1º Trabalho

Processamento Paralelo e Distribuído 2020/01 EARTE
Arquitetura Mestre-Escravo com Java RMI:
Ataque de Dicionário em Mensagem Criptografada

Prof. João Paulo Andrade Almeida

23 de outubro de 2020

O objetivo deste trabalho é praticar programação paralela usando middleware JavaRMI e realizar análise de desempenho em um *cluster* de computadores.

O trabalho deverá ser realizado em duplas. Plágio (cópia entre diferentes grupos) não será tolerado, e grupos com trabalhos copiados (mesmo que parcialmente) receberão nota mínima no trabalho como um todo.

A organização do seu código, clareza e detalhamento dos comentários é um critério importante de avaliação. Documente *bugs* conhecidos ("*known bugs*") no seu relatório. (*Bugs* conhecidos documentados são melhores do que *bugs* desconhecidos e não documentados.)

Inclua o código fonte e o relatório (PDF) em um arquivo `.zip` nomeado com `ataque-dicionario` seguido dos nomes dos componentes do grupo separados por hífen. (Exemplo: `ataque-dicionario-JoseSilva-AntonioVieira.zip`). Envie o arquivo para jpalmeida@inf.ufes.br com o nome do arquivo no campo "subject". **O prazo para entrega é dia 20 de novembro.**

1 Implementando arquitetura mestre-escravo

O trabalho consiste em implementar a arquitetura mestre/escravo para realizar um ataque de dicionário em uma mensagem criptografada. O ataque de dicionário consiste em usar um dicionário como fonte de chaves candidatas. No nosso caso, assumiremos que conhecemos um trecho da mensagem (por exemplo sabemos que a string "ufes" aparece na mensagem, ou a string "PDF"). O ataque será realizado decriptografando a mensagem com cada palavra do dicionário (chave candidata) e procurando o trecho conhecido na mensagem decriptografada. Caso o trecho conhecido seja localizado na mensagem decriptografada com a chave candidata, a chave é considerada uma possível chave para a mensagem, e é usada para obter um possível texto para a mensagem.

O algoritmo de criptografia Blowfish será utilizado (http://www.schneier.com/blowfish-download.html).

A arquitetura mestre escravo consiste em um mestre e vários escravos. As interfaces do mestre e dos escravos devem oferecer uma operação que recebe uma mensagem criptografada e realiza o ataque (veja interfaces abaixo com as operações `attack` e `startSubAttack`).

Use o registry para registrar e achar o mestre, com o nome "mestre".

Os escravos devem se registrar com o mestre ao serem inicializados, através de uma operação oferecida pelo mestre (veja interfaces abaixo com a operação `addSlave`). Escravos devem se "re-registrar" a cada 30 segundos. Isto permite que os escravos detectem falhas no mestre e busquem novamente uma referência para o mestre no registry. (Atenção para a implementação de `addSlave`, pois como ela pode ser chamada por vários escravos simultaneamente, você deve proteger o seu código para acesso concorrente à lista de escravos.)

Ao receber uma requisição do cliente, o mestre solicita a cada escravo registrado a realizar o ataque considerando uma parte do dicionário (passando índices das palavras iniciais e finais, como índice da primeira palavra igual a zero invocando a operação `startSubAttack`). Ao achar uma senha candidata, cada escravo informará esta senha ao mestre (através de callback, veja operação `foundGuess`). Os escravos devem informar o mestre a cada 10 segundos qual o índice da última palavra já testada como senha candidata (através de callback, veja operação `checkpoint`). Essa mesma operação deve ser usada também quando o escravo termina seu trabalho.

A partição dos índices do dicionário deve resultar na busca do dicionário por completo.

O dicionário com chaves candidatas encontra-se no arquivo: http://java.sun.com/docs/books/tutorial/collections/interfaces/examples/dictionary.txt (uma "palavra" por linha). (Coloque uma réplica desse arquivo em cada uma das máquinas onde for rodar escravos, não use a sua conta no labgrad para armazenar o arquivo pois estará usando a rede para ler o arquivo.)

Quando a invocação em um escravo gerar exceção (ou não ter enviado mensagem há mais de 20 segundos durante um ataque), este escravo deve ser removido da lista de escravos registrados. O trabalho deve ser redirecionado para outros escravos.

Como estamos tratando de bytes, a mensagem de entrada pode conter qualquer valor que caiba em 8 bits.

O sistema pode ser testado inicialmente em uma única máquina. (Claro, com vários escravos.)

Inclua no relatório todos os comandos utilizados para inicializar o registry, o cliente, o mestre e os escravos.

Descreva no relatório a solução de robustez, indicando as decisões de projeto que você tomou.

Inclua arquivos de teste utilizados (arquivos de no mínimo 50k) e descreva os testes do seu relatório.

2 Interfaces

Para permitir a interoperação entre os clientes, mestres e escravos de diferentes grupos, as interfaces a serem implementadas são padronizadas, e estão listadas na página da disciplina.

Teste a interoperabilidade de sua implementação com ao menos dois outros grupos. Indique no relatório quais foram esses grupos.

3 Programa Cliente

O programa cliente deve:

- Receber um argumento na linha de comandos que indica o nome do arquivo que contém o vetor de bytes (com a mensagem criptografada) e outro argumento que indica a palavra conhecida que consta da mensagem. Caso o arquivo não exista, o cliente deve gerar o vetor de bytes aleatoriamente e salvá-lo em arquivo. Em um terceiro parâmetro pode ser especificado com o tamanho do vetor a ser gerado. Se esse terceiro parâmetro não existe, o tamanho do vetor deve ser gerado aleatoriamente (na faixa 1000 a 100000).
- Invocar o mestre passando o vetor de bytes, e imprimir chaves candidatas encontradas (se houver). Cada mensagem candidata deve ser colocada num arquivo com o nome da chave e a extensão .msg (por exemplo “house.msg” se a chave for house.)

4 Mestre

O mestre deve imprimir uma mensagem a cada callback recebido dos escravos (checkpoint); incluir o o nome do escravo nesta mensagem, índice atual (e mensagem candidata). Isto possibilitará verificar o andamento dos vários escravos.

O mestre deve imprimir também os tempos medidos a partir de `startSubAttack` para cada checkpoint recebido, assim como o índice atual, para podermos verificar andamento no tempo e desempenho de cada escravo. Imprimir também quando receber o último checkpoint.

5 Análise do Desempenho

Faça uma análise do tempo de resposta observado pelo cliente ao requisitar um ataque. O objeto de estudo da análise é o *speed up* da solução paralela. Inclua um relatório para essa análise, que deve considerar diferentes tamanhos da mensagem criptografada, o caso sequencial e o caso de processamento paralelo com vários escravos.

Crie gráficos para esta análise, usando (pelo menos):

- Diferentes tamanhos de vetor

  (crie um gráfico do tempo de resposta x tamanho da mensagem)

- Considere o caso sequencial e pelo menos 2, 3, 4, 5, 6, 7 e 8 escravos em máquinas diferentes

  (crie gráficos do tempo de resposta nesses casos x tamanho da mensagem)

  (crie um gráfico com o *speed up* da solução com diferentes números de escravos x tamanho da mensagem)

(crie um gráfico com a *eficiência* da solução com diferentes números de escravos x tamanho da mensagem)

(crie gráficos do *overhead* de comunicação com diferentes números de escravos x tamanho da mensagem)

Inclua o código usado para estimar o tempo de processamento para o caso sequencial (rode o caso sequencial em todas as máquinas que usar no seu teste para poder considerar o desempenho de cada máquina).

É recomendado que você crie um cliente especial para a avaliação, que automatize a avaliação, capturando muitas amostragens do tempo de resposta para os diferentes tamanhos de vetor. Dica: gere um arquivo com extensão .csv (http://en.wikipedia.org/wiki/Comma-separated_values) que pode ser aberto pelo Excel para criação dos gráficos.

Documente: tipo de máquinas usadas, sistema operacional, memória disponível, configuração de rede. Se necessário, use o laboratório de graduação.

O grau de detalhamento da análise é um fator importante para a avaliação (inclua outros gráficos e dados relevantes para a sua análise).

O que você pode concluir a partir da análise? Inclua suas conclusões no relatório.

6 Implementação do Blowfish

A Java Cryptography Extension (JCE) que é distribuída com a máquina virtual Java a partir da versão 1.4 poderá ser utilizada.

Veja abaixo 2 exemplos de uso da JCE (um programa para criptografar e outro para decriptografar com Blowfish).

```
package br.inf.ufes.ppd;

import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.security.*;

public class Encrypt {

   private static byte[] readFile(String filename) throws IOException
   {
    File file = new File(filename);
       InputStream is = new FileInputStream(file);
        long length = file.length();
        // creates array (assumes file length<Integer.MAX_VALUE)
        byte[] data = new byte[(int)length];
        int offset = 0; int count = 0;
        while ((offset < data.length) &&
             (count=is.read(data, offset, data.length-offset)) >= 0) {
            offset += count;
        }
        is.close();
        return data;
   }

   private static void saveFile(String filename, byte[] data) throws IOException
   {
    FileOutputStream out = new FileOutputStream(filename);
       out.write(data);
       out.close();
   }

   public static void main(String[] args) {
    // args[0] é a chave a ser usada
    // args[1] é o nome do arquivo de entrada

    try {
       byte[] key = args[0].getBytes();
       SecretKeySpec keySpec = new SecretKeySpec(key, "Blowfish");
       Cipher cipher = Cipher.getInstance("Blowfish");
       cipher.init(Cipher.ENCRYPT_MODE, keySpec);
       byte [] message = readFile(args[1]);
       System.out.println("message size (bytes) = "+message.length);
       byte[] encrypted = cipher.doFinal(message);
       saveFile(args[1]+".cipher", encrypted);

    } catch (Exception e) {
       // don't try this at home
       e.printStackTrace();
    }
   }
}
```

```
package br.inf.ufes.ppd;

import javax.crypto.*;
import javax.crypto.spec.*;
import java.io.*;
import java.security.*;

public class Decrypt {

   private static byte[] readFile(String filename) throws IOException
   {
      …
   }

   private static void saveFile(String filename, byte[] data) throws IOException
   {
      …
   }

   public static void main(String[] args) {
      // args[0] é a chave a ser usada
      // args[1] é o nome do arquivo de entrada

      try {
         byte[] key = args[0].getBytes();
         SecretKeySpec keySpec = new SecretKeySpec(key, "Blowfish");

         Cipher cipher = Cipher.getInstance("Blowfish");
         cipher.init(Cipher.DECRYPT_MODE, keySpec);

         byte [] message = readFile(args[1]);
         System.out.println("message size (bytes) = "+message.length);

         byte[] decrypted = cipher.doFinal(message);

         saveFile(args[1]+".msg", decrypted);
      } catch (javax.crypto.BadPaddingException e) {
         // essa exceção é jogada quando a senha está incorreta
         // porém não quer dizer que a senha está correta se não jogar essa
exceção
         System.out.println("Invalid key.");
      } catch (Exception e) {
         // don't try this at home
         e.printStackTrace();
      }

   }

}
```

Os arquivos das interfaces padronizadas assim como Encrypt.java e Decrypt.java serão disponibilizados na página da disciplina.