Laboratório de Pesquisa em Redes e Multimídia

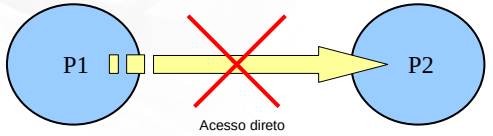
Inter-process Communication (IPC)
Comunicação entre processos

Memória compartilhada
Pipes
Sinais
Sockets
Java RMI

Universidade Federal do Espírito Santo
Departamento de Informática

Comunicação entre processos (1)

- Processos executam em cápsulas autônomas
- Hardware oferece proteção de memória
  - Um processo não acessa outro processo

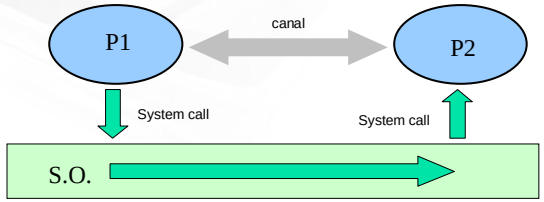


Comunicação entre processos (2)

- Como foi visto, os processos precisam interagir/ cooperar
  - Coordenar o uso de recursos compartilhados
  - Aumentar a velocidade de computação
  - Desenvolvimento modular
  - Atender a requisições simultâneas
  - etc.
- Como interagir ?
  - Via mecanismos de IPC: *Inter-Process Communication*

Comunicação entre processos (3)

- Ocorre através do S.O.
- Implementação de "canais" de comunicação
  - Implicita ou explicitamente



Comunicação entre processos (4)

- Características desejáveis para IPC
  - Rápida
  - Simples de ser utilizada e implementada
  - Um modelo de sincronização bem definido
  - Versátil
  - Funcione igualmente em ambientes distribuídos
- Sincronização é uma das maiores preocupações em IPC
  - Permitir que o *sender* indique quando que um dado foi transmitido
  - Permitir que um *receiver* saiba quando um dado está disponível
  - Permitir que ambos saibam o momento em que podem realizar uma nova IPC

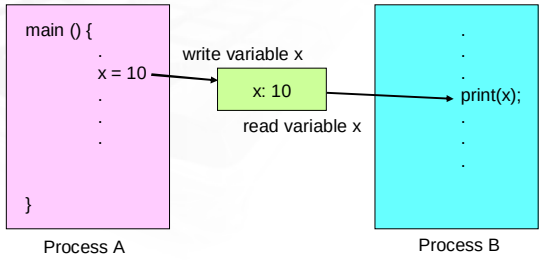
Mecanismos de comunicação

- Fundamentalmente duas abordagens:
  - Suportar alguma forma de espaço de endereçamento compartilhado (processos cooperativos)
    - **Shared memory (mem. compartilhada)**
  - Utilizar mecanismos do próprio S.O. para transportar dados de um processo para outro
    - **Pipes**
    - **Sinais**
    - Sockets
    - etc

Memória Compartilhada (1)

- Quando um mesmo trecho (segmento) de memória encontra-se no espaço de endereçamento de dois ou mais processos
- O S.O. oferece chamadas permitindo a criação da região de memória compartilhada, mas não se envolve diretamente na comunicação entre os processos (e.x., *shmget*, *shmat*)
- Se um processo realiza alguma modificação nesta região, ela é vista por todos os processos que compartilham o segmento de memória

Memória Compartilhada (2)

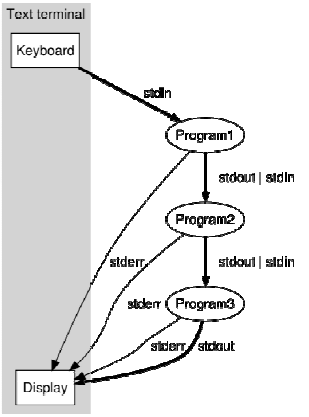


Memória Compartilhada (3)

- Vantagens
  - *Random Access*
    - É possível acessar uma parte específica de uma estrutura de dados e não a estrutura completa
  - Eficiência
    - Corresponde à maneira mais rápida para que dois processos efetuem uma troca de dados
      - Os dados não precisam ser passados ao kernel para que este os repasse aos outros processos, o acesso à memória é direto
- Desvantagens
  - Não existe um mecanismo automático de sincronização
    - Pode exigir o uso de semáforos, locks, etc., por parte dos processos.

Tubos (pipes) (1)

- Conceito inventado para shells Unix
  - Portado para DOS, OS/2, Windows NT, and BeOS
- É uma das formas mais divulgadas de IPC
- Nome origina-se na analogia com um *pipeline*



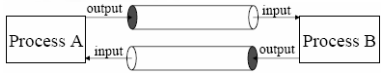
Tubos (pipes) (2)

- Os tubos (ou *pipes*) constituem um mecanismo fundamental de comunicação unidirecional entre processos

- Eles são um mecanismo de I/O com duas extremidades, correspondendo na verdade a filas de caractereres do tipo FIFO
  - Em geral, essas extremidades são implementadas através de descritores de arquivos
- Um pipe tradicional caracteriza-se por ser:
  - Anônimo (não tem nome)
  - Temporário: dura somente o tempo de execução do processo que o criou
- Vários processos podem fazer leitura e escrita sobre um mesmo pipe

Tubos (pipes) (3)

- A capacidade é limitada
  - Se a escrita sobre um pipe continua mesmo depois do pipe estar completamente cheio, ocorre uma situação de bloqueio
- É impossível fazer qualquer movimentação no interior de um tubo.
- Com a finalidade de estabelecer um diálogo entre dois processos usando pipes, é necessário a abertura de um pipe em cada direção.



Tubos (pipes) (5)

- Pipes são muito usadas com redirecionamento de entrada e saída para conectar processos.
- Pipe funciona como um buffer entre os processos

```
ls –l > my.file
sort –n +4 < my.file (n define o tipo de sort,
  no caso numérico e +4 sort pelo campo
  pulando 4)

ls –l | sort –n +4
```


Tubos (pipes) (6)

- Leitura de dados de um pipe remove o dado: pipe não pode ser usado para broadcast
- Dado é byte-stream, não existe delimitação explícita de dados compartilhados
- Se existem multiplos leitores do pipe, o escritor não tem como especificar um leitor em particular

Sinais (1)

- Um sinal é um mecanismo de software usado pelo sistema para informar os processos da ocorrência de eventos "anormais" (e assíncronos) dentro do ambiente de execução
- Podem ser gerados em diferentes situações
  - Chamando *kill()*
  - O kernel usa sinais para notificar um processo da ocorrência de uma exceção de hardware
  - O driver de um terminal manda um sinal a um processo quando uma tecla específica foi apertada
  - ...
- Tipos de sinais:
  - Divisão por zero
  - Acesso inválido à memória
  - Interrupção do programa
  - Término de um processo filho
  - Alarme

Sinais (2)

- Sinais também representam um mecanismo que possibilita a comunicação entre diferentes processos
- Sinais são mensagens pré-definidas, contendo apenas um código numérico
- O processo receptor conhece apenas o tipo do sinal, sem conhecer efetivamente o emissor desse sinal
- Desvantagem
  - Nenhum dado é especificado para ser trocado entre os processos
  - Em geral, o número de sinais a serem utilizados é bastante reduzido
  - Apresenta uma semântica complexa para threads...

Sinais (3)

- Ao receber um sinal:
  - Um processo pode interromper sua execução e desviar para um tratador (handler) previamente definido
  - O sinal pode ser ignorado
  - Ou pode-se confiar no comportamento default do S.O.
    - Abortar o processo, suspender o processo, continuar (resume) a execução do processo

Mais informações...

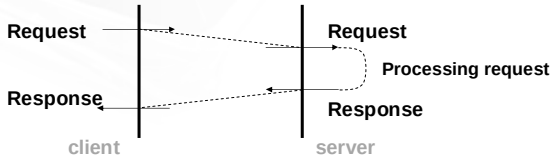
- Stevens, Richard. **UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications**. Prentice Hall, 1999. ISBN 0-13-081081-9
- Tutorial online: POSIX Threads Programming
  - http://www.llnl.gov/computing/tutorials/pthreads/
- Nenad Marovac. "**On interprocess interaction in distributed architectures**", ACM SIGARCH Computer Architecture News, 11(4), 1983
  - http://portal.acm.org/citation.cfm?id=641598&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618

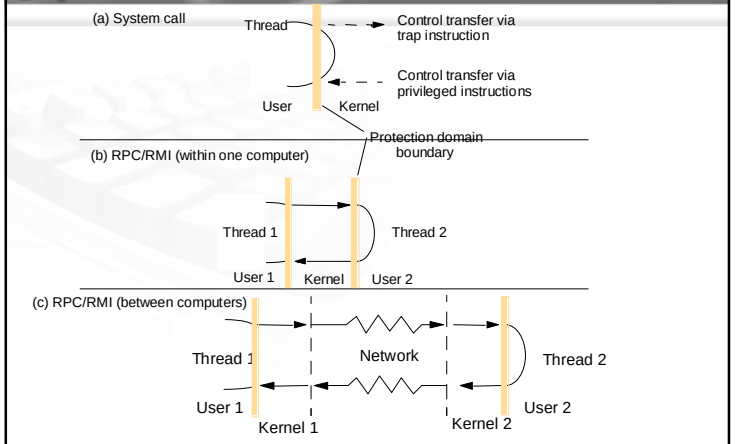
Comunicação em sistemas cliente-servidor

- Sockets
- Remote Method Invocation (RMI)

Cliente/Servidor

- Paradigma para interação:
  - Uma parte do sistema assume papel de executar um serviço (servidor)
  - Outra parte pede a execução de um serviço (cliente)
  - Normalmente muitos clientes para um servidor (compartilha-se/reusa-se o serviço)





Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

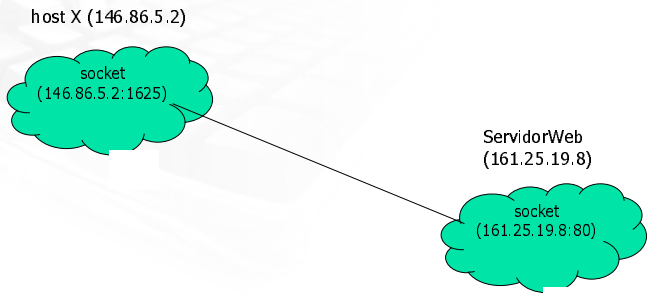
Sockets$_1$

- Definido em cada extremidade para comunicação (um para cada par de processos que se comunicam)
- Socket = IP + Porta.
- Servidor: espera requisições do cliente (processo servidor fica "ouvindo" em uma porta específica)
- Servidor aceita requisição, aceitando uma conexão com o socket do cliente.

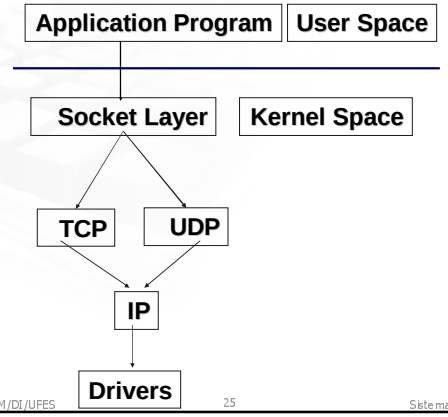
Sockets$_2$

- Para serviços conhecidos, são usadas algumas portas default, e.x., porta 23 para telnet, 21 para ftp e 80 para http
- Cliente: inicia requisição para conexão com servidor. É atribuído uma porta para a conexão do host.
- Se outro processo no mesmo cliente faz requisição, outra porta é atribuída. Isso garante que todas as conexões sejam definidas por um par de sockets único

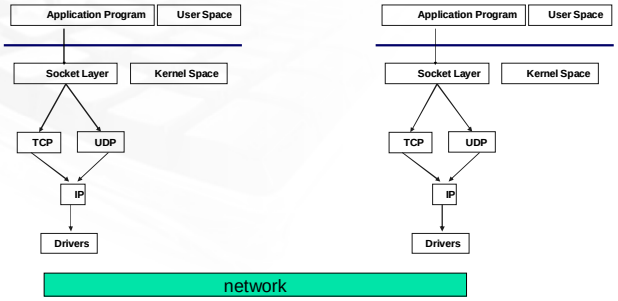
Sockets$_3$



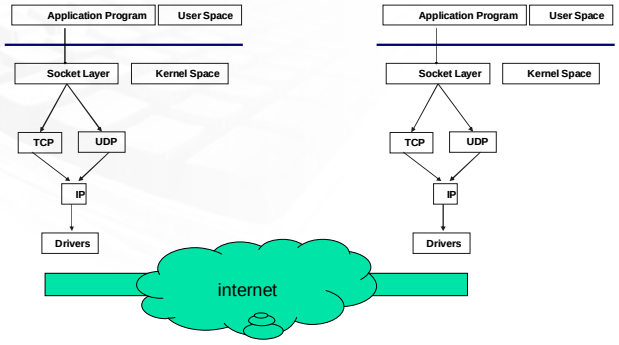
Arquitetura com sockets



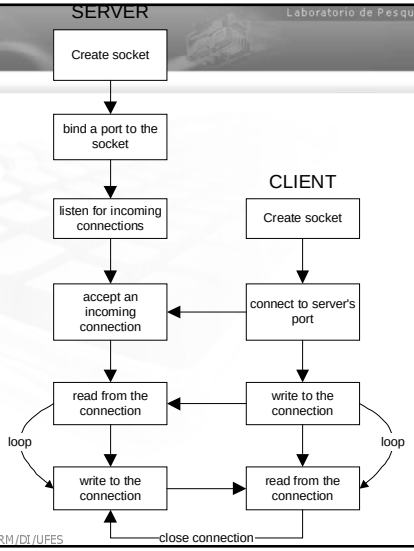
Arquitetura com sockets



Arquitetura com sockets



Sockets



Sockets: Java (Cliente TCP) (1/2)

```
import java.io.*;
import java.net.*;

class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Sockets: Java (Cliente TCP) (2/2)

```
        BufferedReader inFromServer =
                new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));

        sentence = inFromUser.readLine();      aaaa

        outToServer.writeBytes(sentence + '\n');

        modifiedSentence = inFromServer.readLine();
                                                        AAAA
        System.out.println("FROM SERVER: " +
                                modifiedSentence);

        clientSocket.close();

    }
}
```

Create input stream attached to socket

Send line to server

Read line from server

Sockets: Java (Servidor TCP) (1/2)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
    {
      String clientSentence;
      String capitalizedSentence;

      ServerSocket welcomeSocket = new ServerSocket(6789);

      while(true) {
        Socket connectionSocket = welcomeSocket.accept();

        BufferedReader inFromClient =
          new BufferedReader(new
          InputStreamReader(
                connectionSocket.getInputStream()));
```

port 6789

Create welcoming socket at port 6789

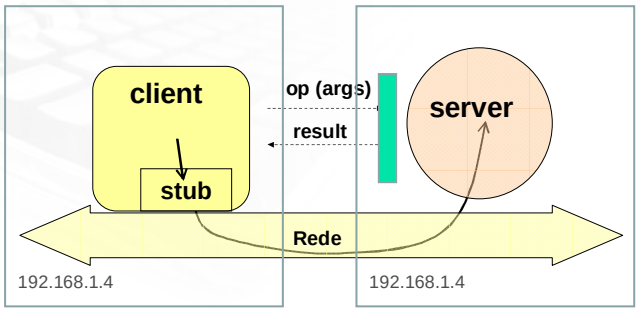
Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Java Remote Method Invocation (RMI)

- Permite que uma thread Java invoque um método em um objeto remoto.
- Objetos são considerados remotos se existirem em JVM diferentes.

Objetos remotos



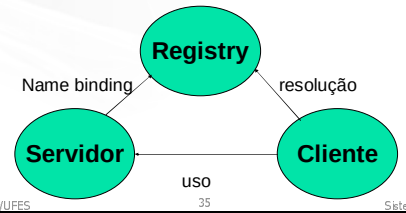
Como programar...

- Para declarar uma interface como remota basta herdar da classe java.rmi.Remote e adicionar uma exceção do tipo RemoteException a todos os seus métodos:
- interface Impressora extends java.rmi.Remote {
  void print (Arquivo arq) throws SemPapel, EnroscouPapel, RemoteException;
  }

http://www.ime.usp.br/~kon/MAC5759/aulas/Aula10.html

Como programar...

- Clientes obtém referências para objetos remotos através do *registry* que é um servidor de nomes.
  - A resolução de nomes é o que produz referências remotas a partir de uma string
  - Os objetos tem que conhecer o *registry*



Semântica das invocações

- Para o cliente, chamadas a métodos locais são iguais a chamadas a métodos remotos. Mas:
  - chamadas locais: passagem de parâmetros por referência
  - chamadas remotas: passagem por valor se objeto convencional (copia o objeto) ou por referência se for objeto remoto
- Classes que são passadas como parâmetro em Java RMI devem implementar java.io.Serializable. Isto permite que a JVM faça a serialização do estado dos objetos de forma que ele possa ser transferido através da rede.
  - Mudanças feitas no objeto do lado do servidor não são refletidas do lado do cliente.

http://www.ime.usp.br/~kon/MAC5759/aulas/Aula10.html

Exemplo: Interface remota (Hello.java)

```
public interface Hello extends java.rmi.Remote {
        String sayHello() throws
   java.rmi.RemoteException;
}
```


Servidor (Server.java)

```
import java.rmi.*;
// ...
public class Server implements Hello {
   public String sayHello() {
        return "Hello, world!";          }
   public static void main(String args[]) {
   try {
      Server obj = new Server();
      Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
      // Bind the remote object's stub in the registry
      Registry registry = LocateRegistry.getRegistry(); // opcional: host
      registry.bind("Hello", stub);
      System.err.println("Server ready");
   } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
   e.printStackTrace();
   }
   }
}
```


Cliente (Client.java)

```
package example.hello;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
    public static void main(String[] args) {
    String host = (args.length < 1) ? null : args[0];
    try {
        Registry registry = LocateRegistry.getRegistry(host);
        Hello stub = (Hello) registry.lookup("Hello");
        String response = stub.sayHello();
        System.out.println("response: " + response);
    } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
    }
}
```


Por debaixo dos panos

- Quando o cliente chama sayHello no "stub" do objeto remoto:
  - O stub do cliente abre uma conexão com a máquina onde está rodando o servidor usando a informação no stub e serializa os parâmetros
  - O stub do lado do servidor aceita a conexão, envia o chamado ao objeto remoto e serializa o resultado (a string "Hello, world!") que é enviado ao cliente
  - O stub do cliente recebe o resultado, deserializa e retorna o resultado para o cliente