Universidade Federal da Fronteira Sul
Ciência da Computação

GEX106 – Computação Distribuída
Prof. Andrei Braga
Trabalho 1

Nome: Jardel Osorio Duarte
Matrícula: 1611100062

Respostas.:

1. (10 pontos) A principal motivação para desenvolver e usar sistemas distribuídos vem do desejo de compartilhar recursos.
Cite pelo menos três recursos úteis de serem compartilhados em sistemas distribuídos e exemplifique a sua utilização.

Impressoras, Internet e cloud service computing.

Impressoras: Costumam ser ligadas a um computador ou laptops. As impressoras podem se conectar de forma distribuída e sem sequer ser necessário o uso de internet, apenas com cabeamento(rede), no sistema distribuído é comumente implementado uma em cada ponto de utilização e a chamada pode ocorrer de qualquer dispositivo cabeado a ela.

Internet: Área de maior crescimento nos últimos 50 anos, visou um modelo distribuído desde seu principio, onde para que um computador se conectar é necessário de uma infraestrutura cabeada e diversos dispositivos/servidores se comunicando através de protocolos como UDP, TCP, HTTP, SMTP, entre muitos outros... Este modelo distribuído também garante a funcionalidade de outras distribuições paralelas através dessas camadas já citadas e do modelo, podemos considerar serviços streaming, jogos multiplayers, serviços peer-to-peer(torrents), etc...

Cloud: Nasce a partir do século 21 sendo ofertado por grandes empresas já consolidados no mercado (AWS, GCP, AZURE, WATSON, entre outros). Atualmente utilizado para compilação de dados massivos, banco de dados (armazenamento, tratamento e análise), machine learning(AI), e diversos. Funciona no modelo client-server, entretendo o cliente usa do mainframe para executar tarefas neste servidor online, contabilizando um custo muito menor de processamento e além da garantia de tempo economiza em um possível upgrade para tratar seus dados em um desktop pessoal.

2. (10 pontos) Cite e defina pelo menos três desafios com os quais desenvolvedores têm que lidar ao construir sistemas distribuídos.

Confiabilidade e ordenação em multicast, Sockets e Servidores com horários alternados.

Servidores com horários alternados: Comum acontecer pelo fuso horário, porém hoje já não é um problema arbitrário, embora ainda assim possa ocorrer entregas de pacotes com prazos alterados, enfileiramento de pacotes de forma errônea.

Sockets: existem exceptions no modelo RMI, gerados à partir de os programas podem encontrar muitos tipos de erros e condições inesperadas, de diversos graus de gravidade. Durante a execução de um método, muitos problemas diferentes podem ser descobertos: por exemplo, valores inconsistentes nas variáveis do objeto ou falhas nas tentativas de ler ou gravar arquivos ou soquetes de rede.

Confiabilidade e ordenação em multicast: Embora utilize de modelos FIFO, para send e receive o multicast possui além de “as garantias de confiabilidade, a comunicação em grupo exige garantias extras em termos da ordem relativa das mensagens entregues para vários destinos. A ordem não é garantida pelas primitivas de comunicação entre processos. Por exemplo, se o multicast é implementado por uma série de mensagens de um para um, elas podem ficar sujeitas a atrasos arbitrários. Problemas semelhantes podem ocorrer se for usado multicast IP. Para levar isso

em conta, os serviços de comunicação em grupo oferecem multicast ordenado, com a opção de uma ou mais propriedades.
citando este problema por termos visto em pratica.

3. (15 pontos) Os itens a seguir tratam das arquiteturas cliente-servidor e peer-to-peer, duas arquiteturas comumente adotadas em sistemas distribuídos:

(a) Descreva como processos interagem em um sistema distribuído para cada uma destas arquiteturas.

O peer-to-peer possui processos iguais nos dois lados da comunicação, onde as funções necessárias para o sistema ser distribuído devem estar implementadas em todos os processos, sendo assim a simetria é dada por cada processo ter a função cliente e servidor ao mesmo tempo.

O client-server é diferente, onde o servidor tem a função de ouvir ou aguardar(listen) uma requisição do cliente, onde após o envio da solicitação existe um retorno do servidor e estes dispositivos começam a trocar datagramas(packages), a comunicação acontece de forma remota e em muitos casos faz uso de sockets para retornos realtime. Embora o client tenha somente a implementação de processos client e o servidor tenha apenas a implementação de processos server.

(b) Dê um exemplo de uma situação onde a arquitetura cliente-servidor é empregada. Faça o mesmo para a arquitetura peer-to-peer.

Redes sociais, e-commerces atualmente com um histórico crescente de acessos;
Sites para torrents;

(c) Cite uma vantagem e uma desvantagem da arquitetura cliente-servidor em relação à arquitetura peer-to-peer.

Transmissões limitadas apenas por banda sem dependências de seeders, acredito que todo usuário p2p já sofreu ao menos uma vez na vida por não conseguir loadar um arquivo que queria para o fim de semana ou no momento especifico, ocasiões impossíveis em um modelo client-server a não ser que o servidor esteja tombado(também pouco comum).

4. (20 pontos) Os itens a seguir tratam do mecanismo de chamada de procedimento remoto e do exemplo prático deste mecanismo usando Sun RPC visto em aula:

(a) O mecanismo de chamada de procedimento remoto tenta oferecer pelo menos dois tipos de transparência ao programador. Cite quais são estas transparências e explique o que elas significam.

A RMI está ligada intimamente com a RPC e por isto as transparência também estão relacionadas porém algumas destas propriedades de transparência que possuem certa semelhança, um exemplo são as chamadas locais e remotas empregam a mesma sintaxe mas as interfaces remotas normalmente expõem a natureza distribuída da chamada subjacente e suportando exceções remotas exemplos:

As diferenças a seguir levam a uma maior expressividade na programação de aplicações
e serviços distribuídos complexos.

• O programador pode usar todo o poder expressivo da programação orientada a objetos no desenvolvimento de software de sistemas distribuídos, incluindo o uso de objetos, classes e herança, e também pode empregar metodologias de projeto orientado a objetos relacionadas e ferramentas associadas.

• Complementando o conceito de identidade de objeto dos sistemas orientados a objetos, em um sistema baseado em RMI, todos os objetos têm referências exclusivas (sejam locais ou remotos) e tais referências também podem ser passadas como parâmetros, oferecendo, assim, uma semântica de passagem de parâmetros significativamente mais rica do que na RPC.

(b) A implementação de uma chamada de procedimento remoto envolve vários componentes de software. Cite os componentes que estão explicitamente implementados em arquivos dados no exemplo prático visto em aula. Indique o arquivo onde cada componente citado está implementado.

O middleware é um dos principais componentes de softwares que estão implementados nas camadas entre os objetos do aplicativo e os módulos remotos e somente no middleware as funções de seus componentes são:

• O proxy: A função de um proxy é tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador; mas, em vez de executar uma invocação local, ele a encaminha em uma mensagem para um objeto remoto. Ele oculta os detalhes da referência de objeto remoto, do empacotamento de argumentos, do desempacotamento dos resultados e do envio e recepção de mensagens do cliente. Existe um proxy para cada objeto remoto a que um processo faz uma referência de objeto remoto. A classe de um proxy implementa os métodos da interface remota do objeto remoto que ele representa. Isso garante que as invocações a métodos remotos sejam apropriadas para o objeto remoto em questão. Entretanto, o proxy implementa os métodos de uma forma diferente. Cada método do proxy empacota uma referência para o objeto alvo, o OperationId e seus argumentos em uma mensagem de requisição e a envia para o objeto alvo. A seguir, espera pela mensagem de resposta; quando a recebe, desempacota e retorna os resultados para o invocador.

• Despache: Um servidor tem um despachante e um esqueleto para cada classe que representa um objeto remoto. Em nosso exemplo, o servidor tem um despachante e um esqueleto para a classe do objeto remoto B. O despachante recebe a mensagem de requisição do módulo de comunicação e utiliza o OperationId para selecionar o método apropriado no esqueleto, repassando a mensagem de requisição. O despachante e o proxy usam o mesmo OperationId para os métodos da interface remota.

Observação do método e seus parâmetros:

```
public byte[ ] doOperation (RemoteRef s, int operationId, byte[ ] arguments){
        /*
        Envia uma mensagem de requisição para o servidor remoto e retorna a resposta.
        Os argumentos especificam o servidor remoto, a operação a ser invocada e
        os argumentos dessa operação.*/
}
```

• Esqueleto: A classe de um objeto remoto tem um esqueleto, que implementa os métodos da interface remota, mas de uma forma diferente dos métodos implementados no servente que personifica o objeto remoto. Um método de esqueleto desempacota os argumentos na mensagem de requisição e invoca o método correspondente no servente. Ele espera que a invocação termine e, em seguida, empacota o resultado, junto às exceções, em uma mensagem de resposta que é enviada para o método do proxy que fez a requisição.

Observação do modelo visto em aula(em linguagem C):

```
#include <stdio.h>
#include <time.h>
#include <rpc/rpc.h>
#include "date.h"

long bin_date() {
        long result;

   result = time((long *)0);
        return result;
}

char *str_date(long bintime) {
        char *result;

   result = ctime(&bintime);

        return result;
}

int main(int argc, char **argv) {
  long lresult; /* return from bin_date */
  char *sresult; /* return from str_date */

  if (argc != 1) {
    fprintf(stderr, "usage: %s\n", argv[0]);
    exit(1);
  }

  /* First call the remote procedure bin_date */
  lresult = bin_date();
  printf("time = %ld\n", lresult);

  /* Now call the remote procedure str_date */
  sresult = str_date(lresult);
  printf("date = %s", sresult);

  exit(0);
}
```

implementação da local.c com os métodos para serem chamadas no servidor remoto, e execute no cliente tem como objetivo imprimir na tela a nossa data atual. A operação só é concebida através de uma interface que publica o evento usando uma linguagem de definição de interface, que neste exemplo foi dada como:

```
program DATE_PROG {
  version DATE_VERS {
    long BIN_DATE(void) = 1;
    string STR_DATE(long) = 2;
```

```
  } = 1;
} = 0x31423456;
```

Identificamos os procedimentos com números(neste exemplo 1 e 2), sendo essas cada um deles variáveis relacionadas as funções implementadas no local.c. Após finalizar a função “version DATE_VERS” identificamos ela com um índice para interface criada e finalizamos o programa com um número hexadecimal identificador do programa; Essa interface agora é o limitador de operações de execuções para este modelo remoto.

Os stubs desta interface se encontram em um outro arquivo date.h

```
#else /* K&R C */
#define BIN_DATE 1
extern  long * bin_date_1(); //funções de chamada STUB CLIENT
extern  long * bin_date_1_svc(); //STUB SERVIDOR
#define STR_DATE 2
extern  char ** str_date_1(); //funções de chamada STUB CLIENTE
extern  char ** str_date_1_svc(); //STUB SERVIDOR
extern int date_prog_1_freeresult ();
#endif /* K&R C */
```

sendo as funções bin_date e str_date implementações responsáveis pelos stubs CLIENT E SERVER.

No arquivo client.c, tem funções
```
  /* First call the remote procedure bin_date */
  if ((lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
  }
  printf("time on host %s = %ld\n", server, *lresult);

  /* Now call the remote procedure str_date */
  if ((sresult = str_date_1(lresult, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(4);
  }
  printf("date on host %s = %s", server, *sresult);
```
similares aos encontrados no local.c, entretando agora as funções correspondem ao jeito que é especificado no stub, disparando outra requisição.

```
  /* First call the remote procedure bin_date */
  if ((lresult = bin_date_1(NULL, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(3);
  }
  printf("time on host %s = %ld\n", server, *lresult);

  /* Now call the remote procedure str_date */
  if ((sresult = str_date_1(lresult, cl)) == NULL) {
    clnt_perror(cl, server);
    exit(4);
  }
  printf("date on host %s = %s", server, *sresult);
```

Em sequência os file date_cltn.c correspondente ao cliente é invocado e gera a requisição para mandar para o servidor (ou seja estamos operando no stub do cliente), esses arquivos são gerados automaticamente a partir do date.x.

Ao chegar no servidor temos os procedimentos também criados em um file data_svc.c onde encontramos o despachante.
Função do despachante e procedimentos stubs do lado do servidor:
date_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
. //procedimentos
.//procedimentos
. //procedimentos

```
        case BIN_DATE:
                _xdr_argument = (xdrproc_t) xdr_void;
                _xdr_result = (xdrproc_t) xdr_long;
                local = (char *(*)(char *, struct svc_req *)) bin_date_1_svc;
                break;

        case STR_DATE:
                _xdr_argument = (xdrproc_t) xdr_long;
                _xdr_result = (xdrproc_t) xdr_wrapstring;
                local = (char *(*)(char *, struct svc_req *)) str_date_1_svc;
                break;
```

Se a chamada for relacionada ao bin_date() ele vai chamar o procedimento stub do lado servidor relacionado ao caso bin_date, do contrario se a chamada for str_date, o despachante chama o procedimento stub do servidor relacionado a este outro método.

Este despachante se comunica com a implementação do servidor server.c onde tem funções extremamente relacionadas com as do local.c.

```
#include <time.h>
#include <stdlib.h>
#include "date.h"

long *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
        static long  result;

   result = time((long *)0);

   printf("time = %ld\n", result);

        return &result;
}

char **
str_date_1_svc(long *bintime, struct svc_req *rqstp)
{
        static char * result;
```

```
    result = ctime(bintime);

    printf("date = %s", result);

        return &result;
}
```

Ou seja, criamos os procedimento de local.c e server.c por serem serviços específicos que queremos na comunicação e após isso geramos a partir da interface os stubs e dispatchers para que haja o chamada no servidor e o retorno para execução remota o cliente; fazendo então possível a troca desses datagramas com a compilação das nossas funções implementadas.

5. (20 pontos) Os itens a seguir tratam do mecanismo de invocação a método remoto e do exemplo prático deste mecanismo usando Java RMI visto em aula:

(a) A implementação de uma invocação a método remoto envolve vários componentes de software. Cite os componentes que estão explicitamente implementados em arquivos dados no exemplo prático visto em aula. Indique o arquivo onde cada componente citado está implementado.

Funciona praticamente como um middleware garantindo algumas propriedades da comunicação, entretando a RMI codifica e decodifica os processos programados e também os esqueletos gerados na compilação(objetos com métodos que podem ser invocados entre JVM são chamados de objetos remotos), abaixo vamos ver o passo a passo do shapelist, um exemplo visto em aula e aprofundando nos principais componentes implementados lá.

Camadas que compõem o java RMI são:

STUB/esqueleto → oferece as interfaces usadas pelos objetos durante a comunicação tanto no cliente quanto no servidor;

Referência Remota → Cria e gerencia referências para objetos remotos

Camada de Transporte → Implementa o protocolo no formato de solicitação para envio de objetos remotos na rede.

Para garantir a invocação remota do ShapelList alteramos os modelos de uma chamada normal de execução local, e implementamos uma interface ShapeListServant.java com os construtores que antes estavam na nossa ShapeList.java

```
public class ShapeListServant implements ShapeList {
        //foca no implements pois agora estamos
        //implementando para a shapelist, não é uma
        //empresa mas poderia ser
  private Vector theList;

  public ShapeListServant () throws RemoteException {
    theList = new Vector();
  }

  public void newShape(GraphicalObject g) throws RemoteException {
    theList.addElement(g);
```

```
  }

  public Vector allShapes() throws RemoteException {
    return theList;
  }
}
```

e na nossa classe ShapList usamos um extends de uma classe Remote, deixando somente assim:

```
public interface ShapeList extends Remote { //foca no extends
  void newShape(GraphicalObject g) throws RemoteException;
  Vector allShapes() throws RemoteException;
}
```

até então criamos a interface mas não fizemos um binding para gerar um sistema distribuído, sendo assim na classe ShapelistClient apontando uma string com o servidor desejado.

```
public class ShapeListClient {
  public static void main(String args[]) {
    try {
      ShapeList aShapeList = (ShapeList) Naming.lookup("//68.183.118.94/ShapeList");
      System.out.println("Servidor encontrado");
```

E em sequência implementamos um código para instanciar método remoto do servidor para que os clientes possam encontrar o servidor através do binder.

```
public class ShapeListServer {
  public static void main(String args[]){
    try {
      ShapeList aShapeList = new ShapeListServant();
      ShapeList stub =
        (ShapeList) UnicastRemoteObject.exportObject(aShapeList, 0);

      Registry r = LocateRegistry.createRegistry(1099);
      r.rebind("ShapeList", aShapeList);
      System.out.println("Servidor ShapeList pronto");
    }
    catch(Exception e) {
      System.out.println("ShapeList server main " + e.getMessage());
    }
  }
}
```

Na classe ShapeListServer nos criamos um objeto do tipo ShapeListServent(), criamos um stub através da função UnicastRemoteObject.exportObject(aShapeList, 0); função que faz o processo de marshilling/unmarshilling, e ainda criamos o bind associando o shapelist com o objeto que foi criado r.rebind("ShapeList", aShapeList); agora o nosso servidor está preparado para comunicação com o client.

Observações fundamentais: para trabalhar com tipo de dados abstratos é fundamental que usássemos ferramentas de serialização sendo assim fizemos o marshilling/unmarshilling a partir da chamada no objeto, após incluir a biblioteca java.io.serializable

na classe do objeto ficou assim:

```
public class GraphicalObject implements Serializable {
  public String type;
  public Rectangle enclosing;
  public Color line;
  public Color fill;
  public boolean isFilled;
.
.
.
}
```

(b) Considere o método newShape da interface ShapeList definida no exemplo prático visto em aula. Este método é implementado como uma operação idempotente? Justifique sua resposta.

Sim é uma função idempotente e o que justifica isso é que podemos criar mais de um objeto, sendo assim chamamos a função newShape diversas vezes enquanto nosso programa esta sendo executado e faremos isto sem incluir um elemento em sequência, ou seja é uma requisição com entrada e saída que não altera, tendo sempre o mesmo efeito no conjunto.

(c) Cite um exemplo de uma operação idempotente. Faça o mesmo para uma operação não-idempotente. Explique por que cada operação citada é idempotente ou não.

uma operação colocar um elemento em um conjunto é idempotente, pois sempre terá o mesmo efeito no conjunto toda vez que for executada. Podemos citar como exemplo a newShape anteriormente vista.

Uma operação para incluir um elemento em uma sequência não é idempotente, pois amplia a sequência sempre que é executada. Um servidor cujas operações são todas idempotentes não precisa adotar medidas especiais para evitar suas execuções mais de uma vez. Podemos utilizar a allShapes ou theList como exemplo, pois a cada execução ela modifica e amplia os objetos na lista.

6. (25 pontos) Pesquise por um exemplo prático – diferente do que foi visto em aula – do mecanismo de invocação a método remoto usando Java RMI e faça o seguinte:

(a) Descreva o exemplo prático pesquisado. Cite a referência principal para o exemplo e, caso existam, outras referências relevantes.

O exemplo prático é um serviço de criptografia e descriptografia que utiliza de java RMI para conexão cliente servidor. foi implementado por Edílson da Costa do Nascimento, Ighor Amaral Souza, Rodrigo Barroso Gadelha1, estudantes da Universidade do Pará (UFPA). A lógica desenvolvida por eles foi usar da criptografia de chave simétrica com cifra de substituição, que substitui um elemento por outro.
O programa possui 4 classes além da stub e skeleton geradas pela RMI, a primeira é uma classe Criptografia que contém os métodos criptografar e descriptografar.

```
public interface Criptografia extends java.rmi.Remote {
    public String criptografar (String a) throws java.rmi.RemoteException;
    public String descriptografar (String a) throws java.rmi.RemoteException;
```

```
}
```

Logo em seguida tem a classe da interface remota, nomeada como CriptografiaImpl que implementa (está seria nossa interface que gera o nosso Stub posteriormente) a classe remota Criptografia com a lógica que foi comentada anteriormente, gerador aleatório de hash.

E no servidor temos o binding acontecendo pro servidor local na função Naming.rebind("//localhost/criptoService", obj), passando nosso obj.

```
import java.rmi.Naming;

public class ServidorCriptografia{
	public ServidorCriptografia(){
		try{
			Criptografia obj = new CriptografiaImpl();
			Naming.rebind("//localhost/criptoService", obj);
		}
		catch (Exception e){
			System.out.println("Erro: "+e);
		}
	}

	public static void main(String[] args){
		new ServidorCriptografia();
	}
}
```

observação importante é que se o servidor estiver ouvindo em uma máquina diferente a chamada no file client.class deve conter o host especifico para direcionar a execução remota.

Exemplo

Compile: java ClienteCriptografia 192.168.12.24

Os arquivos ClienteCriptografia.java e CriptografiaImpl.java estaram sendo anexados junto com a entrega deste trabalho, portando estarei focando na compilação e demonstração da execução no host local.

Este trabalho esta disponível em <http://www.linhadecodigo.com.br/artigo/2831/exemplo-pratico-do-uso-de-rmi-em-sistemas-distribuidos-servico-de-criptografia.aspx#ixzz6rZQgFMg4>

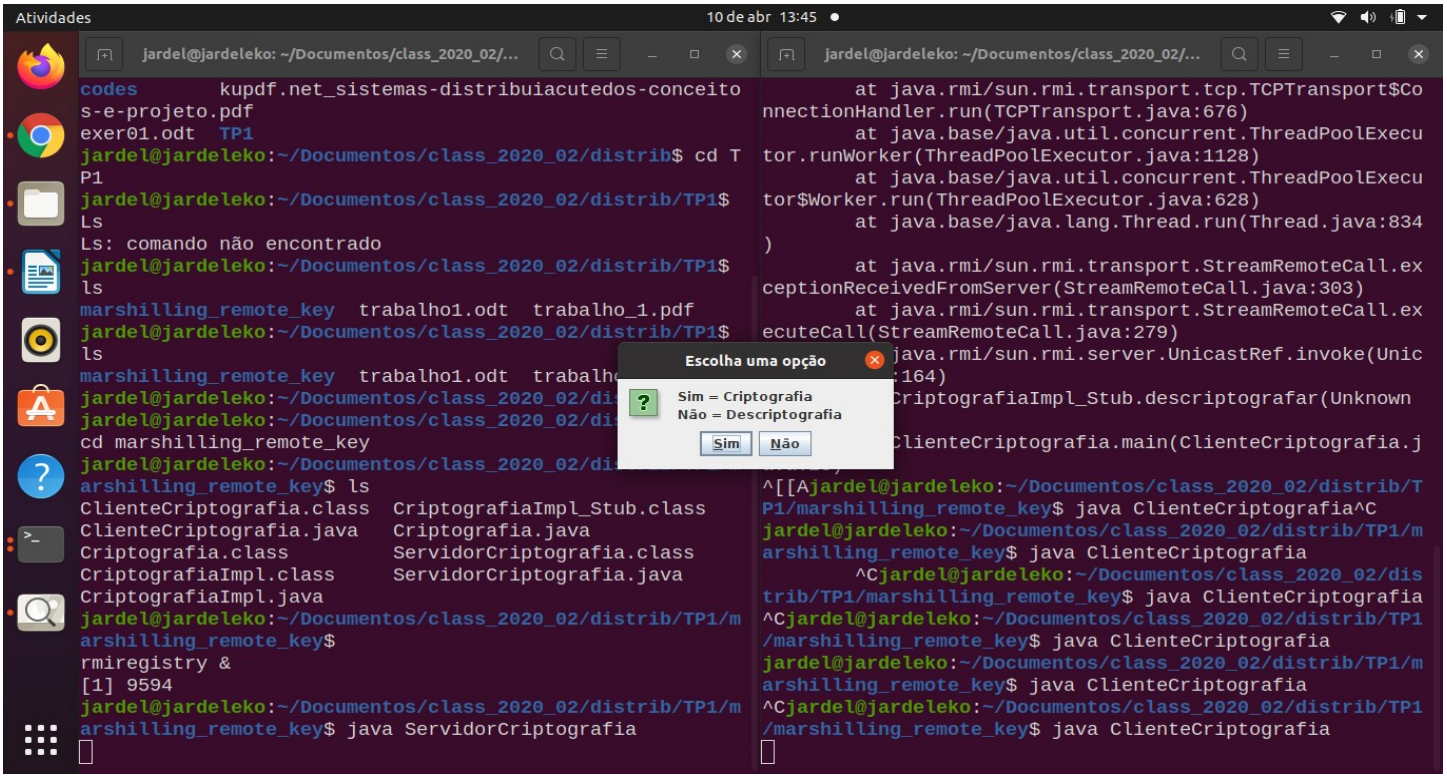
(b) Entenda e execute o exemplo prático (mesmo que apenas localmente no seu computador). Apresente pelo menos uma captura de tela demonstrando o funcionamento da aplicação.

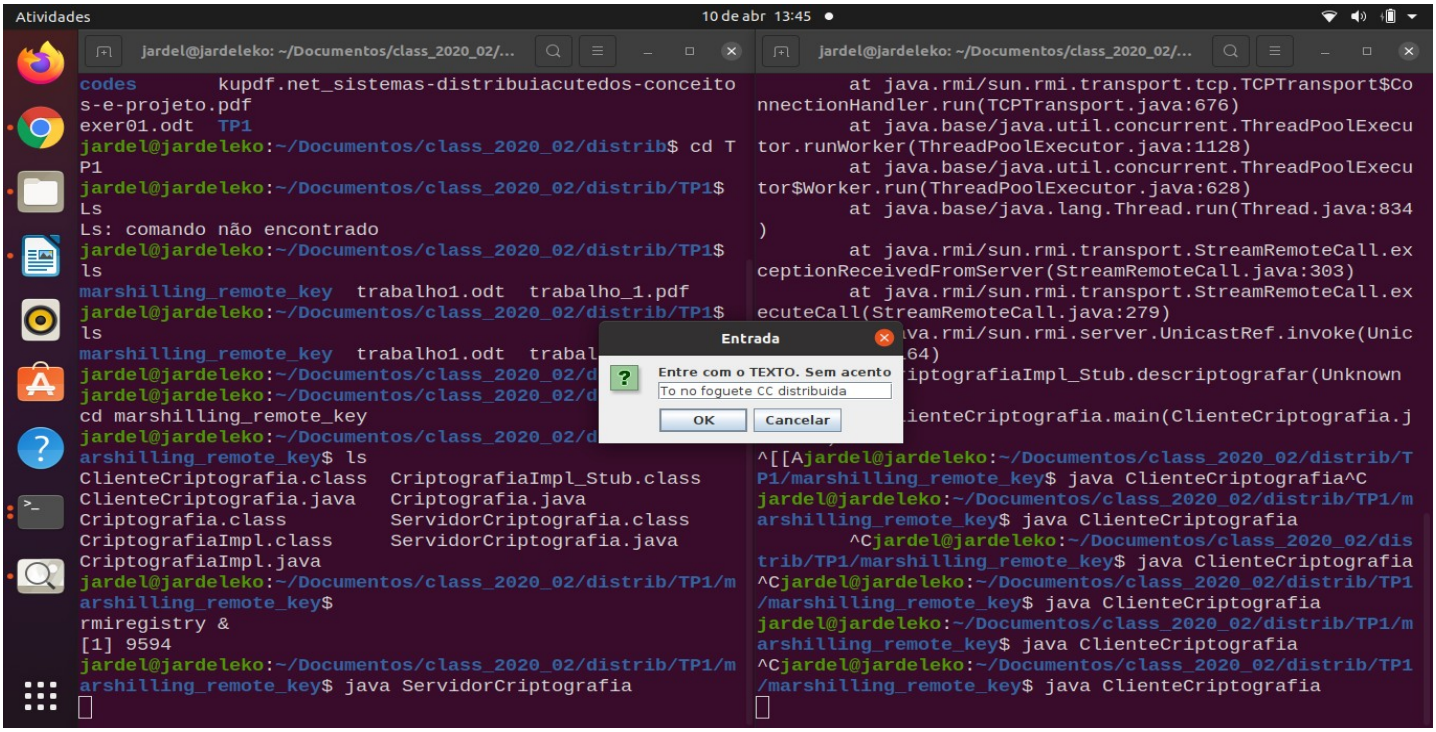
Para execução deste programa foi necessário a instalação do java jdk e também dos seguintes comandos:

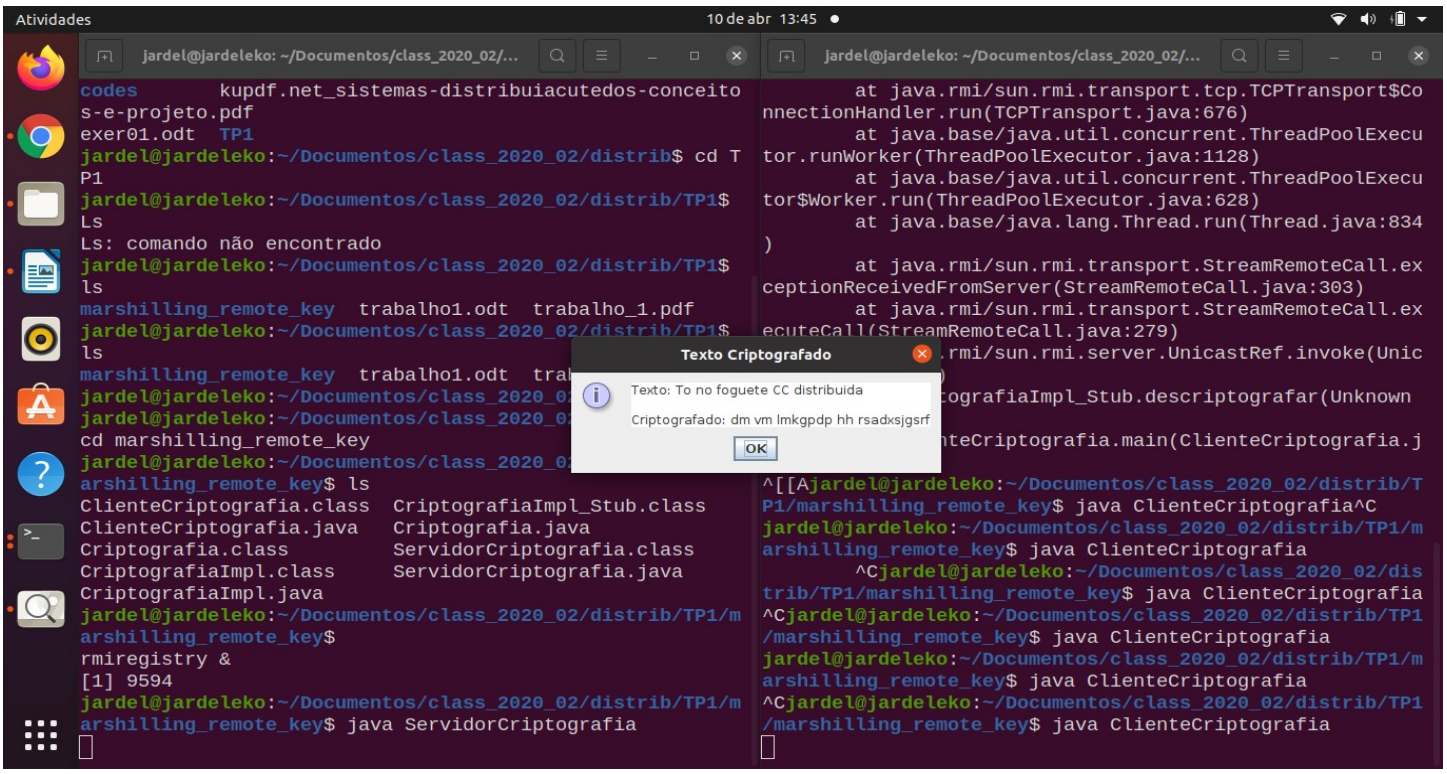
no diretório do arquivo: javac *.java
no diretório do arquivo: rmic CriptografiaImpl
no diretório do arquivo: rmiregistry &
no diretório do arquivo: java ServidorCriptografia

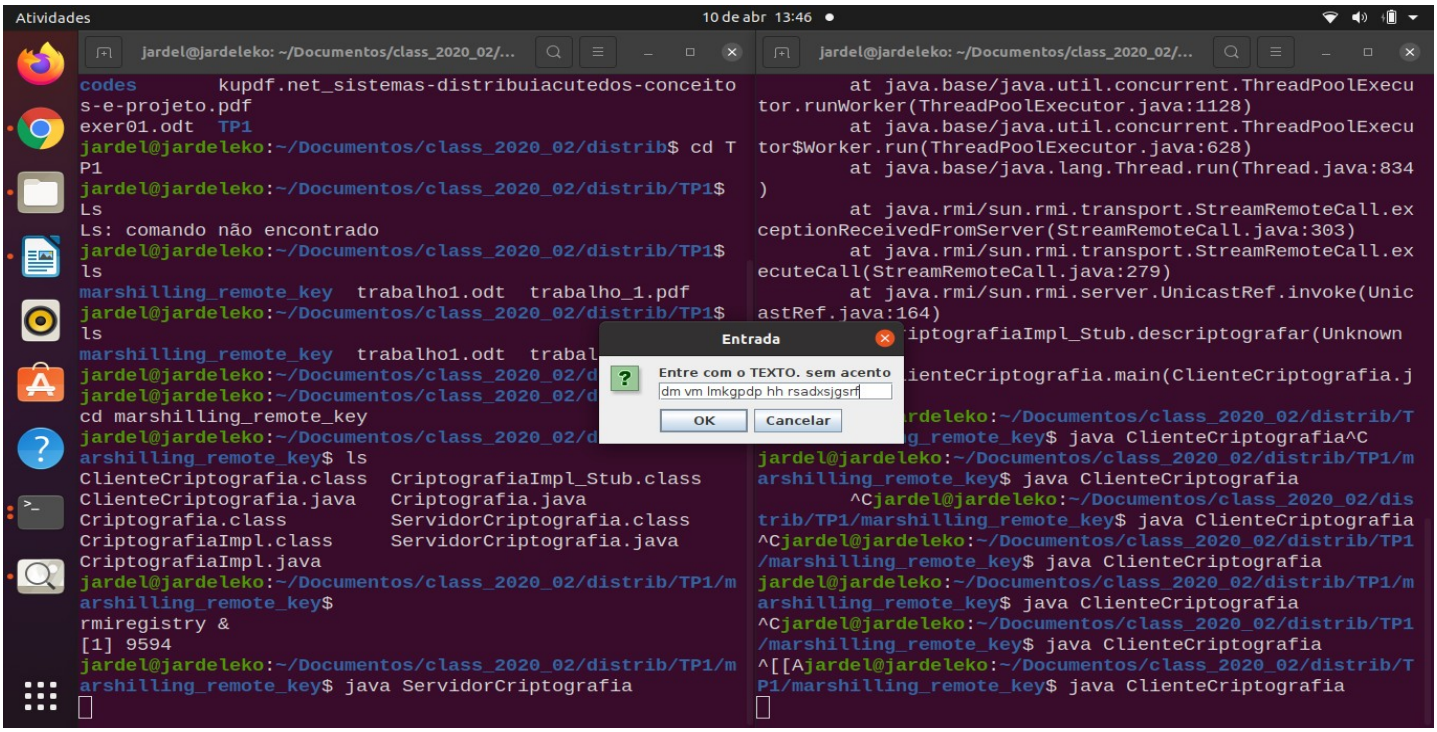
no diretório do arquivo(outro terminal): java ClienteCriptografia

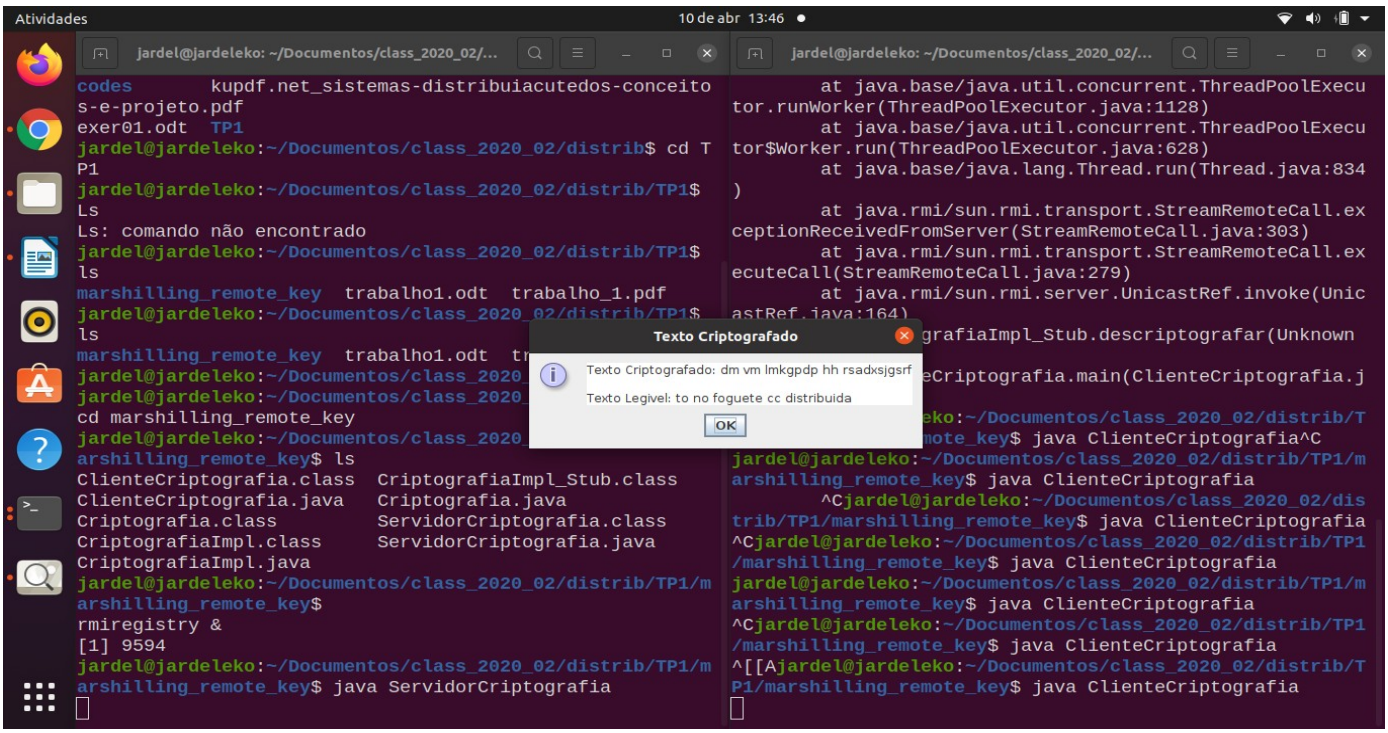
após feito estes passos o resultado foram:






Atividades
10 de abr 13:45
jardel@jardeleko: ~/Documentos/class_2020_02/...
codes kupdf.net_sistemas-distribuiacutedos-conceitos-e-projeto.pdf
exer01.odt TP1
jardel@jardeleko:~/Documentos/class_2020_02/distrib$ cd TP1
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ Ls
Ls: comando não encontrado
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ ls
marshilling_remote_key trabalho1.odt trabalho_1.pdf
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ ls
marshilling_remote_key trabalho1.odt
cd marshilling_remote_key
arshilling_remote_key$ ls
ClienteCriptografia.class CriptografiaImpl_Stub.class
ClienteCriptografia.java Criptografia.java
Criptografia.class ServidorCriptografia.class
CriptografiaImpl.class ServidorCriptografia.java
CriptografiaImpl.java
marshilling_remote_key$
rmiregistry &
[1] 9594
marshilling_remote_key$ java ServidorCriptografia
Texto Criptografado
Texto: To no foguete CC distribuida
Criptografado: dm vm lmkgpdp hh rsadxsjgsrf
OK
at java.rmi/sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:676)
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
at java.base/java.lang.Thread.run(Thread.java:834)
at java.rmi/sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:303)
at java.rmi/sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:279)
ClienteCriptografia.main(ClienteCriptografia.j
^[[Ajardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia^C
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia


Atividades
10 de abr 13:46
jardel@jardeleko: ~/Documentos/class_2020_02/...
codes kupdf.net_sistemas-distribuiacutedos-conceitos-e-projeto.pdf
exer01.odt TP1
jardel@jardeleko:~/Documentos/class_2020_02/distrib$ cd TP1
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ Ls
Ls: comando não encontrado
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ ls
marshilling_remote_key trabalho1.odt trabalho_1.pdf
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1$ ls
marshilling_remote_key trabalho1.odt
cd marshilling_remote_key
arshilling_remote_key$ ls
ClienteCriptografia.class CriptografiaImpl_Stub.class
ClienteCriptografia.java Criptografia.java
Criptografia.class ServidorCriptografia.class
CriptografiaImpl.class ServidorCriptografia.java
CriptografiaImpl.java
marshilling_remote_key$
rmiregistry &
[1] 9594
marshilling_remote_key$ java ServidorCriptografia
Entrada
Entre com o TEXTO. sem acento
dm vm lmkgpdp hh rsadxsjgsrf
OK
Cancelar
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
at java.base/java.lang.Thread.run(Thread.java:834)
at java.rmi/sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:303)
at java.rmi/sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:279)
at java.rmi/sun.rmi.server.UnicastRef.invoke(UnicastRef.java:164)
ClienteCriptografia.main(ClienteCriptografia.j
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia^C
jardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^Cjardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia
^[[Ajardel@jardeleko:~/Documentos/class_2020_02/distrib/TP1/marshilling_remote_key$ java ClienteCriptografia



(c) Descreva limitações da aplicação fornecida no exemplo prático. Cite funcionalidades que poderiam ser adicionadas à aplicação.

Acredito que a maior limitação deste trabalho foram as strings respostas, penso que daria para desenvolver um modelo mais fiel as hashings que estamos acostumados(sha1, md5, sha256, entre outras) utilizando de caracteres especiais e números, outra observação é quando temos um espaço ‘ ‘, a string resposta também tem um espaço o que garante maior facilidade para uma possível força bruta. Enfim acredito que este projeto tenha ficado interessante para aplicação RMI e poderia ser utilizado para diversos fins bem como o citado pelos desenvolvedores na documentação, ou seja, garantia de segurança na troca de e-mails corporativos.

Referências:

Sistemas Distribuídos: Conceitos e Projeto. 5. ed. Porto Alegre: Bookman, 2013. George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair.).

Nascimento, E. Souza, I. Gadelha, R. **Exemplo prático do uso de RMI em sistemas distribuídos:** Serviço de Criptografia**.** Curso de Especialização em Redes de Computadores.Instituto de Ciencias Exatas e Naturais - Faculdade de Computação (UFPA). Belém – PA – Brazil

Disponível em: <http://www.linhadecodigo.com.br/artigo/2831/exemplo-pratico-do-uso-de-rmi-em-sistemas-distribuidos-servico-de-criptografia.aspx#ixzz6rZQgFMg4>