# Paradigma Cliente/Servidor

Mário Meireles Teixeira
UFMA – DEINF

## Comunicação em Sistemas Distribuídos

- Os processos em um SD estão lógica e fisicamente separados. Precisam se comunicar para que possam interagir
- O desempenho de um SD depende criticamente do desempenho do seu subsistema de comunicação
- O paradigma de comunicação mais comumente usado para a interação entre processos é o *Paradigma Cliente-Servidor*

## Conceitos: Paradigma C/S

- **Cliente**
  - Necessita do acesso aos recursos administrados pelos servidores para realizar suas tarefas. Dele partem as solicitações de serviço
  - Elemento *pró-ativo*, consumidor de serviços
- **Servidor**
  - Responsável por gerenciar um determinado tipo de recurso do sistema, administrando o acesso concorrente dos clientes ao mesmo
  - Elemento *reativo*, fornecedor de serviços



## Conceitos: Paradigma C/S

- Todo recurso compartilhado em um Sistema Distribuído (HW, SW ou dados) está sob a guarda de um processo servidor
- A relação cliente/servidor se estabelece entre cada interação entre processos, sendo um conceito de software, não de hardware
  - Clientes e Servidores podem executar na mesma máquina
  - Um servidor pode ser cliente de outro processo
- Servidor x Serviço
- Cliente x Usuário

## Conceitos: Paradigma C/S

- Quem são os clientes ?
  - Pode ser qualquer computador (*workstation*) conectado ao sistema através da rede
- Quem são os servidores ?
  - Geralmente computadores de grande capacidade, que oferecem aos clientes recursos como discos, impressoras, bancos de dados, conexões com outras redes, serviços da Web...




## Vantagens e Desvantagens

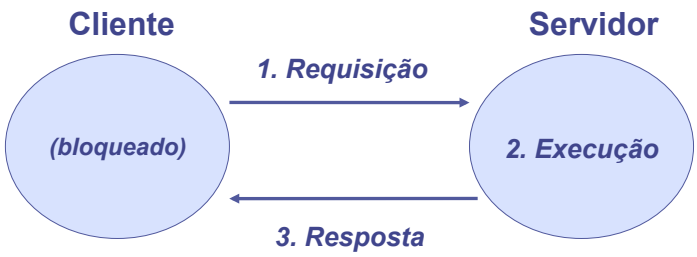
- Vantagens:
  - Compartilhamento de recursos
  - Balanceamento de carga
  - Tolerância a falhas
  - Escalabilidade
  - Transparência
  - Autonomia e Flexibilidade
  - Capacidade de processamento local e remota
  - Filosofia de *Sistemas Abertos*
  - Multiplataforma
  - Custos menores...

## Vantagens e Desvantagens

- Desvantagens:
  - Administração do sistema mais complexa
  - Variados pontos de falha no sistema
  - Dificuldades na interoperabilidade entre componentes de fornecedores diferentes
  - Mudança no paradigma de desenvolvimento de software



## Comunicação no Modelo C/S



## Comunicação no Paradigma C/S

- No Paradigma Cliente-Servidor, a comunicação objetiva principalmente a realização de serviços:
  - Cliente envia requisição ao servidor
  - Servidor recebe a mensagem e processa a solicitação
  - Servidor retorna os resultados ao cliente
- A comunicação entre as partes é implementada usando-se *passagem de mensagens*
- Em nível de programação, normalmente utilizam-se abstrações como *sockets*, *RPCs*, *objetos distribuídos*, *serviços web*



## Comunicação no Paradigma C/S

- Um processo servidor pode ter vários clientes e não precisa ter um conhecimento prévio a respeito deles
- Os clientes, antes de contactar um servidor pela primeira vez, geralmente consultam algum Serviço de Nomes (*binder, port mapper*) existente no sistema
- Cliente ou Servidor são papéis que os processos assumem durante uma interação particular

## Passagem de Mensagens

- Os processos comunicam-se através do envio de mensagens, utilizando primitivas do tipo *send/receive*
- Solução de "baixo nível"
  - O programador tem que se preocupar com o envio de mensagens, sincronização, erros de transmissão, mensagens perdidas, timeouts, detalhes de protocolos...
  - Porém, oferece melhor desempenho
- Exemplos de interfaces de passagem de mensagens:
  - *Sockets (TCP/IP)*, IPX/SPX (Netware), TLI, NetBIOS (IBM e Microsoft)



## Passagem de Mensagens

- Primitivas de Comunicação:
  - Send (msg, dest)
  - Receive (msg, orig)
- **Interação Síncrona**
  - O Receive síncrono faz com que o receptor fique bloqueado até a chegada de uma mensagem
  - No Send síncrono, o processo emissor fica bloqueado até que ocorra o Receive
  - Neste caso, diz-se que o Send e o Receive são *bloqueantes*

## Passagem de Mensagens

- **Interação Assíncrona**
  - Após o Send, o processo emissor está liberado tão logo a mensagem tenha sido copiada para um buffer local
  - O programa receptor apenas informa sua intenção de receber uma mensagem, alocando um buffer para recepção
    - O receptor fica sabendo da chegada de uma mensagem através de *polling* ou interrupções
  - Diz-se que o Send e o Receive são *não-bloqueantes*



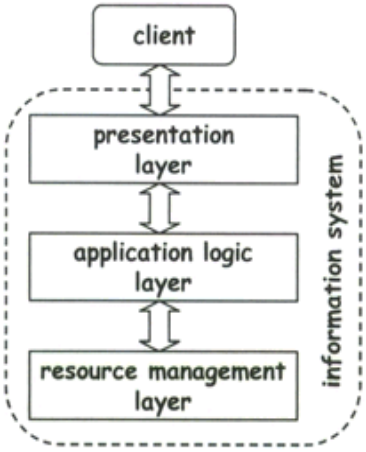
## Passagem de Mensagens

- **Interação Assíncrona**
  - A comunicação assíncrona oferece melhor desempenho, mas sua programação é mais difícil que a forma síncrona
  - As primitivas também podem ser *confiáveis* e *não-confiáveis*
  - Mais sobre este assunto:
    - *Programação com Sockets*

# Camadas de Aplicação em Sistemas de Informação

- **Apresentação:** disponibiliza a informação às entidades externas e permite sua interação com o sistema
  - Cliente = Camada de Apresentação ??
  - Cliente Web comum vs. Applet Java
- **Lógica da Aplicação (regras de negócio):** o que o sistema realmente faz, os *serviços* que ele oferece – garante as regras de negócio, coordena os processos
- **Gerenciamento de Recursos (camada de dados):** as diferentes fontes de dados que dão suporte à aplicação (SGBD, arquivos, outros sistemas)

# Camadas de Aplicação

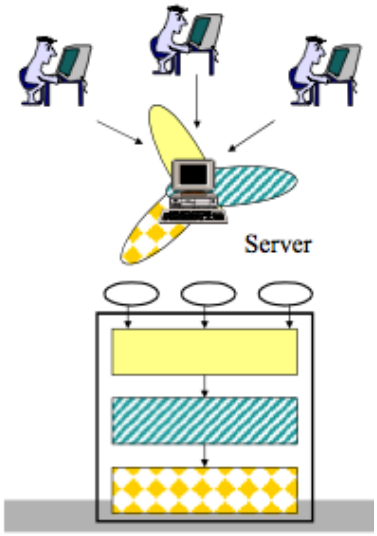


# Arquitetura de Sistemas Cliente-Servidor

- Um sistema cliente-servidor pode ser conceitualmente dividido em camadas, de acordo com suas funcionalidades
- Essas camadas (*tiers*) podem ser combinadas e distribuídas de diferentes maneiras
- Assim, temos sistemas: *1-tier, 2-tier, 3-tier e N-tier*



# Arquitetura 1-tier: a era do mainframe




- Todas as camadas são combinadas em uma única entidade (sistemas monolíticos)
- Arquitetura típica de sistemas baseados em *mainframes*
- Usuários têm acesso ao sistema a partir de terminais burros: todo o processamento (inclusive apresentação) é controlado pelo servidor central

# Arquitetura 1-tier

**Vantagens**

- único contexto de execução
- não há necessidade de manter e publicar uma interface de serviços do sistema
- produz em geral sistemas mais eficientes (melhor desempenho)

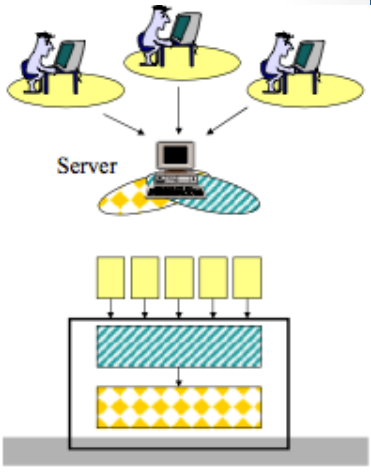
**Desvantagens**

- código monolítico (difícil de manter e evoluir)
- baixo reuso de código
- na contramão da indústria de software (há muitas décadas)

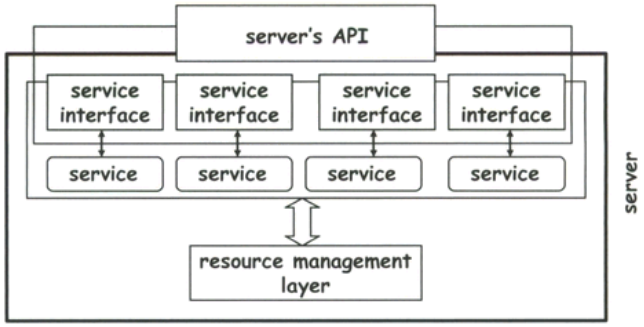


# Arquitetura 2-tier: a era da API

- Com o advento do PC, a camada de apresentação migra para os clientes
- É possível aproveitar o poder computacional dos clientes
- Múltiplas camadas de apresentação customizadas
- Two-tier = Client/Server
  - thin/fat client
- Surgimento da RPC → API do servidor



# Marcos na evolução de sistemas distribuídos: 2-tier




- Conceito de API: torna possível suportar múltiplos clientes; evolução do servidor sem afetar os clientes
- Noção de 'serviço': cliente invoca um serviço implementado por um servidor
- Web Services são o subproduto mais recente de toda essa evolução



# Arquitetura 2-tier

**Vantagens**

- Camadas de negócio e dados permanecem juntas (mais eficiência)
- Camada de apresentação independente do servidor (mais portabilidade)

**Desvantagens**

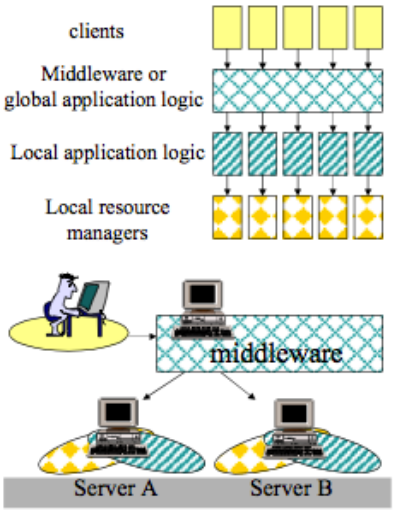
- Servidor tem um limite máximo de clientes que pode suportar (escalabilidade)
- Cliente tem dificuldades de 'conversar' com múltiplos servidores... ☹
- Solução: Middleware

# Arquitetura 3-tier: a era do middleware

- Resolve o problema da integração entre múltiplos servidores
- Middleware: nova camada entre clientes e servidores
  - simplifica o projeto dos clientes, pois reduz o número de interfaces que este precisa conhecer
  - encapsula a lógica de integração entre os sistemas e a lógica de aplicação de alto nível
  - localiza os recursos, faz acesso a eles, coleta/organiza os resultados (middleware = elemento mediador)
  - inicialmente, surgiu com SGBD's; posteriormente, mais genérico (object bus, service bus)

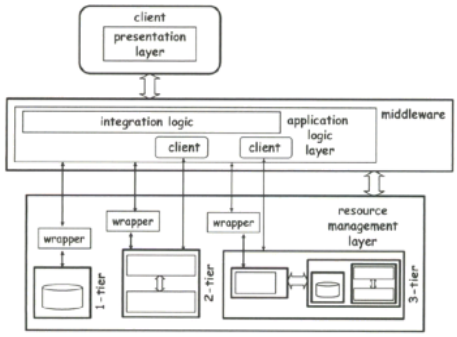


# Arquitetura 3-tier



# 3-tier: subdivisão em camadas

- Na arquitetura 3-tier, existe uma clara separação entre as três camadas:
  - camada de apresentação reside no cliente
  - lógica de integração e da aplicação, na camada internediária
  - camada de gerenciamento de recursos são os próprios servidores que o 3-tier visa integrar
    - cada servidor neste nível pode estar subdividido em novas camadas (tiers)






# Comparação entre arquiteturas

**2-tier**

- lógica da aplicação e gerenciamento de recursos na mesma máquina
  - comunicação eficiente
  - baixa escalabilidade

**3-tier**

- cada camada em seu próprio servidor (ou cluster)
- lógica da aplicação mais independente dos recursos (mais portabi-lidade e reusabilidade)
- comunicação entre módulos computacio-nalmente mais cara

# Arquitetura N-tier

- Surge, de maneira geral, em dois tipos de cenário:
  - Interligação entre sistemas distintos
  - Interconectividade através da Internet
- Interligação entre sistemas distintos
  - A camada de recursos pode incluir não apenas bancos de dados, mas também sistemas 2-tier e 3-tier completos
  - Arquitetura N-tier ou multi-tier
- Interconectividade através da Internet
  - A Web surge como (mais) uma camada de software
  - O servidor web é incorporado à camada de apresentação (uma camada a mais, na verdade)
    - Deu origem aos 'servidores de aplicação', que oferecem serviços usando a Web como plataforma de acesso a eles

# Arquitetura N-tier