável por um, assim, ao enviar um checkpoint estará salvando em seu próprio subataque o andamento dos índices, fazendo com que o primeiro problema não ocorresse mais. Outro ponto, é que verificar se um trabalho terminou basta apenas que o mestre cheque se todos os subataques referentes àquele ataque foram terminados, isso garante que o mestre só irá enviar a resposta ao cliente quando todos os escravos tiverem terminado sua tarefa.

Outro problema que surgiu durante o desenvolvimento do mestre foi com relação a concorrência no acesso de variáveis. Existiram algumas partes da estrutura que não estavam tendo sua execução serializada com o *synchronized* de Java, e ao se adicionar vários clientes e escravos ocorreram exceções relacionados a concorrência. Um caso onde esse problema ocorreu foi quando o mestre ficava checando se um trabalho havia terminado para enviar a resposta ao cliente. Para realizar essa verificação o mestre ficava checando a todo o momento uma variável booleana que representava o *status* do trabalho, se fosse falsa o trabalho não tinha terminado, caso contrário já tinha terminado. O problema era que quando o trabalho terminava e essa váriavel de *status* ia ser modificada o mestre concorrentemente checava ela fazendo com que ocorresse exceção de acesso concorrente. Esse problema foi corrigido utilizando a opção *wait* e *notify* de Java que faz com que a *thread* que verifica o *status* do trabalho ficasse dormindo e quando o trabalho estivesse terminado ele era notificado e enviava a resposta para o cliente.

3 Interoperabilidade

A interoperabilidade do trabalho foi testada com os seguintes grupos:

- Grupo: David Morosini e Gustavo Monjardim.
- Grupo: André Barreto e Eric Santos.
- Grupo: Eduardo e Gustavo.
- Grupo: Eduardo e Thiago Borges.

3.1 Problemas Encontrados

Um problema que aconteceu durante um dos testes de interoperabilidade foi no uso de um roteador, quando todo mundo estava conectado à rede um dos grupos abria o mestre e os outros criavam escravos, porém percebeu-se que havia uma demora de cerca de 2 a 3 minutos para que o mestre recebesse a solicitação e registrasse o cliente. E mais, ao se criar um ataque havia uma demora muito grande para que os escravos recebessem a solicitação do mestre, isso fez com que fosse inviável o uso do roteador para a realização dos testes.

Um outro problema que ocorreu foi que um grupo havia modificado a interface original adicionando um construtor com isso ocorria a exceção *Unmarshalling Exception* que fazia com que não encontrasse as classes, pois em Java ao adicionar algum método a uma interface ele a considera uma nova versão diferente da original.

Problema dos indices

O restante dos problemas basicamente se relacionavam com o uso do registry

4 Análise de Desempenho

Após a implementação do trabalho foram realizadas análises de desempenho com o objetivo de se conseguir observar se realmente há uma melhoria ao se utilizar sistemas distribuídos. As análises estão nas seções seguintes.

4.1 Máquinas e Equipamentos Utilizados

Para a realização das análises foram utilizados dois tipos de computadores:

1. Para a análise de desempenho de paralelismo em uma única máquina com vários escravos foi utilizado um notebook com processador *Intel Core i5-2410M CPU 2.30GHz x 4* e 6GB de memória RAM com o Sistema Operacional *Linux Mint KDE 64-bit*.

2. Já para a análise de desempenho distribuído foram utilizados os computadores do laboratório de graduação (LabGrad) que possuem as seguintes configurações: processador *AMD Athlon(tm) Dual Core Processor 5000B* com 4GB de memória RAM e o Sistema Operacional utilizado foi o *Ubuntu 16.04.4 LTS*.

Inicialmente, a ideia era realizar todos os testes nas máquinas do LabGrad porém essa divisão dos testes foi realizada pois ambos os LabGrads estavam reservados para aulas, e quando não estavam os outros alunos da graduação estavam utilizando, fazendo com que a disponibilidade de tempo nos laboratórios fosse bem pequena.

Outra questão é que não seria possível testar o sistema em várias máquinas em casa já que só tinha disponível um computador, sendo assim, optei por fazer a análise de paralelismo no computador pessoal e deixar a análise de desempenho distribuído para ser realizada no LabGrad.

4.2 Desempenho de Paralelismo em uma Única Máquina

4.3 Desempenho Distribuído em Várias Máquinas

5 Conclusão

Ao final desse trabalho é possível notar que um sistema distribuído é uma ferramenta muito poderosa pois permite aproveitar recursos de diferentes equipamentos para executar uma tarefa em comum.

Com a distribuição do serviço entre as máquinas é possível obter uma melhora significativa no tempo de resposta speedup. Observa-se ainda que não é possível atingir o speed up ideal segundo aLei de Amihdal devido ao overhead existente na rede, processamento, etc.

Outro ponto que é importante e que foi possível notar durante a implementação desse trabalho é a complexidade de funcionamento que existe na máquina responsável por realizar todo o gerenciamento, nesse caso representada pelo mestre. Ele tem que lidar: com o controle de máquinas ativas, com a distribuição das tarefas, com o controle do status das tarefas, com o controle de concorrência, redistribuição de tarefas, entre outros. Além de todas essas tarefas ele ainda tem que cuidar do acesso concorrente à variáveis de controle, que é um grande problema ao se trabalhar com muitas *threads*, assim, percebe-se a necessidade que sempre vai existir para que algum pedaço do programa não seja paralelizável.

falar das analises

Referências