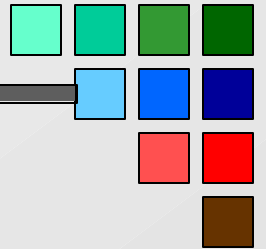


# Sistemas Distribuídos

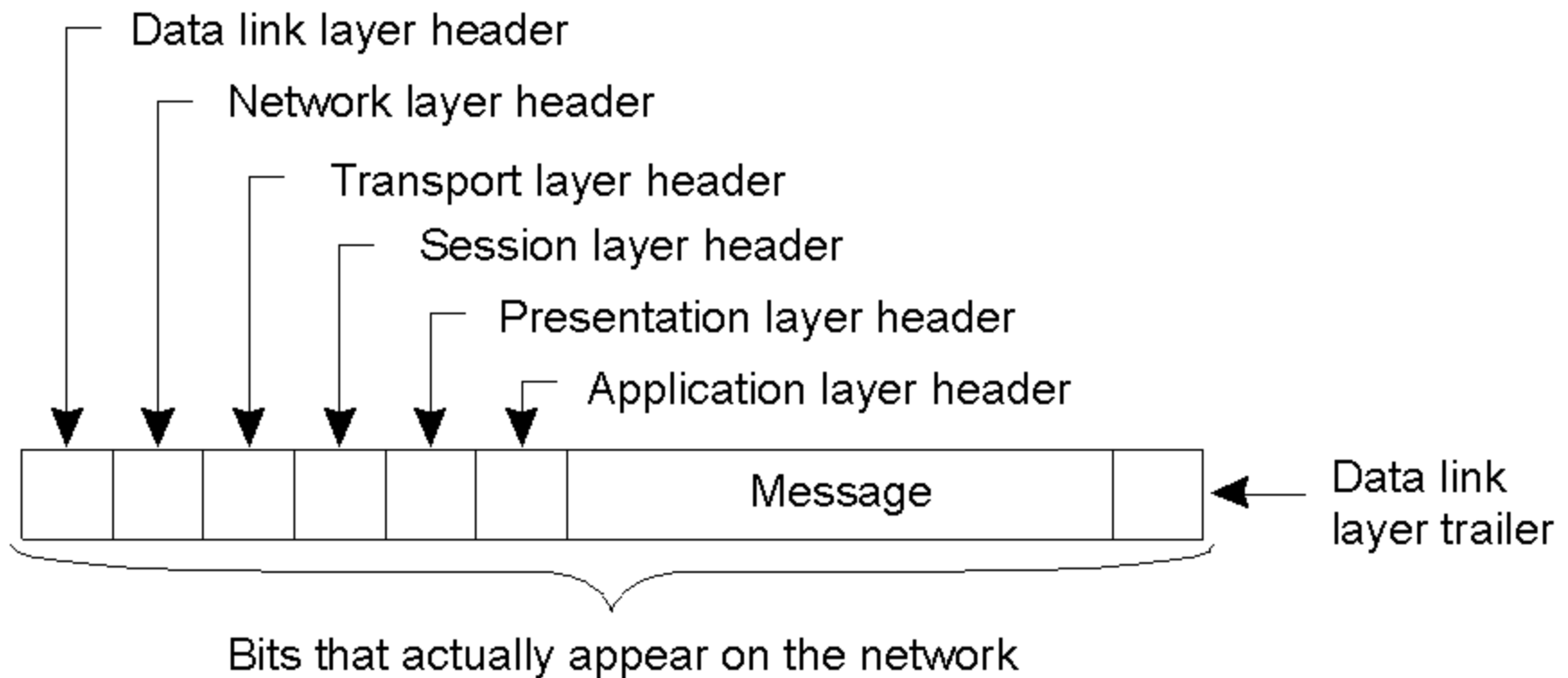
**Prof. Ricardo Ribeiro dos Santos**  
**[ricrs@ec.ucdb.br](mailto:ricrs@ec.ucdb.br)**

**Curso de Engenharia de Computação – UCDB – Julho/2003**

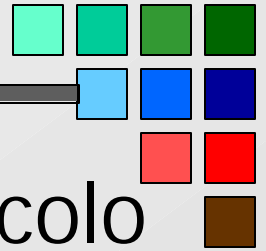
# Relembrando...



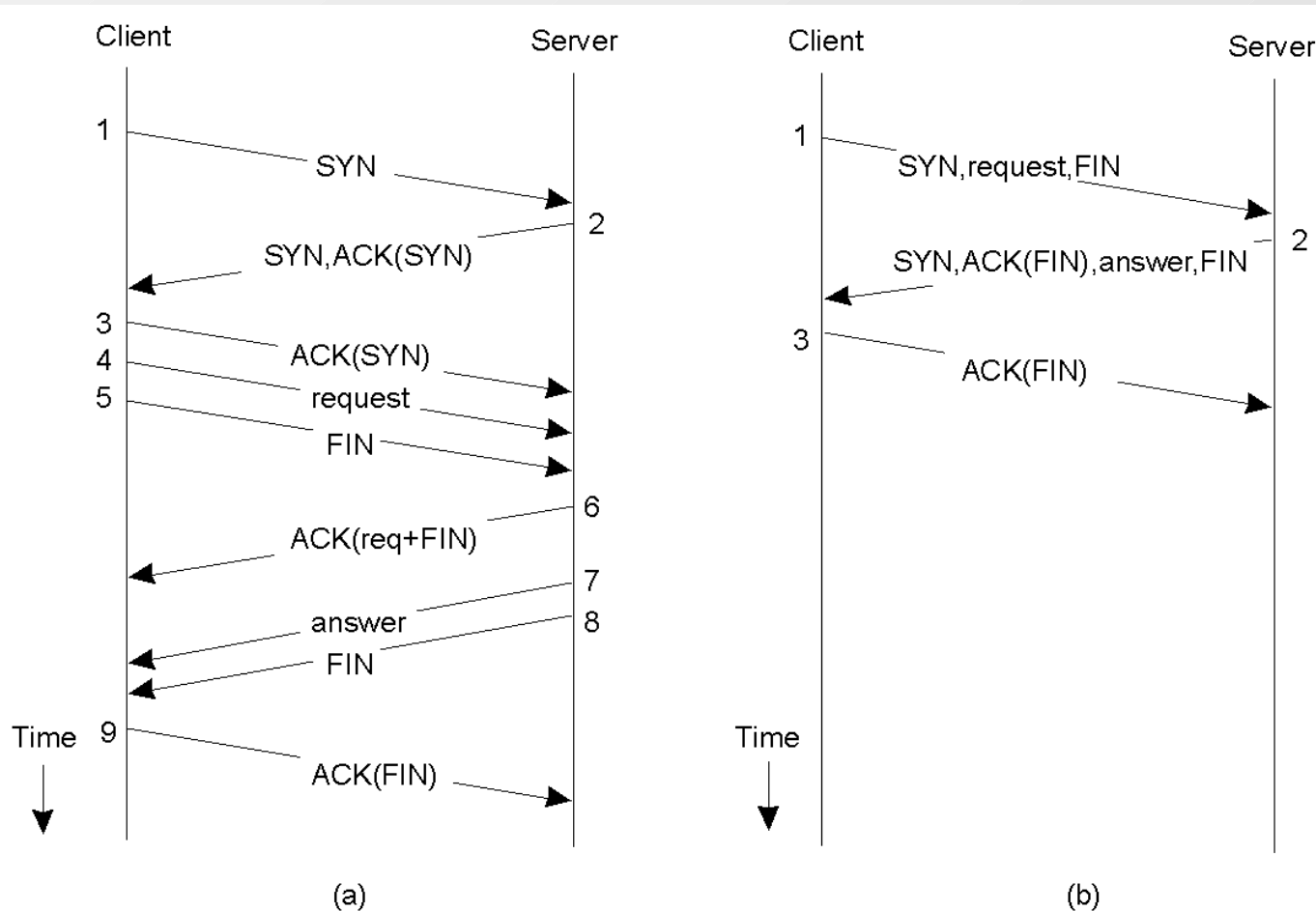
- Encapsulamento nas camadas de rede e protocolos



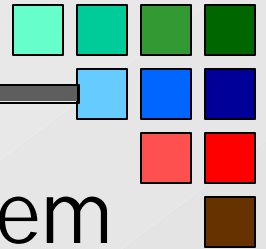
# Relembrando...



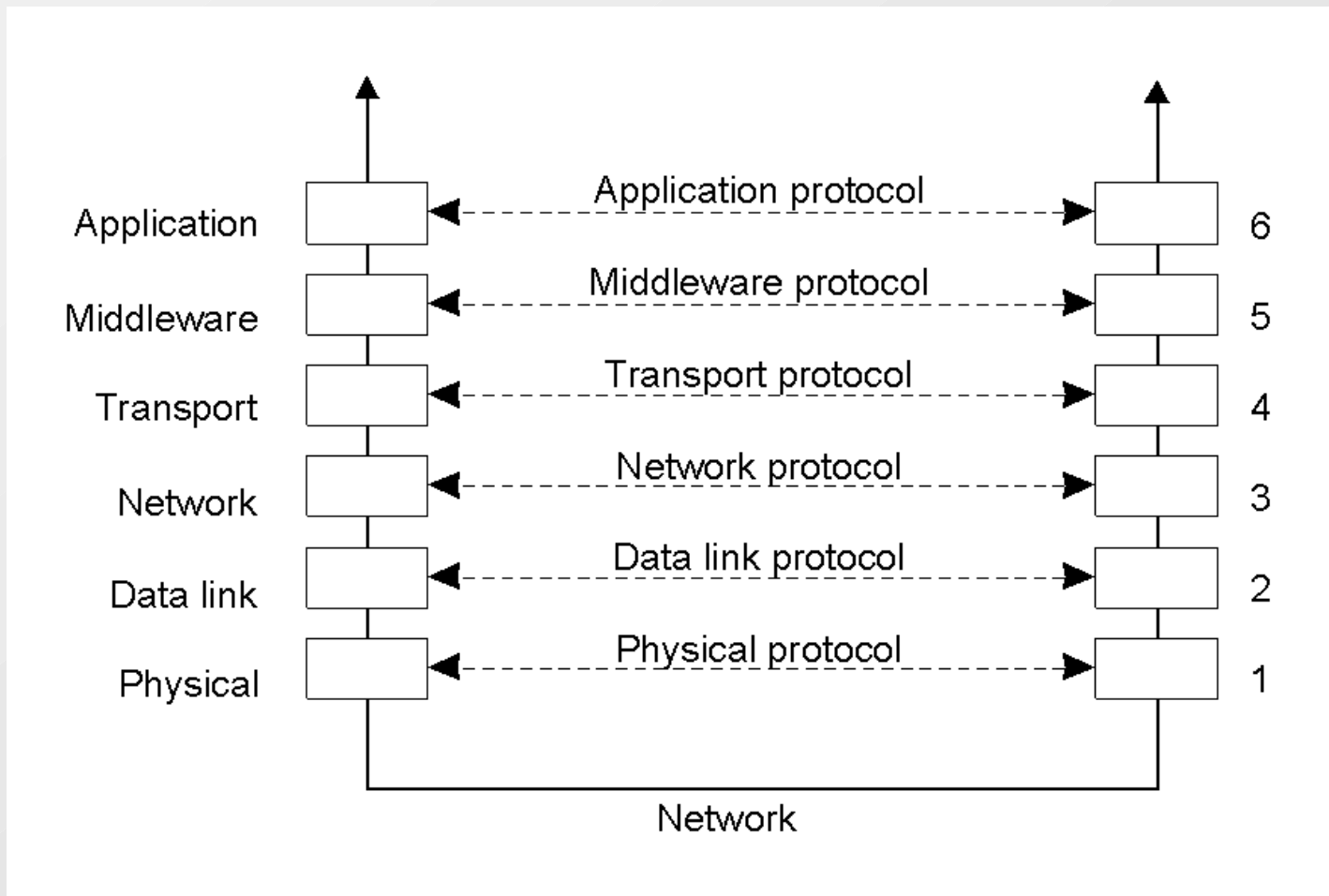
- Iteração entre cliente e servidor no protocolo TCP



# Middleware

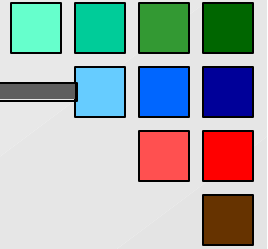


- Camada *middleware* no modelo de Rede em camadas



# Tópicos

---



- Comunicação entre processos
- Comunicação Cliente/Servidor
- Interface *Sockets*
- RPC - Chamadas a procedimentos remotos

# Comunicação entre processos



- Por que é necessária?
  - Troca de informações
  - Sincronismo
  - Cooperação

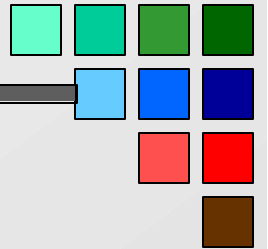
# Comunicação entre processos



- Como mapear estruturas de dados em mensagens?
  - As estruturas de dados a serem enviadas são convertidas antes do envio e reconstruídas após recepção
    - Pode haver conflito na representação de dados
  - Valores são convertidos para uma representação comum, transmitidos e reconstruídos na recepção
  - Alguns exemplos de representação para mensagens trocadas entre aplicações:
    - XDR (*eXtended Data Representation* - Sun), Courier (Xerox), CDR (*Common Data Representation*)

# ***Marshalling/Unmarshalling***

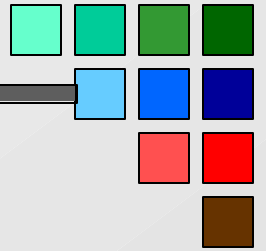
---



- *Marshalling*
  - Processo de montagem dos dados para transmissão
- *Unmarshalling*
  - Processo inverso
- Ambos os processos podem ser realizados explicitamente pelo programa ou gerado automaticamente



# Marshalling/Unmarshalling



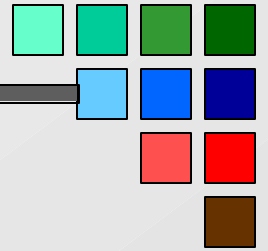
- Exemplo SUN XDR

Mensagem 'Smith', 'London', 1934

→ 4 bytes ®

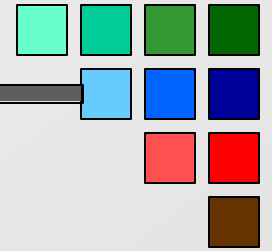
5	← Tamanho da seqüência
"Smit"	← "Smith"
"h_ _ _"	
6	← Tamanho da seqüência
"Lond"	← "London"
"on_ _"	
1934	← Cardinal

# Operações *Send* e *Receive*



- Operações *Send* e *Receive*
  - *Send (dest, message)*: Transmite a mensagem ***message*** para o destino ***dest***
  - *Receive (source, message)*: Recebe a mensagem ***message*** da fonte ***source***
- Comunicação Síncrona e Assíncrona
  - Uso de filas
    - Transmissor insere mensagem
    - Receptor retira mensagem

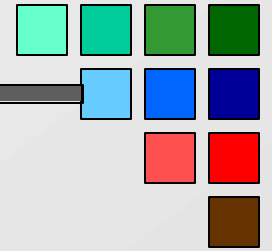
# Comunicação Síncrona e Assíncrona



- Modo síncrono
  - Fonte e destino sincronizam-se em cada mensagem. Operações bloqueantes
- Modo assíncrono
  - Fonte não é bloqueada ao enviar mensagem (uso de *buffer* local)
- Uso de *timeouts*
  - Libera uma operação bloqueante após certo tempo de espera


# Endereço de destino da mensagem

---




- Independente do tipo de comunicação, o endereço de destino deve ser conhecido pelo processo fonte (clientes)
  - Na internet  $\Rightarrow$  n° de uma porta associada ao processo destino + endereço internet da máquina
- Para ser transparente  $\Rightarrow$  serviço de nomes

# Confiabilidade na comunicação entre processos



- Não confiável  $\Rightarrow$  transmissão sem confirmação
  - Por exemplo, UDP
- Processo deve ter o seu mecanismo para checar se a mensagem foi entregue
- Mensagens podem ser:
  - Perdidas
  - Duplicadas
  - Entregues fora de ordem
  - Atrasadas

# Confiabilidade na comunicação entre processos



- No caso de LANs  $\Rightarrow$  confiabilidade é maior, mas ainda podem ocorrer erros
  - Por exemplo, checksum, buffer cheio, etc..
- Serviço confiável pode ser construído sobre um serviço não confiável através de mensagens de confirmação
- No entanto...
  - Quanto mais confiável há mais sobrecarga!

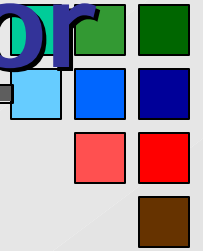
# Comunicação Cliente/Servidor



- **Modelo Cliente-Servidor**

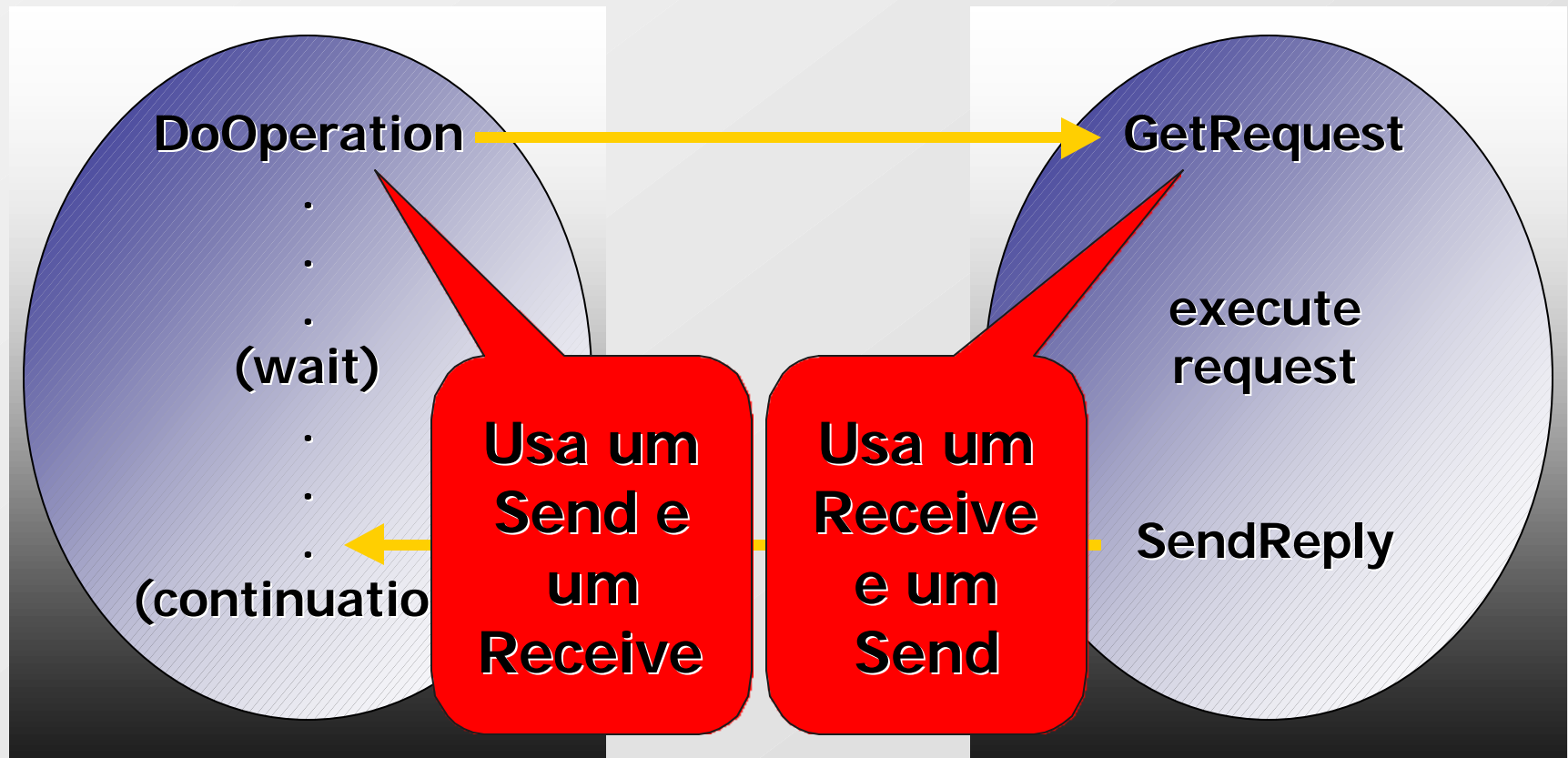
- É um modelo de processamento distribuído sob ponto de vista de aplicação
- O termo **servidor** se aplica a qualquer programa que fornece um serviço
  - Os servidores aceitam solicitações que chegam pela rede,
  - efetuam o serviço, e
  - retornam resultado ao chamador
- O termo **cliente** se refere a um programa que:
  - Envia uma solicitação a um servidor e espera por uma resposta

# Comunicação Cliente/Servidor



Cliente

Servidor





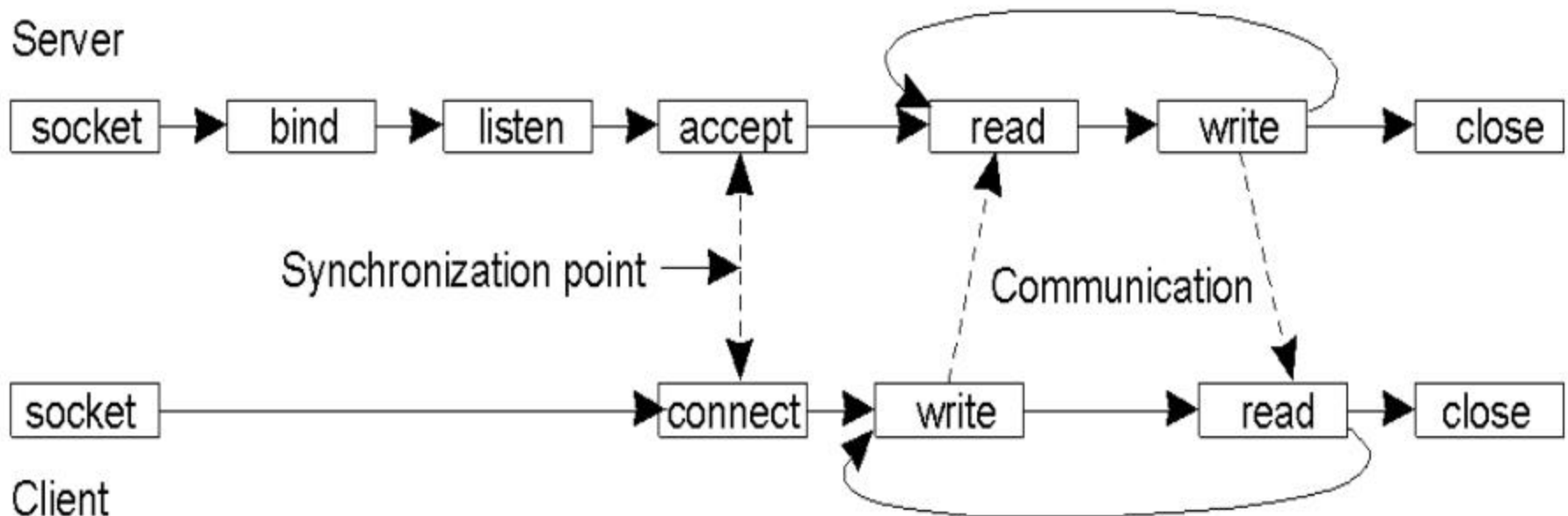
# Comunicação Cliente/Servidor



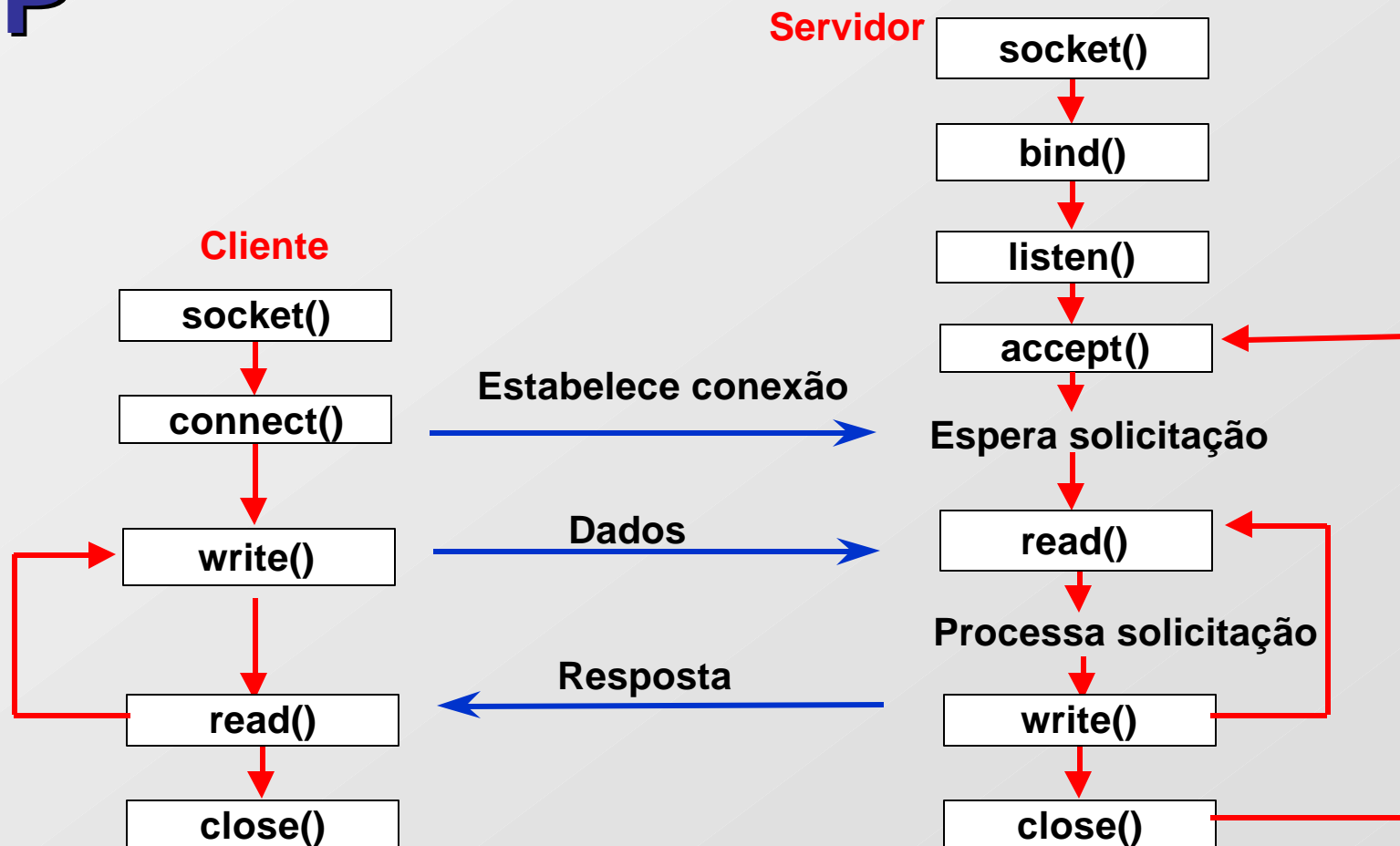
- Possíveis falhas na comunicação
  - Perda da mensagem enviada
  - Perda da resposta enviada
  - Duplicação de mensagem
  - Time-out

# Interface *Socket*

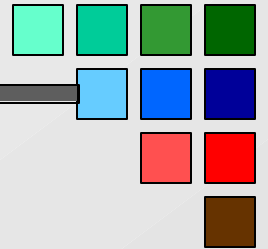
- É uma API que permite a construção de aplicações sobre TCP/IP
- A interface *socket* está baseada no modelo de programação cliente/servidor
- Etapas e funções para comunicação entre processos utilizando *sockets*



# Exemplo de chamadas de sockets num programa usando TCP

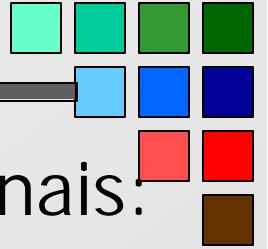


# Interface Sockets



- De maneira resumida, aplicações de rede que utilizam a interface sockets são implementadas na seguinte ordem:
  - Criação do socket → `socket()`
  - Atribuição do endereço local → `bind()`
  - Conexão (caso esteja utilizando um protocolo orientado à conexão como TCP) com o destino → `connect()`
  - Enviar dados → `send()`, `sendto()`, `sendmsg()`, `write()` ou `writenv()`
  - Receber dados → `recvfrom()`, `recvmsg()`, `read()` ou `readv()`
  - Fechar o socket → `close()`

# Interface Sockets

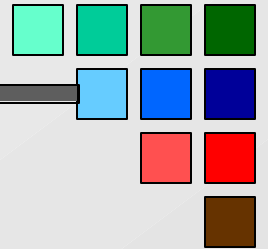


- Existem ainda algumas funções que são opcionais:
  - Obter informações sobre o socket → `getsockopt()`
  - Especificar um tamanho da fila de recepção → `listen()`
  - Obter endereço IP a partir do nome da máquina → `gethostbyname()`
  - Permitir que um servidor manipule serviços concorrentes → `select()`
  - Converter um endereço IP em notação ponto decimal (string) para formato de rede → `inet_addr()`
  - Converter um endereço IP no formato de rede para o formato ponto decimal (string) → `inet_ntoa()`

# Exemplo (em Java) de um cliente *socket* utilizando o protocolo UDP

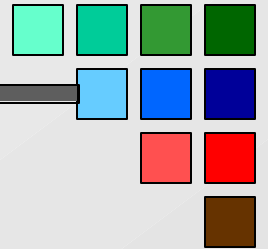
```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int srvPort = 6789;
            DatagramPacket req = new DatagramPacket(m, args[0].length(), aHost, srvPort);
            aSocket.send(req);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

# Exemplo (em Java) de um servidor UDP



```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

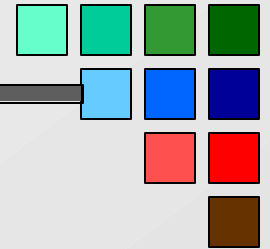
# Exemplo (em Java) de um cliente TCP



```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(InetAddress.getByName(args[0]), serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);
            String data = in.readUTF();
            System.out.println("Received: " + data) ;
        }catch (UnknownHostException e){System.out.println("Sock:" + e.getMessage());}
        }catch (EOFException e){System.out.println("EOF:" + e.getMessage());}
        }catch (IOException e){System.out.println("IO:" + e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException e)
        {System.out.println("close:" + e.getMessage());}}
    }
}
```

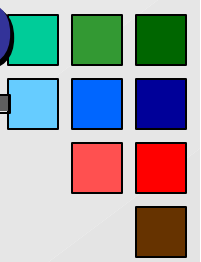


# Exemplo (em Java) de um Servidor TCP



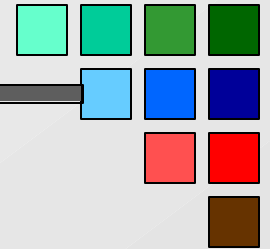
```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            DataOutputStream out;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                out =new DataOutputStream( clientSocket.getOutputStream());
                out.writeUTF("Recebido!");
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}
```

# Algumas dicas para execução dos aplicativos de exemplo...



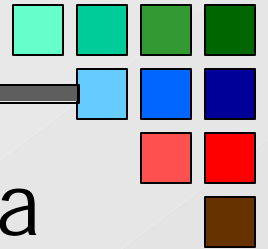
- Arquivos devem conter o nome da classe pública;
- Comando para compilação:
  - **javac <arquivo.java>**
- Comando para execução:
  - **java <arquivo.class>**
- Não esquecer que Java também é sensível às letras maiúsculas/minúsculas
- Sugestão de leitura (Java)
  - DEITEL, H. M., DEITEL, P. J. **Java - Como Programar**. 3ª edição. 2000, Bookman.

# Exercícios Sockets



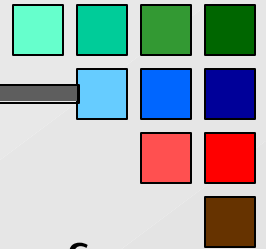
- Implementar exemplos de Sockets TCP e UDP
  - DEITEL, H. M., DEITEL, P. J. **Java - Como Programar**. 3ª edição. 2000, Bookman.
  - Figs. 21.3, 21.4 21.5 e 21.6
- Indicar e Comentar sobre objetos e métodos para:
  - Criação de sockets
  - Estabelecimento de conexão (sockets TCP)
  - Envio/Recebimento de mensagens
  - Fechamento de sockets
- Em duplas
- Entrega: 08/08
  - Documento impresso
    - Programas comentados
    - Programas fontes

# RPC



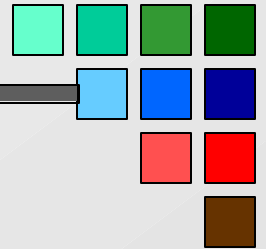
- RPC = *Remote Procedure Call*. Chamada a procedimentos remotos
- De forma geral, constitui-se em uma abstração de alto nível para o desenvolvimento de aplicações de rede
- Programação em RPC está mais voltada para os objetivos da aplicação e não para os detalhes de comunicação em rede
  - RPC  $\neq$  *Sockets* no que diz respeito ao nível de desenvolvimento

# RPC

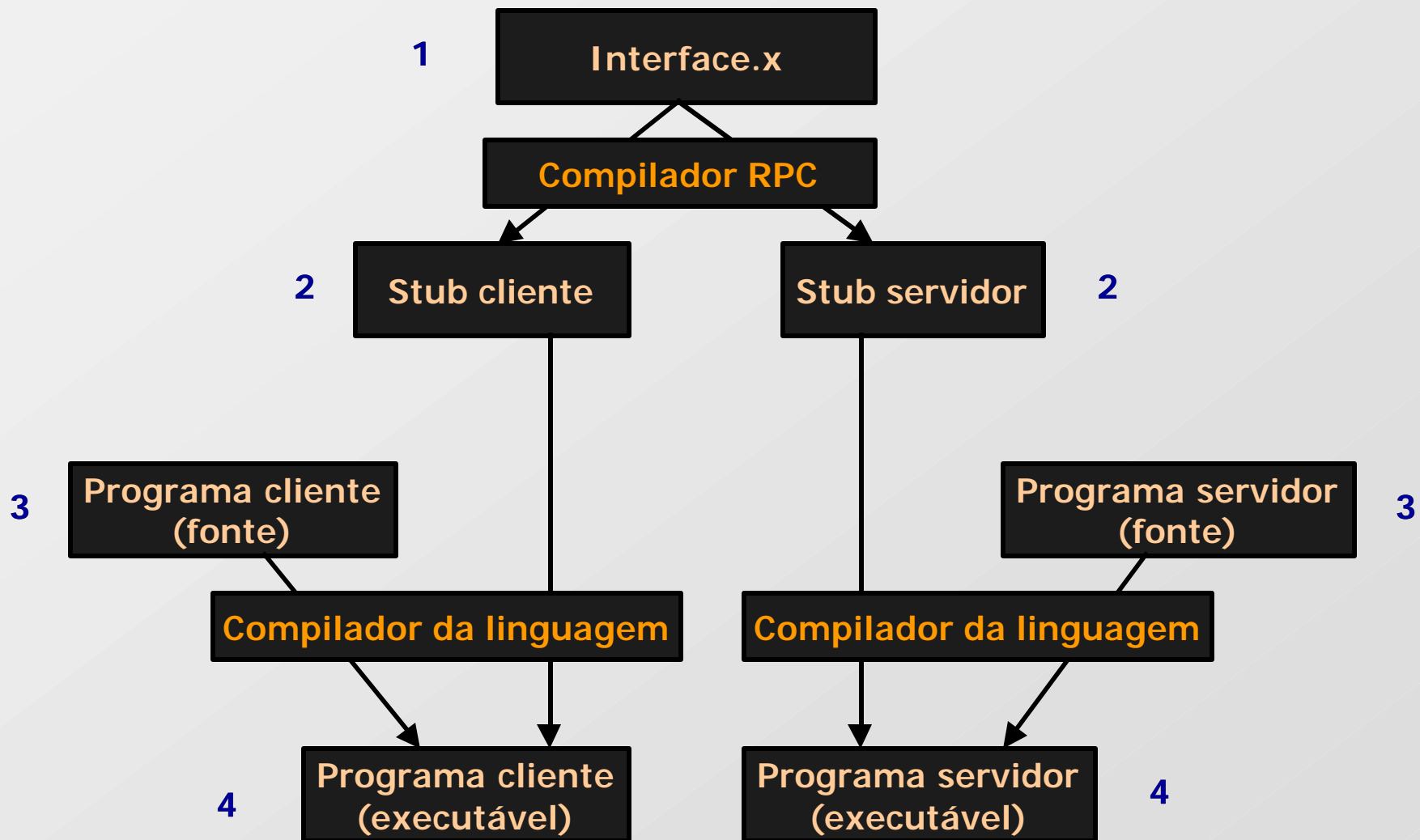


- Etapas de desenvolvimento (resumida):
  - Desenvolver o arquivo com as definições de interfaces
  - Compilar esse arquivo com um compilador RPC
    - Um exemplo de um compilador RPC é o **rpcgen** que geralmente está disponível em muitas distribuições do Linux
  - Desenvolver os programas cliente e servidor
  - Compilar esses arquivos
    - Nessa etapa pode-se utilizar um compilador da linguagem de programação adotada para desenvolver os programas cliente e servidor. No Linux, e utilizando a linguagem C, é possível utilizar o compilador **cc**
    - **Exemplo:**
      - `cc -o arquivoexecutavel arquivofonte.c`

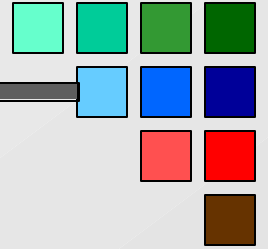
# RPC



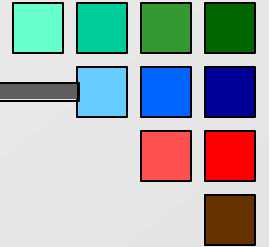
- Etapas de desenvolvimento:



# RPC



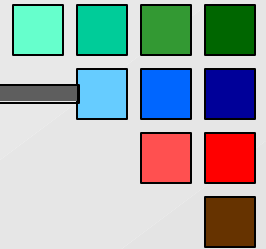
- Arquivo de definição de interfaces
  - Pode conter qualquer extensão
  - Contém as funções e estruturas que devem ser implementadas pelo programa servidor e estarão acessíveis para o(s) cliente(s)
  - Não é utilizado para implementar, apenas para definir!
  - As definições são escritas em uma linguagem neutra (linguagem RPC ou linguagem IDL)



- Arquivo de definição de interfaces – Exemplos
  - Definição de duas funções: bin\_date e str\_date

```
program DATE_PROG {  
    version DATE_VERS {  
        long  BIN_DATE(void) = 1;  
        string STR_DATE(long) = 2;  
    } = 1;  
} = 0x31234567;
```



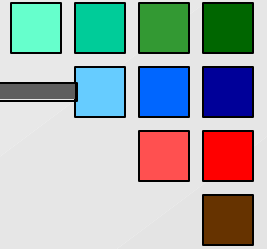


- Arquivo de definição de interfaces – Exemplos
  - Definição de uma função (ordena) com parâmetro de entrada e valor de retorno

```
struct aux{  
    int a[400000];    int esq;    int dir;    int i;    int j;  
};
```

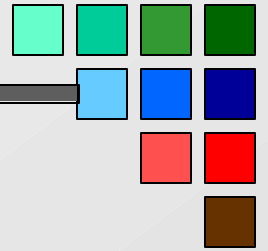
```
struct aux3{  
    int a[400000];  
};
```

```
program STUB_PROG{  
    version STUB_VERS{  
        aux3 ordena(aux)=1;  
    }=1;  
}=0x31234567;
```



- *Stubs*

- São gerados pelo próprio compilador RPC
- Contém os detalhes (conversão de dados, chamada de funções de biblioteca, definição de protocolos) para comunicação entre cliente e servidor
- Geralmente, é gerado um *stub* cliente e um *stub* servidor
- Não necessitam ser editados pelo usuário



- Programa fonte
  - É a implementação do programa cliente e servidor
  - De forma geral:
    - Cliente chama funções no servidor
    - Servidor aceita as requisições do cliente e manipula as referências

# Exemplo de uma chamada cliente/servidor no modelo RPC

