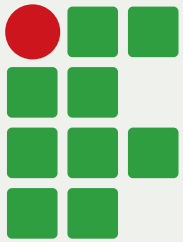
Comunicação

STD29006 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

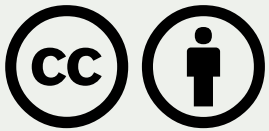
mello@ifsc.edu.br



INSTITUTO FEDERAL
Santa Catarina

Câmpus
São José

Licenciamento



Processos

Processo para o sistema operacional

- Para **cada processo** existe um **espaço de memória reservado**
- **Um processo não pode interferir** no funcionamento de outro processo
- **Escalonamento de processos** é uma atividade crucial de sistemas operacionais modernos
  - Estados: Pronto (*ready*), Em execução (*running*) e Em espera (*waiting*)
  - A **concorrência** entre processos é tratada de forma **transparente para o usuário**
    - chaveamento de contexto: CPU, alocação de segmento de memória, zerar segmento, etc.

Comunicação entre Processos (IPC)

- Arquivos
- Signals
- Pipe
- Sockets

```
echo -e "Fulano\nAna\nPedro" > alunos.txt

cat alunos.txt
```


Comunicação entre Processos (IPC)

- Arquivos
- Signals
- Pipe
- Sockets

```
kill -9 PID
```

- Veja aqui um exemplo em C de como tratar sinais Unix

Comunicação entre Processos (IPC)

- Arquivos
- Signals
- Pipe
- Sockets

```
cat /etc/passwd | grep aluno
```


Comunicação entre Processos (IPC)

- Arquivos
- Signals
- Pipe
- Sockets

- Para listar os sockets unix dos processos em execução

```
ss -lx

# ou com o netstat (obsoleto)

netstat -tln --unix
```


Processo pode ser **composto por diversas Threads**

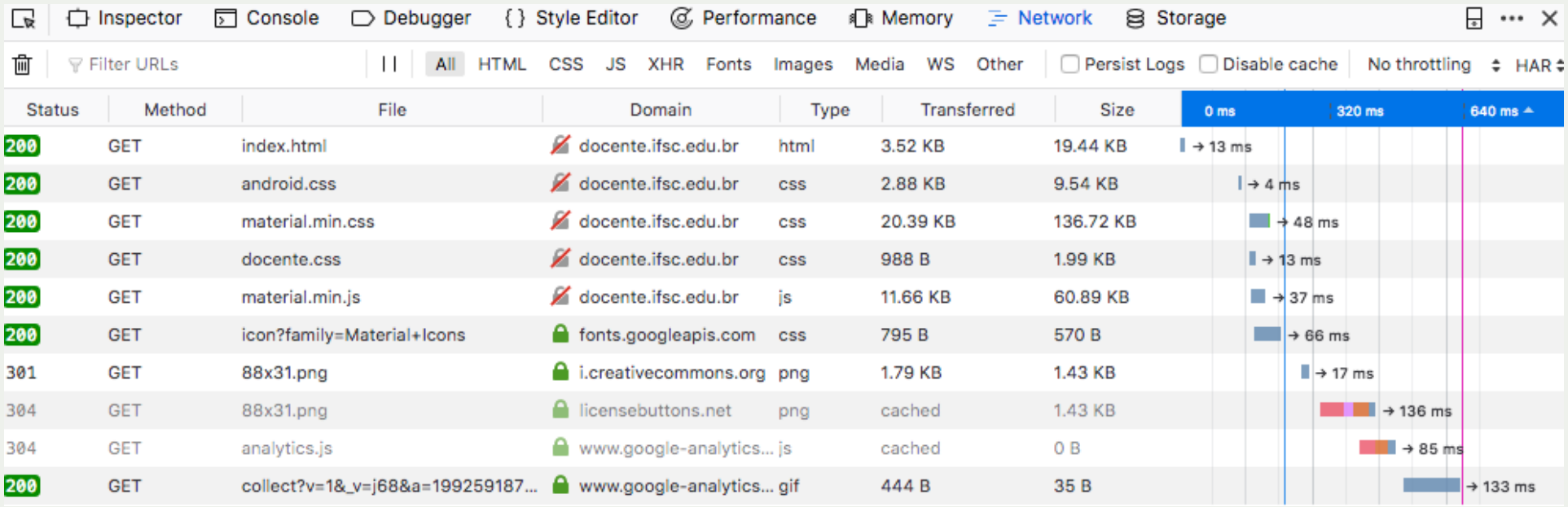
- Cada thread possui um **pedaço independente de código** e não interfere no funcionamento de outras threads
- **Dados de um processo** podem ser **compartilhados** facilmente por **todas as threads**

- **Concorrência não é transparente** para o desenvolvedor
  - Contexto da thread: ready, running, waiting, blocked

Cliente multithread – Ex: navegador web

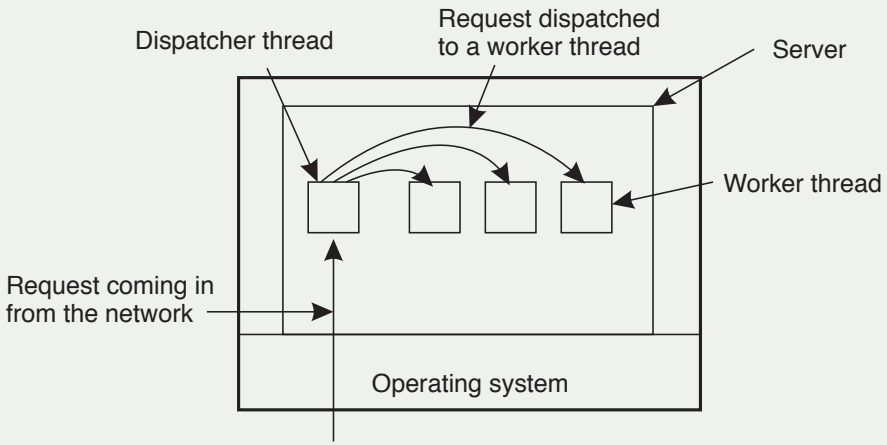
- Documento HTML é composto por texto e uma coleção de mídias
- Navegador web abre uma conexão TCP para obter cada elemento
- Cada elemento é exibido assim que é descarregado

Cliente multithread – Ex: navegador web



Servidor multithread – Ex: servidor de arquivos

- Operações de I/O são chamadas de sistema bloqueantes
- **Comportamento padrão do servidor de arquivos**
  1. Aguardar por pedido relacionado com uma operação com arquivos
  2. Processa o pedido e envia a resposta



- Servidor possui uma thread para esperar pedidos de clientes
  - Dispara uma thread para atender cada cliente

Aguarda conexões e dispara Threads

```
public class Servidor{
   public static void main(String args[]){
      ServerSocket servidor = new ServerSocket(1234);
      while(true){
         Socket conexao = servidor.accept();
         Thread t = new ServidorThread(conexao);
         t.start();
      }
   }
}
```

```
public class ServidorThread extends Thread{
   private Socket conexao;
   public ServidorThread(Socket c){
      this.conexao = c;
   }
   public void run(){
      System.out.println("Thread executada");
   }
}
```


Distribuição de tarefas

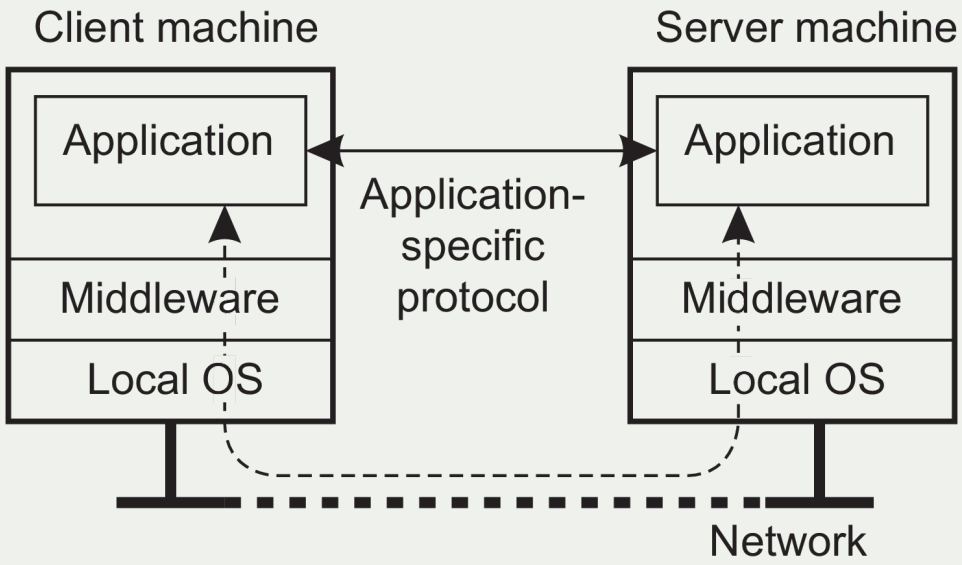
- **Aplicação multithread**
  - *Thread* principal aguarda por conexões dos clientes
  - Nova *thread* é disparada para atender cada cliente
- **Aplicação distribuída**
  - **Processo coordenador** (*main*)
    - Responsável por distribuir tarefas, coordenar e compilar as respostas dos trabalhadores
  - **Processos trabalhadores** (*workers*)
    - Responsáveis por processar tarefas

Comunicação em Sistemas Distribuídos

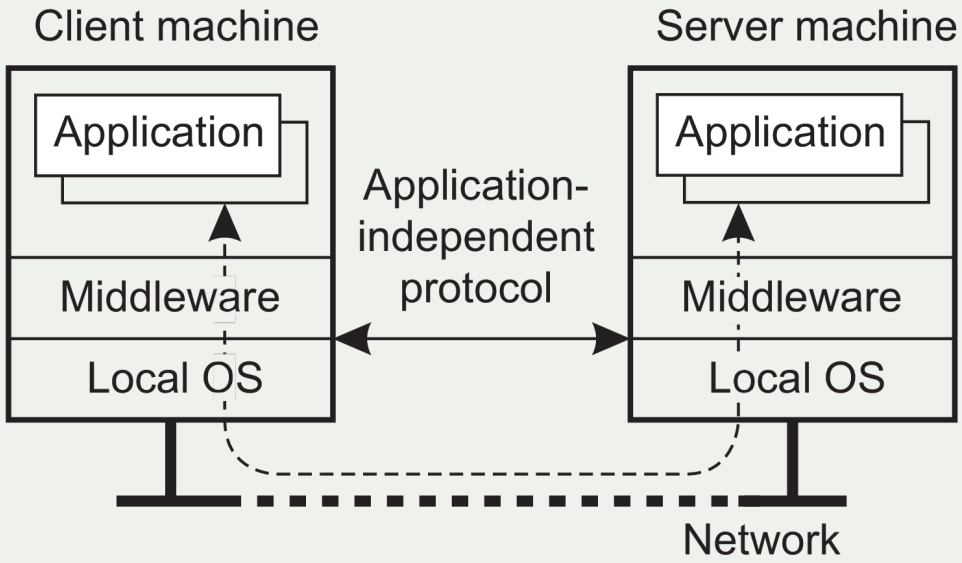
Comunicação entre processos é a essência de SD, pois processos são executados em diferentes máquinas

- **Protocolos de rede organizados em camadas**
  - Flexibilidade para o desenvolvimento de softwares
  - Softwares da camada de aplicação não precisam ter ciência sobre o formato de um quadro ethernet

Comunicação entre cliente e servidor: interface de usuário



- Aplicação faz uso de protocolo próprio para comunicação
- Ex: Aplicativo cliente local sincroniza com servidor remoto

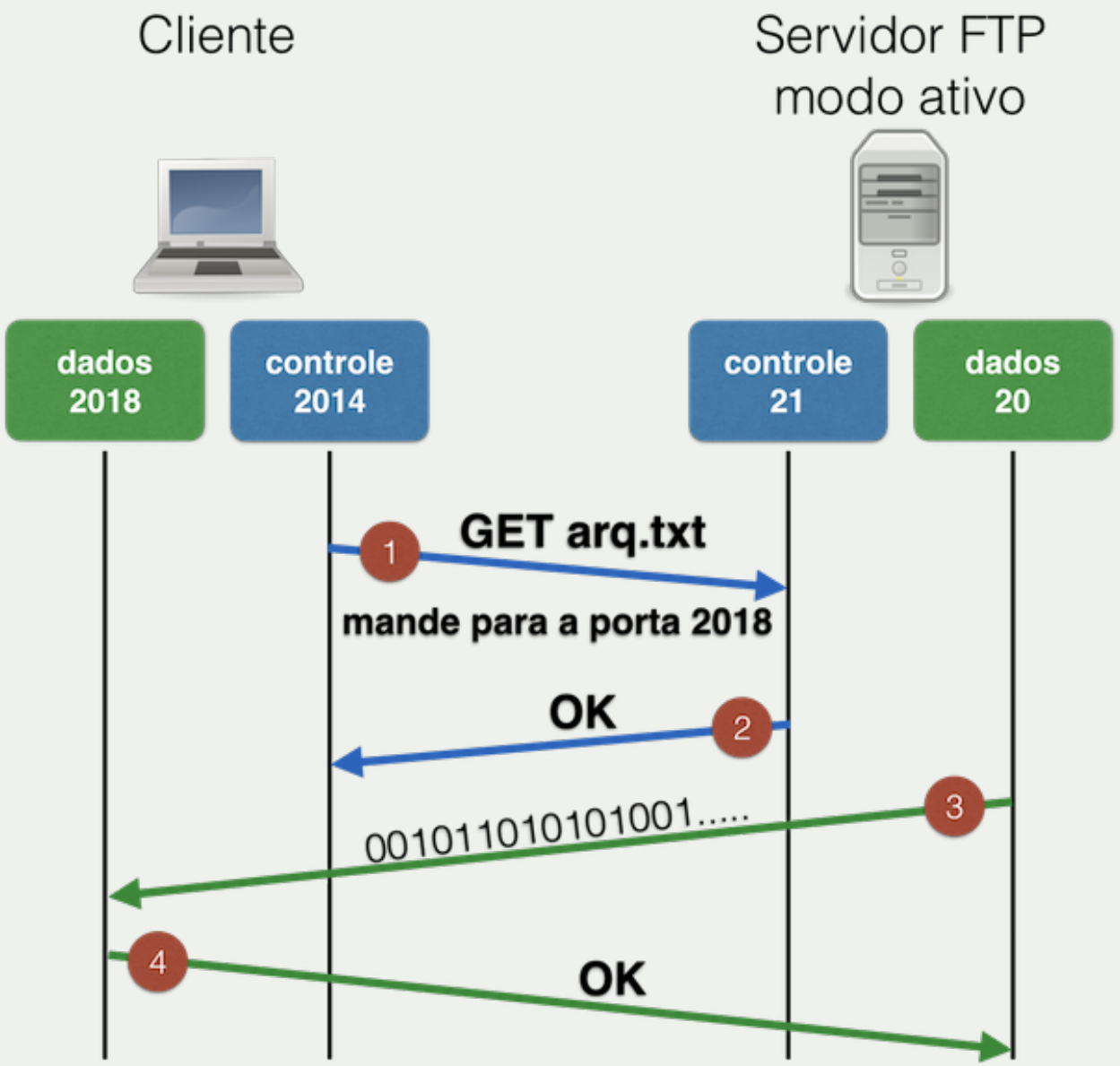


- Cliente leve usado somente como terminal para o usuário
- Armazenamento, processamento são executados no servidor

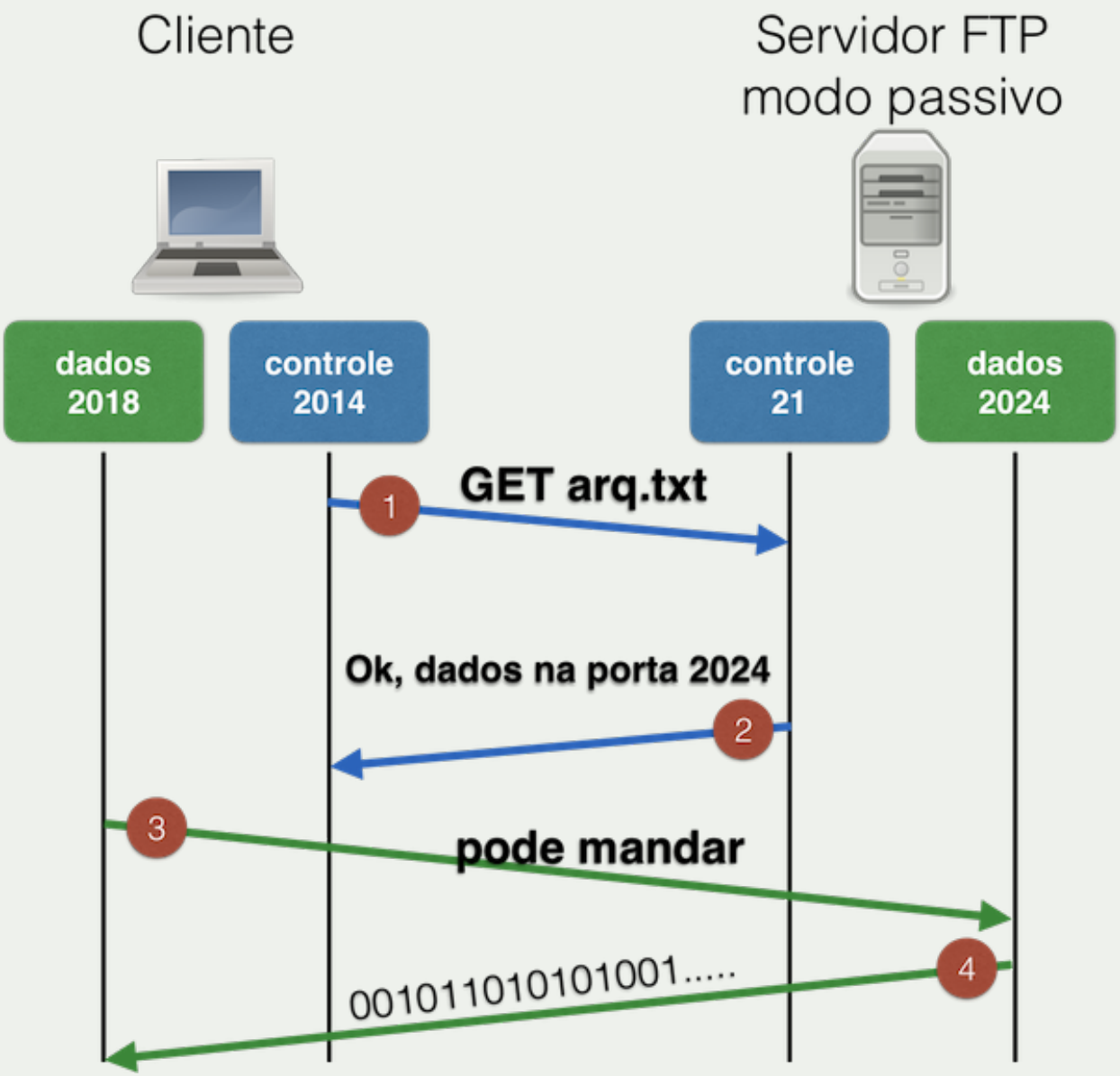
Contatando um aplicativo servidor (*endpoint*)

- Um aplicativo **servidor** está
  - associado a um endereço IP e porta (ex: 127.0.0.1:8000)
  - sobre um protocolo de transporte (ex: TCP ou UDP)
- O **cliente precisa conhecer** essas informações **previamente** para que possa conectar no servidor
- Um serviço pode estar associado a qualquer porta livre, porém é interessante evitar as portas padronizadas
- Alguns serviços já possuem portas padronizadas pela IANA
  - 80 HTTP/TCP, 21 FTP/TCP, 25/TCP SMTP
  - `cat /etc/services`

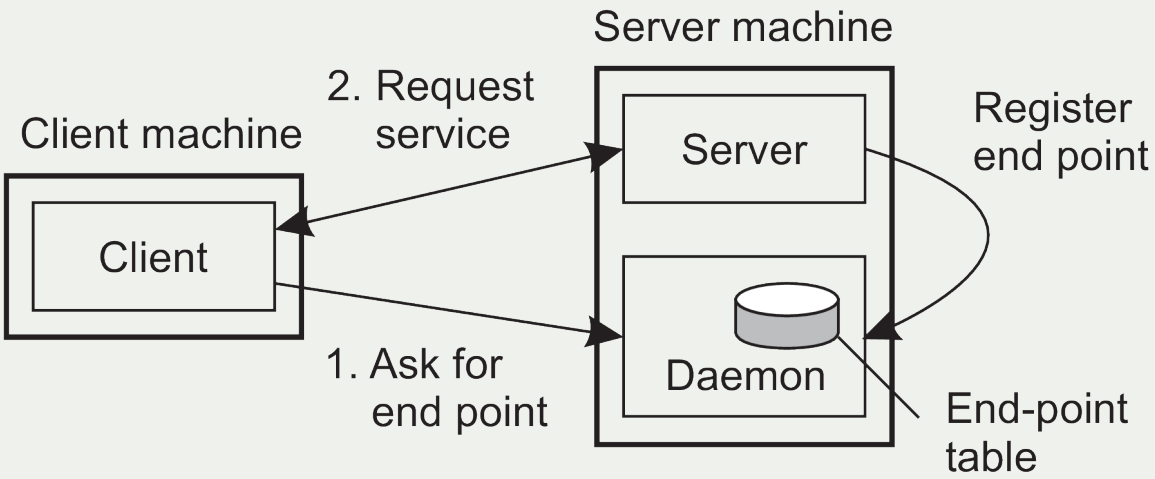
Exemplo FTP: modo ativo e passivo

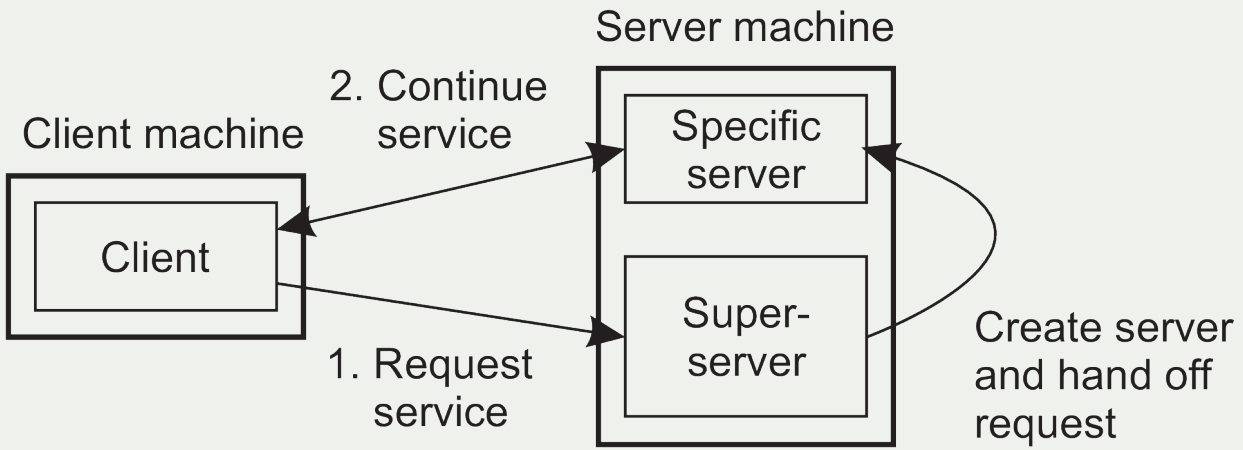


Exemplo FTP: modo ativo e passivo



Atribuição dinâmica de *endpoint*





Comunicação fora do fluxo principal (*out-of-band*)

Como interromper um servidor uma vez que ele aceitou a comunicação e está transferindo dados?

Comunicação fora do fluxo principal (*out-of-band*)

Como interromper um servidor uma vez que ele aceitou a comunicação e está transferindo dados?

1. Thread/processo ouvindo em uma outra porta para receber dados de controle

Comunicação fora do fluxo principal (*out-of-band*)

Como interromper um servidor uma vez que ele aceitou a comunicação e está transferindo dados?

1. Thread/processo ouvindo em uma outra porta para receber dados de controle
2. Facilidades da camada de transporte (não muito usual)
   - TCP permite o envio de mensagens urgentes dentro de uma mesma conexão
   - Mensagens urgentes podem ser capturadas fazendo uso de SIGNALs do S.O.

Servidores e manutenção do estado

- **Servidor que não mantém estado (*stateless server*)**
  - Não mantém qualquer informação sobre o cliente após atendê-lo e fechar a conexão
  - Clientes e servidores são independentes, minimizando problemas de inconsistência de estado diante de uma falha do servidor
  - Pode ter o desempenho prejudicado, uma vez que o servidor não consegue correlacionar pedidos subsequentes

Servidores e manutenção do estado

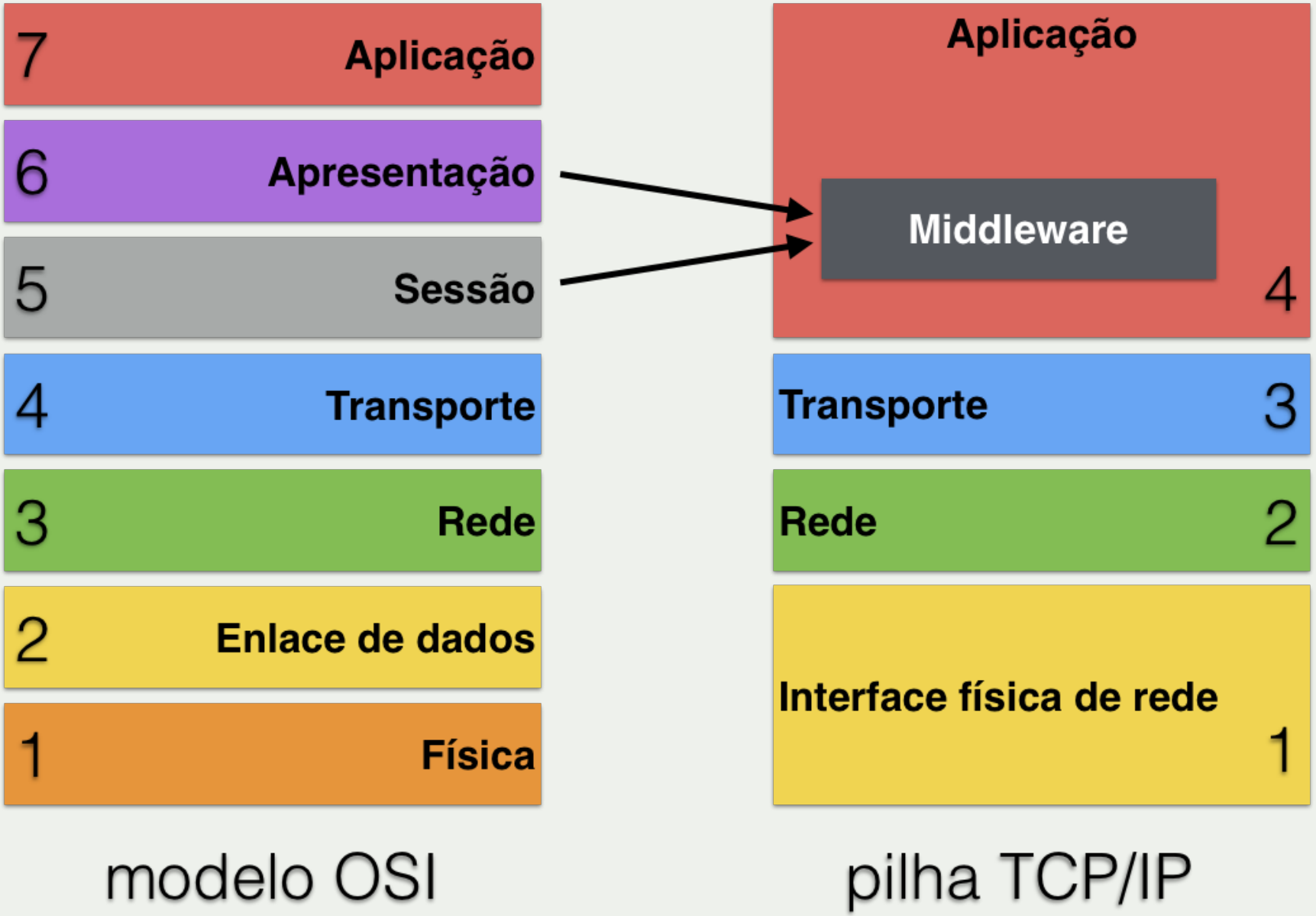
- **Servidor que não mantém estado (*stateless server*)**
  - Não mantém qualquer informação sobre o cliente após atendê-lo e fechar a conexão
  - Clientes e servidores são independentes, minimizando problemas de inconsistência de estado diante de uma falha do servidor
  - Pode ter o desempenho prejudicado, uma vez que o servidor não consegue correlacionar pedidos subsequentes
- **Servidor que mantém estado (*stateful server*)**
  - Mantém informações sobre a interação com o cliente
  - Ex: Sabe quais arquivos foram abertos, sabe quais dados o cliente já possui em *cache*
  - Oferece um ótimo desempenho, uma vez que o cliente pode manter cópias locais das informações, contudo a confiabilidade é sua maior dificuldade (inconsistência)

Middleware

Aplicação que provê um **conjunto de protocolos** de comunicação atuando como mediador entre processos clientes e servidores

- **Protocolos de alto nível** independentes de aplicações
- Provê suporte para transações, sincronização, protocolos de autenticação, etc

Middleware


7 Aplicação
6 Apresentação
5 Sessão
4 Transporte
3 Rede
2 Enlace de dados
1 Física
Aplicação
Middleware
4
Transporte 3
Rede 2
Interface física de rede 1
modelo OSI
pilha TCP/IP

Middleware – Tipos de comunicação

- **Persistência**
- **Sincronismo**
- **Fluxo**

Middleware – Tipos de comunicação

- **Persistência**
  - **Persistente** – mensagem fica armazenada no middleware o tempo que for necessário até ser entregue para o receptor – ex: e-mail
  - **Transitória** – mensagem fica armazenada no middleware apenas enquanto emissor e receptor estiverem ativos – ex: tcp/udp
- **Sincronismo**
- **Fluxo**

Middleware – Tipos de comunicação

- **Persistência**
  - **Persistente** – mensagem fica armazenada no middleware o tempo que for necessário até ser entregue para o receptor – ex: e-mail
  - **Transitória** – mensagem fica armazenada no middleware apenas enquanto emissor e receptor estiverem ativos – ex: tcp/udp
- **Sincronismo**
  - **Síncrono** – emissor fica bloqueado esperando resposta
  - **Assíncrono** – não fica bloqueado e recebe uma notificação quando a resposta estiver disponível
- **Fluxo**

Middleware – Tipos de comunicação

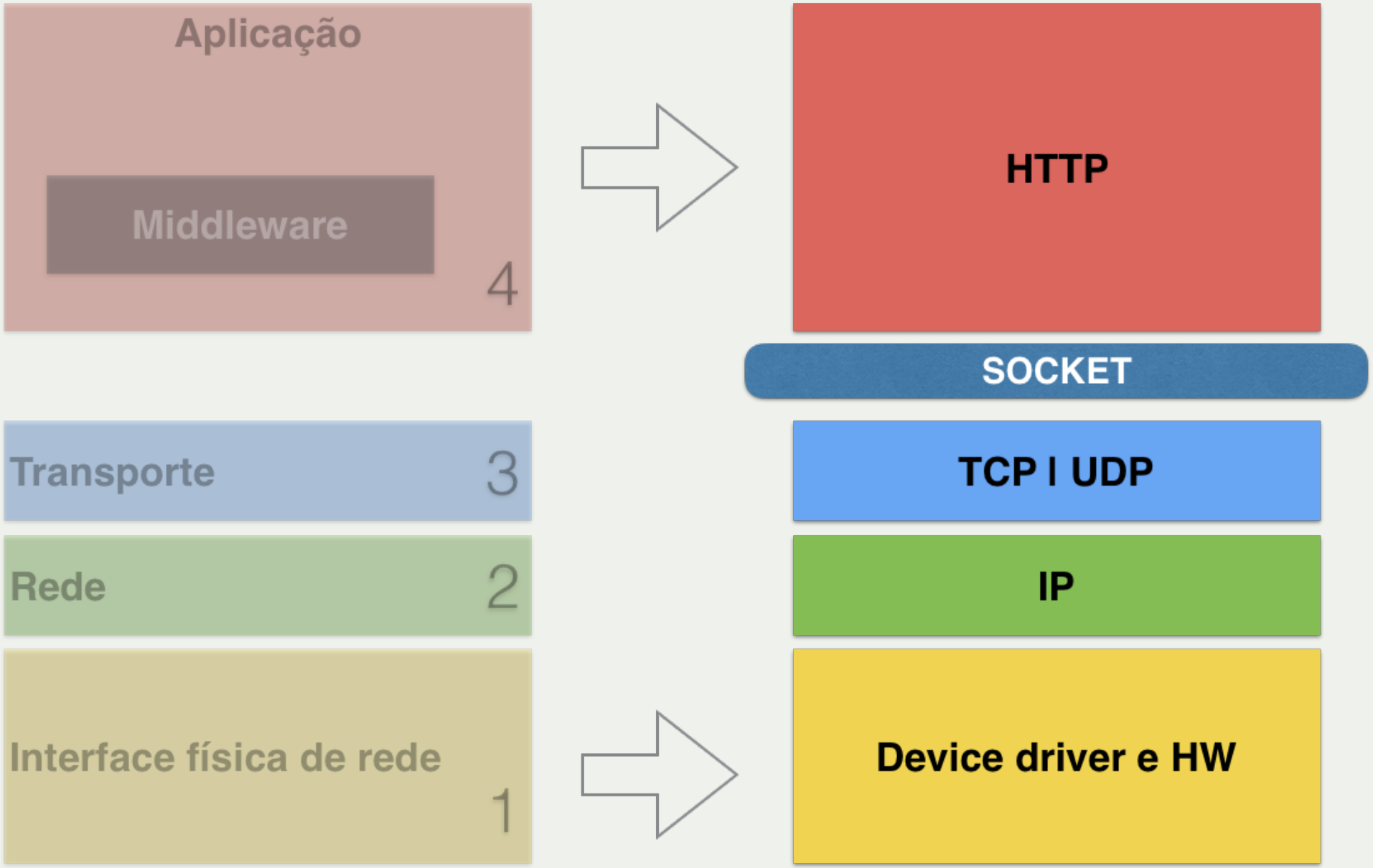
- **Persistência**
  - **Persistente** – mensagem fica armazenada no middleware o tempo que for necessário até ser entregue para o receptor – ex: e-mail
  - **Transitória** – mensagem fica armazenada no middleware apenas enquanto emissor e receptor estiverem ativos – ex: tcp/udp
- **Sincronismo**
  - **Síncrono** – emissor fica bloqueado esperando resposta
  - **Assíncrono** – não fica bloqueado e recebe uma notificação quando a resposta estiver disponível
- **Fluxo**
  - **Discreto** – partes trocam mensagens, sendo cada mensagem tratada como uma unidade completa de informação – ex: navegação web
  - **Fluxo contínuo** – são trocadas diversas mensagens consecutivas e estão relacionadas entre si – ex: rádio pela Internet

Modelo cliente/servidor

- Geralmente baseada no modelo de **comunicação síncrono** e com **persistência transitória**
- Cliente e servidor precisam estar ativos ao mesmo tempo e cliente fica bloqueado até receber a resposta

Sockets

Sockets


Aplicação
Middleware
4
HTTP
SOCKET
Transporte
3
TCP I UDP
Rede
2
IP
Interface física de rede
1
Device driver e HW

Formas de comunicação entre processos no mesmo S.O.

- Arquivos, Signals, Pipe, etc.

```
ls -lR /etc/ | grep passwd
```


Formas de comunicação entre processos no mesmo S.O.

- Arquivos, Signals, Pipe, etc.

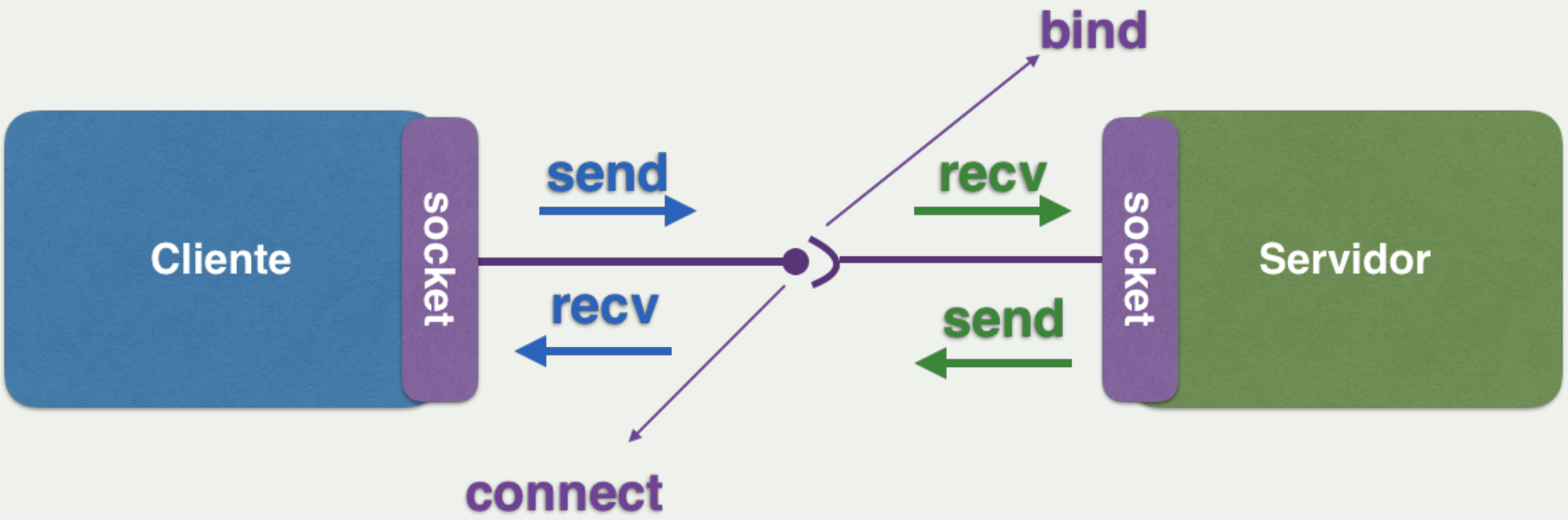
```
ls -lR /etc/ | grep passwd
```

Sockets permite a **comunicação entre processos**, executados em **diferentes máquinas**

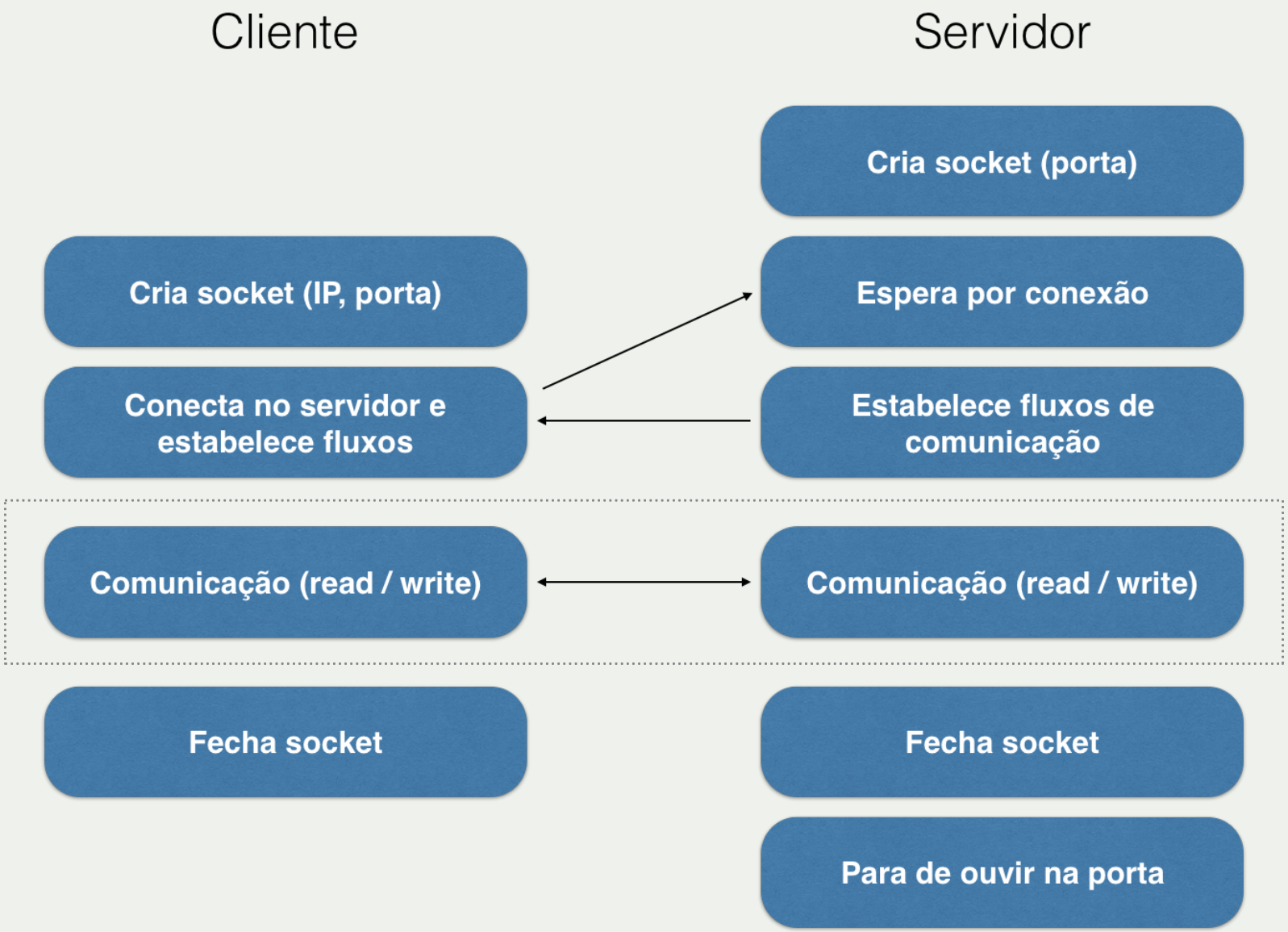
- API em C criada em 1983 no 4.2 BSD UNIX, é padrão em todos S.O.

- Sockets IP são identificados: protocolo de transporte, endereço IP e porta
  - TCP – Orientado a conexão
  - UDP – Orientado a datagramas (sem conexão)

Cliente e Servidor com Sockets



Cliente e Servidor com Sockets TCP



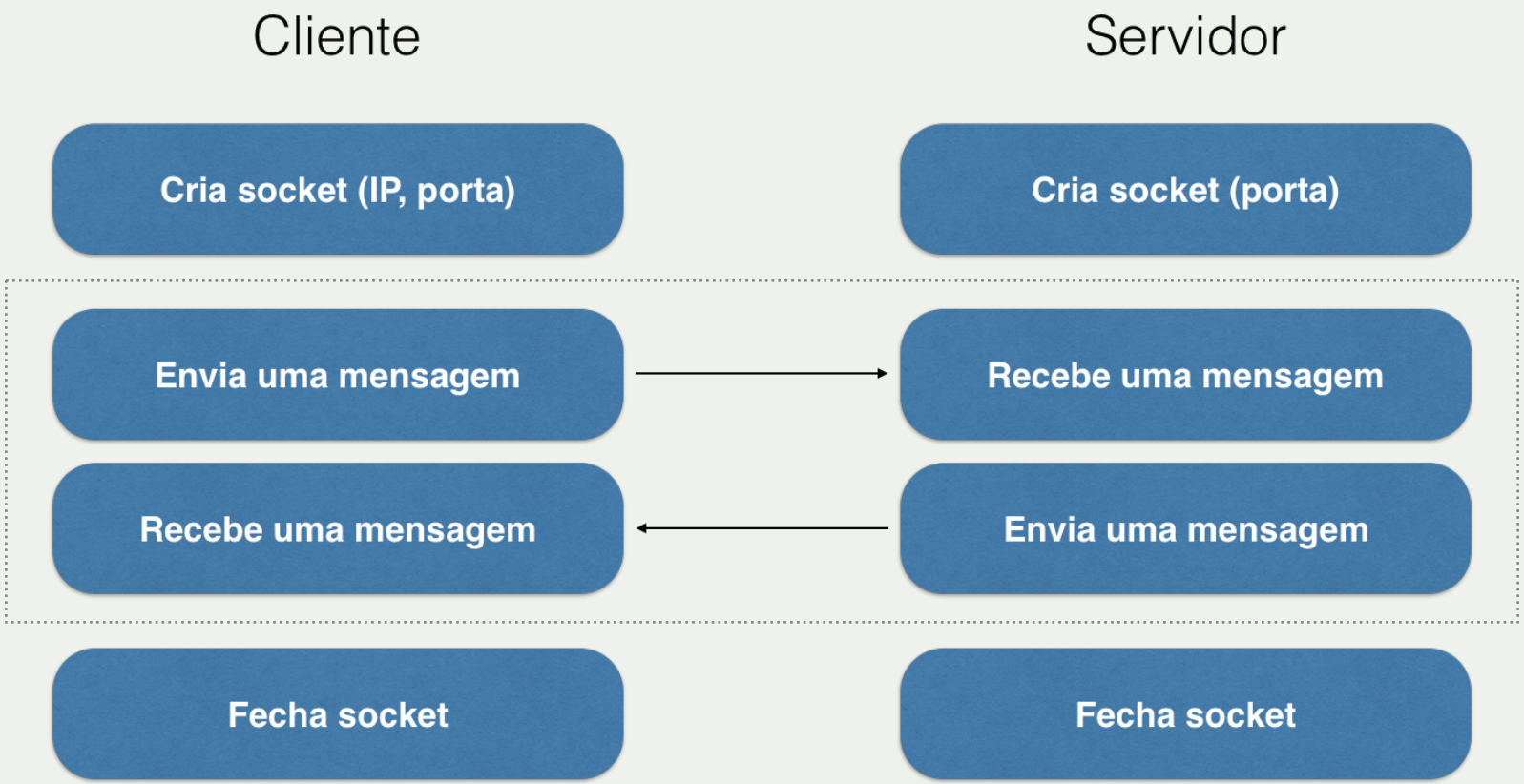
Python 3.6: cliente e Servidor com Sockets TCP

```
import socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((socket.gethostname(), 1234))
    s.listen()
    conexao, addr = s.accept()
    with conexao:
        print(f"Cliente conectado: {addr}")
        while True:
            dados = conexao.recv(1024) # 1024 é a quantidade máxima de dados
            if not dados:
                break
            print(f"Mensagem recebida: {dados.decode()}")
            conexao.sendall(dados)
```

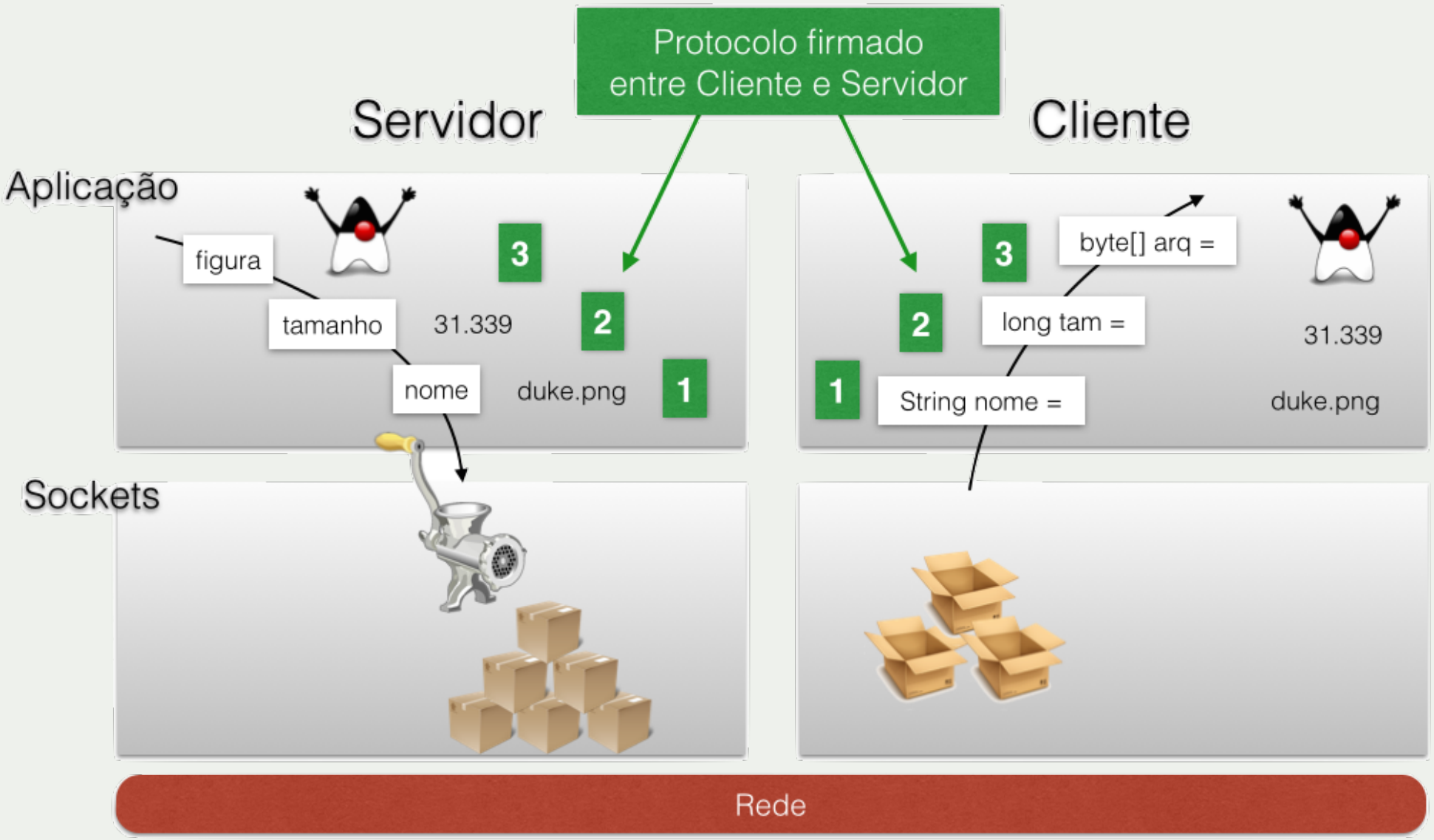
- **Cliente**

```
import socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("127.0.0.1", 1234))
    s.sendall(b"Ola, mundo")
    dados = s.recv(1024)
print(f"Resposta do servidor: {dados.decode()}")
```


Cliente e Servidor com Sockets UDP



Protocolo definido entre Cliente e Servidor



Protocolo definido entre Cliente e Servidor – em Java

Cliente conecta e o servidor envia a primeira mensagem

Servidor

```
OutputStream os = conexao.getOutputStream();
DataOutputStream dout = new DataOutputStream(os);

dout.writeUTF("duke.png"); // envio da mensagem 1
dout.writeLong(31.339);    // envio da mensagem 2
dout.write(blocos);        // envio da mensagem 3
```

Cliente

```
Socket conexao = new Socket("127.0.0.1", 1234);
InputStream is = conexao.getInputStream();
DataInputStream dis = new DataInputStream(is);

String nome = dis.readUTF();                    // mensagem 1
long tamanhoArq = dis.readLong();               // mensagem 2
byte[] arq = dis.read(blocos, 0, tamanhoArq); // mensagem 3
```


Transmissão de dados pela rede

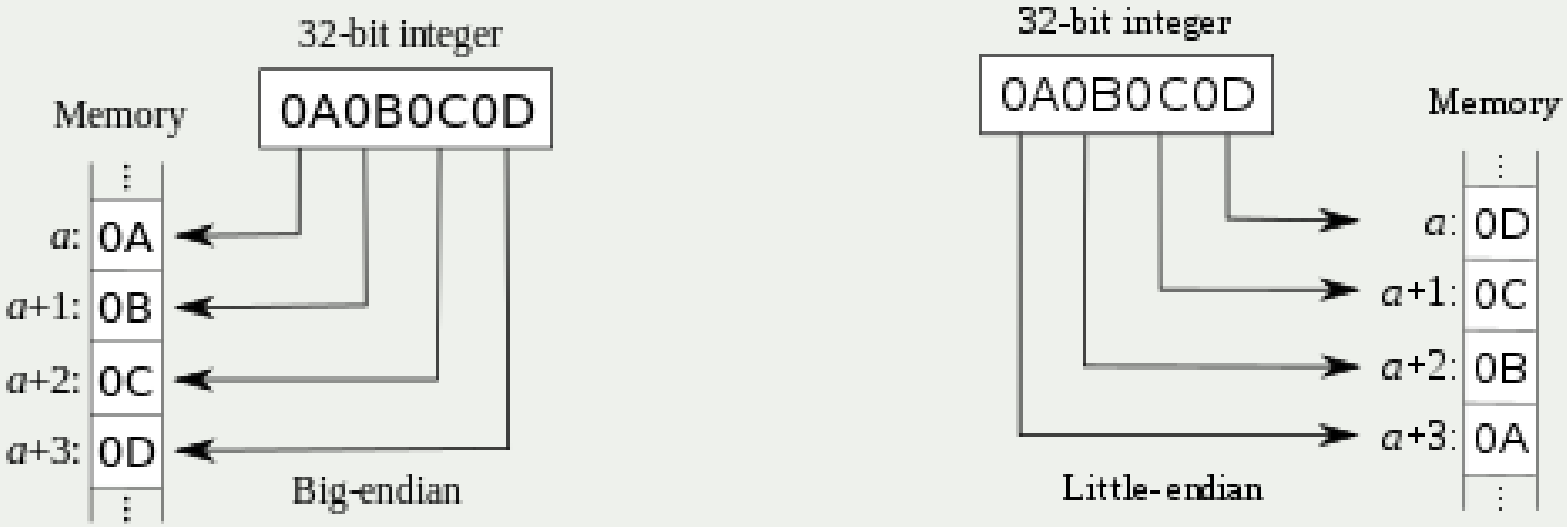
Transmissão de dados pela rede

- Transmissão de dados pela rede requer um acordo prévio entre cliente e servidor para que ambos possam **representar os dados corretamente** em seus ambientes
  - Mensagens são transmitidas como **fluxos de bytes**
- Máquinas distintas podem ter diferença
  - Na ordenação de bytes
  - Na quantidade de bytes para representar inteiros
  - Na representação de valores reais
  - Na codificação de caracteres (i.e. ASCII *vs* UNICODE)

Heterogeneidade de representação de dados

- Os **processos cliente** e **servidor** podem ser executados em **diferentes arquiteturas** de máquinas (i.e. Intel *versus* PowerPC)
- A **extremidade** (*endianness*) se refere à ordem usada para representar valores numéricos na memória ou quando transmitidos pela rede
  - **big-endian** – bytes em ordem decrescente do seu peso numérico – byte mais significativo é armazenado primeiro
  - **little-endian** – bytes em ordem crescente do seu peso numérico – byte menos significativo é armazenado primeiro
- Fazendo uma analogia com a escrita de um número 123 no papel
  - 123 (big-endian)
  - 321 (little-endian)

Heterogeneidade de representação de dados



- Arquitetura x86-64 usa little-endian (convenção da Intel)
- Poucas arquiteturas (PowerPC antigo, Xilinx MicroBlaze, etc) usam o big-endian, porém foi convencionado pela IETF para ser usado pelos protocolos da Internet
  - Cabeçalho IP usa big-endian
- **bi-endian** podem operar com ambas (ARM, MIPS, IA-64)

Quantidade de bytes para representar um número inteiro

- Em Java o tipo primitivo `long` sempre ocupa 8 bytes
- Na linguagem C depende da arquitetura de máquina

```c
#include<limits.h>
int main(void){
    int i; long l;
    printf("%ld, %d",sizeof(i),INT_MAX);
    printf("%ld, %ld",sizeof(l),LONG_MAX);
}

/* Resultado em uma máquina de 64bits */
4, 2147483647
8, 9223372036854775807

/* Resultado em uma máquina de 32bits */
4, 2147483647
4, 2147483647
```

Codificação de caracteres

- 01 byte para representar um caracter em ASCII
- De 01 a 04 bytes para representar um caracter em UTF-8
- RFC 3629 define o UTF-8 como elemento padrão dos protocolos da Internet
- Python 3 tem o Unicode como o padrão para texto
- JVM usa o padrão do sistema operacional
  - Existe uma proposta de padrão para tornar o UTF-8 como a codificação padrão para toda API do JDK

Prática: ferramenta netcat

Ferramenta netcat

- Servidor aceita uma única conexão e depois encerra

```
# Ouvir na porta 1234/TCP
nc -l 1234
```

```
# Conectar no servidor
nc IP-do-servidor 1234
```

- Cliente enviando um arquivo para o servidor

```
nc -l 1234 > arquivo.txt
```

```
nc IP-do-servidor 1234 < arq.txt
```

- Usando UDP

```
nc -u -l 1234
```

```
nc -u IP-do-servidor 1234
```

- Um simples servidor web

```
echo -e 'HTTP/1.1 200 OK\n\n <html><h1>Ola mundo</h1></html>' | nc -l
    1234
```


Leitura obrigatória

- Leitura do capítulo 4 do livro *Distributed Systems* 3rd edition
- Ler e executar os códigos sobre Python3 disponíveis em https://github.com/std29006/oo-java-e-python

Aulas baseadas em

TANENBAUM, ANDREW S.; STEEN, MAARTEN VAN

SISTEMAS DISTRIBUIDOS: PRINCÍPIOS E PARADIGMAS

COULOURIS, GEORGE; KINDBERG, TIM; DOLLIMORE, JEAN

SISTEMAS DISTRIBUÍDOS: CONCEITOS E PROJETO

PAUL KRZYZANOWSKI

DISTRIBUTED SYSTEMS – RUTGERS UNIVERSITY