

ESBD 4

Sistemas Distribuídos

2023

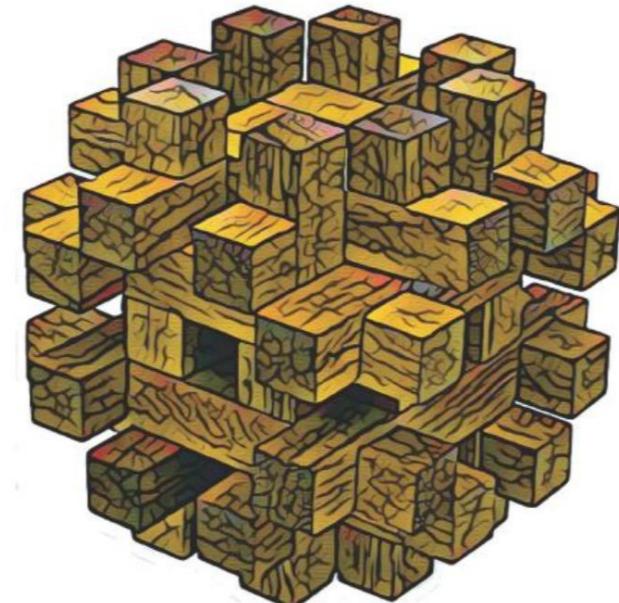


**INFORMAÇÃO,
TECNOLOGIA
& INOVAÇÃO**

LIVRO TEXTO PARA O CURSO

Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



3th Edition – Version 3.03 – 2020

Site Oficial: <https://www.distributed-systems.net/index.php/books/ds3/ds3-ebook/>

Capítulos

- 1. Introdução**
- 2. Arquiteturas**
- 3. Processos**
- 4. Comunicação**
- 5. Nomes**
- 6. Coordenação**
- 7. Consistência e Replicação**
- 8. Tolerância a Falhas**
- 9. Segurança**

06/maio/2023



CAPÍTULO 1

INTRODUÇÃO

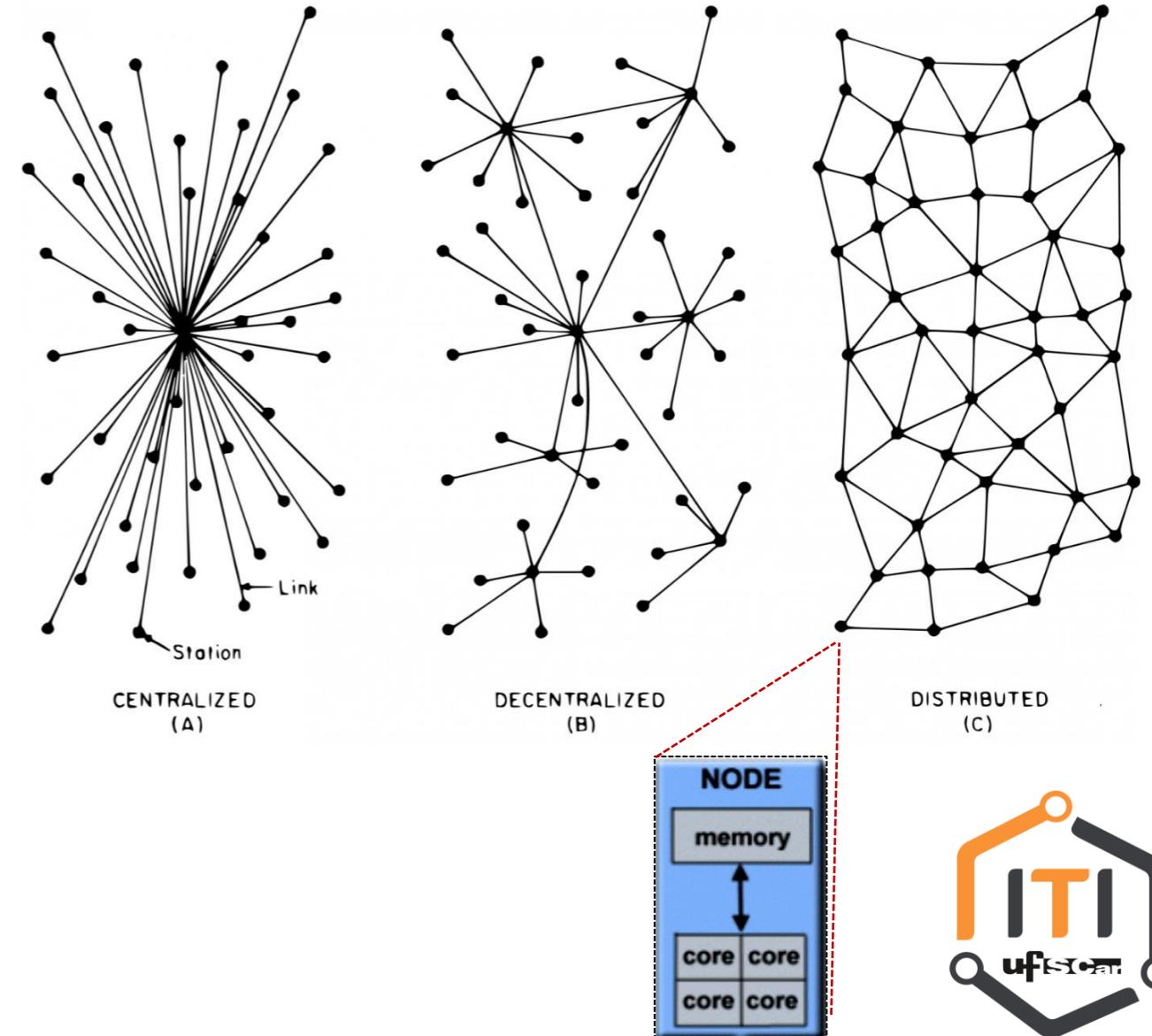
O QUE É UM SISTEMA DISTRIBUÍDO ?

- Definição:

- Um sistema distribuído é uma coleção de **elementos computacionais autônomos** que aparece para seus usuários como um **sistema coerente único**.

- Propriedades Características:

- Elementos computacionais autônomos, também conhecidos por **“nodes” (nós)**, são dispositivos de hardware ou processos de software.
- Sistema coerente único: usuários ou aplicações percebem (enxergam) um sistema único → nós (nodes) necessitam **colaboração**.



INTRODUÇÃO O QUE É UM SISTEMA DISTRIBUÍDO

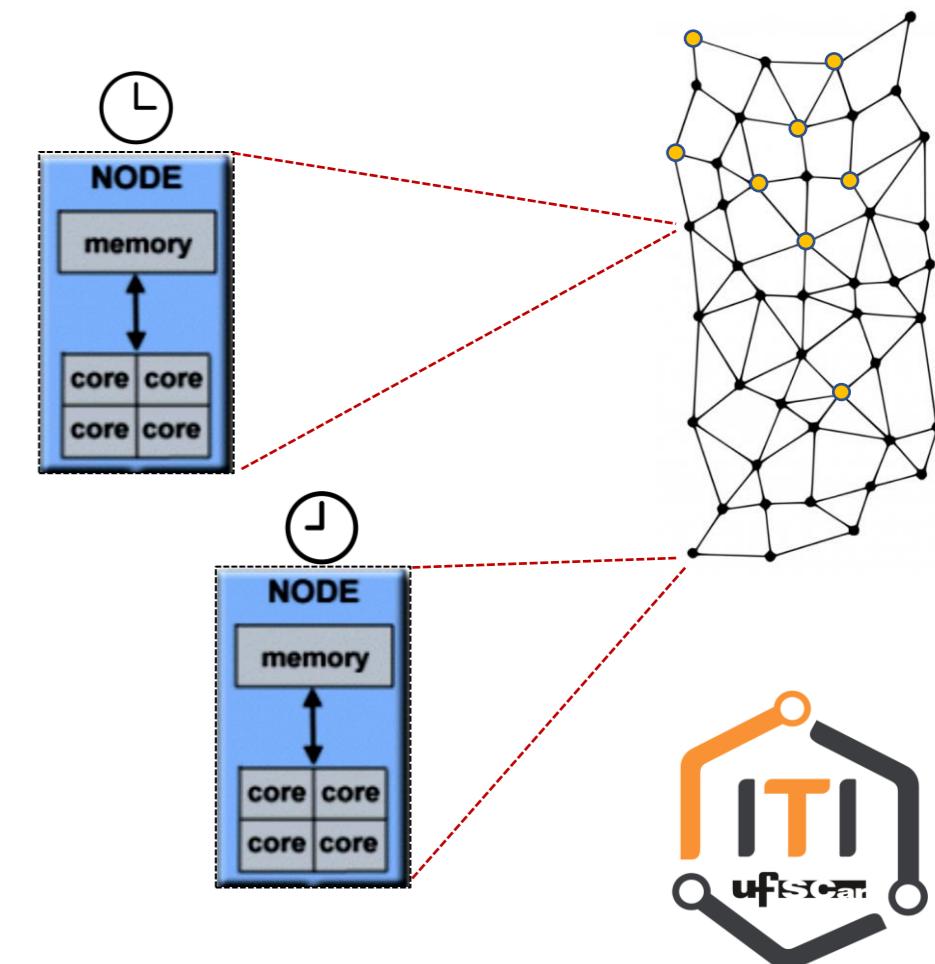
COLEÇÃO DE NÓS AUTÔNOMOS

- **Comportamento Independente:**

- Cada nó é autônomo e portanto possuí sua **própria noção de tempo**: não existe **relógio global**. Leva a problemas fundamentais de sincronização e coordenação.

- **Coleção de nós:**

- Como gerenciar **associação ao grupo (group membership)** ?
- Como saber se você está de fato se comunicando com um **membro (não) autorizado** de um grupo ?



CARACTERÍSTICA 1 COLEÇÃO DE NÓS AUTÔNOMOS

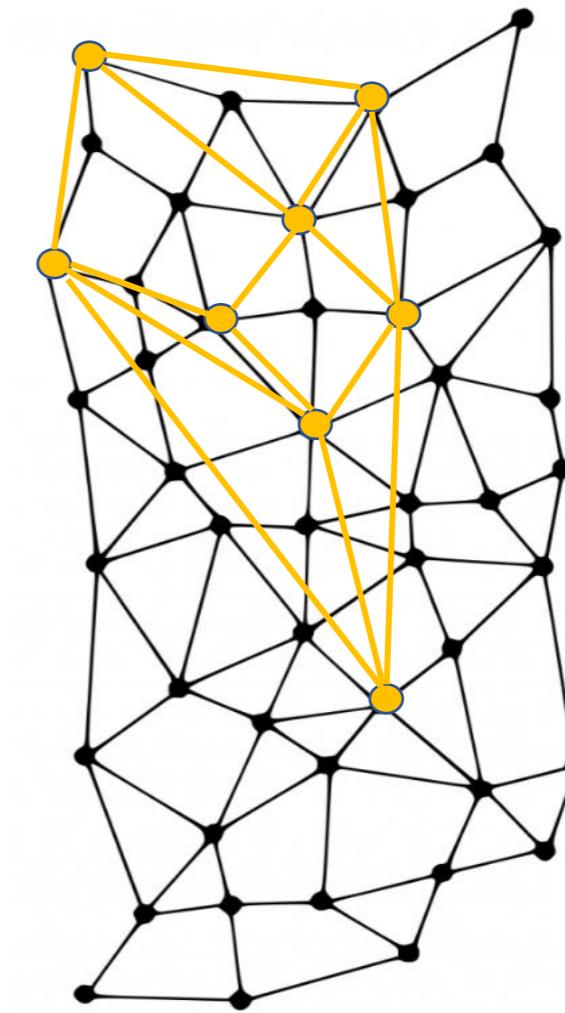
ORGANIZAÇÃO

- **Rede Sobreposta (Overlay Network):**

- Cada nó da coleção se comunica somente com outros nós no sistema, seus **vizinhos**. O conjunto de vizinhos pode ser dinâmico, ou pode ainda ser conhecido somente de forma implícita (isto é, demanda *lookup*).

- **Tipos de redes Overlay:**

- Um exemplo bem conhecido de redes sobrepostas: **sistemas peer-to-peer**.
 - **Estruturada**: cada nó tem um **conjunto de vizinhos bem definidos** com os quais ele (nó) pode se comunicar;
 - **Não estruturada**: cada nó possuí referências para **outros nós selecionados randomicamente** de nós do sistema



CARACTERÍSTICA 2 SISTEMA COERENTE ÚNICO

ORGANIZAÇÃO

- Essência

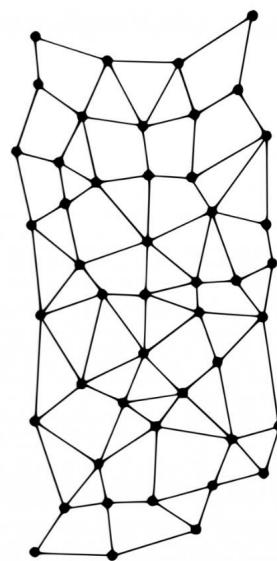
- Uma coleção de nós como um todo opera o mesmo, independente de onde, quando e como a interação entre usuário e sistema ocorre.

- Exemplos

- Um usuário final não é capaz de dizer onde a computação está ocorrendo;
- Onde exatamente os dados são armazenados deveria ser irrelevante para a aplicação;
- Se os dados foram ou não replicados está completamente escondido;

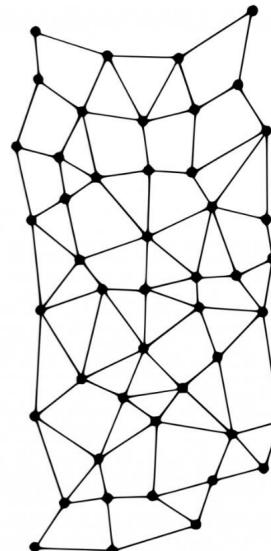
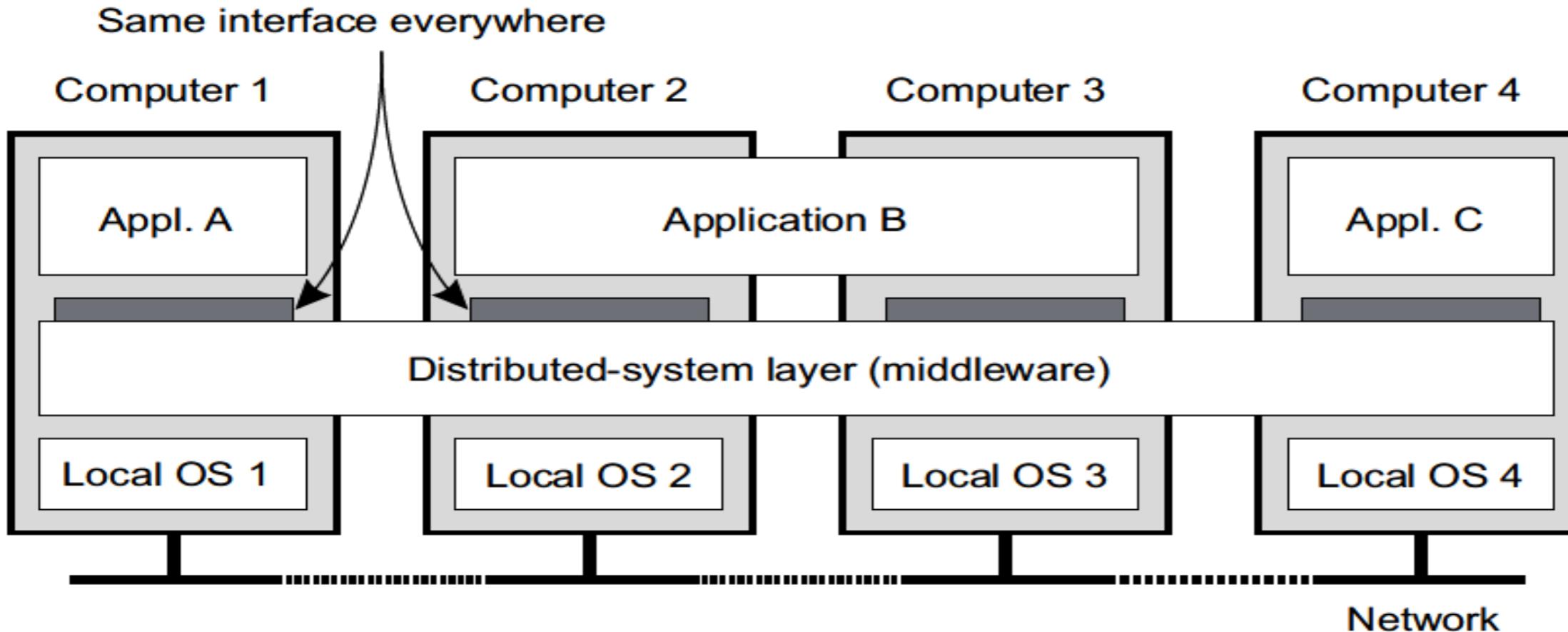
- O problema: falhas parciais

- É inevitável a qualquer hora somente uma parte de um sistema distribuído possa falhar. Esconder falhas parciais e recuperá-las é frequentemente difícil e em geral impossível de esconder.



MIDDLEWARE O SO DE SISTEMAS DISTRIBUÍDOS

MIDDLEWARE E SISTEMAS DISTRIBUÍDOS

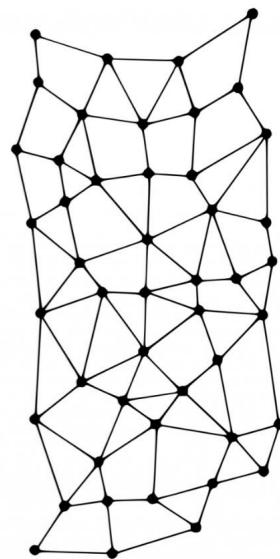


- O que tem em um middleware ?

- Componentes e funções de uso comum que não precisam ser implementadas nas aplicações de forma separada.

O que buscamos alcançar ?

- Suporte ao compartilhamento de recursos
- Transparência na distribuição
- Abertura (openness)
- Escalabilidade (scalability)



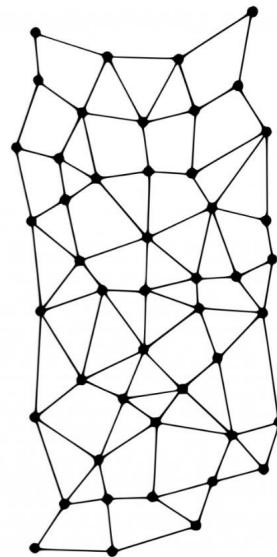
COMPARTILHANDO RECURSOS

- Exemplos Canônicos (de acordo com as normas)

- Armazenamento de arquivos compartilhados baseado em nûvem
- Sistema de *streaming* multimídia *peer-to-peer* assistido
- Serviços de email compartilhado (pense em serviços terceirizados)
- Compartilhamento de Web Hosting (pense em redes de distribuição de conteúdos)

- Observação

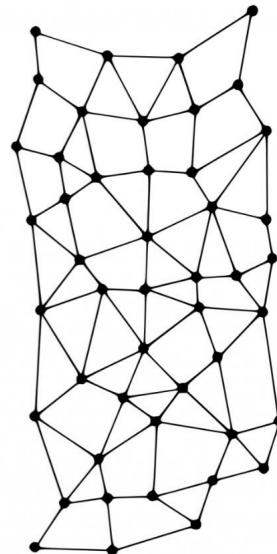
- “A rede é o computador” (John Cage, então na Sun Microsystems)



TRANSPARÊNCIA NA DISTRIBUIÇÃO

TIPOS

Transparência	Descrição
Acesso	Esconde diferenças em representação de dados e como um objeto é acessado
Localidade	Esconde a localização do objeto
Relocação	Esconde que um objeto pode se mover para outra localização enquanto em uso
Migração	Esconde que um objeto pode se mover para outra localização
Replicação	Esconde que um objeto é replicado
Concorrência	Esconde que um objeto pode ser compartilhado por vários usuários independentes
Falha	Esconde falhas e recuperações de um objeto

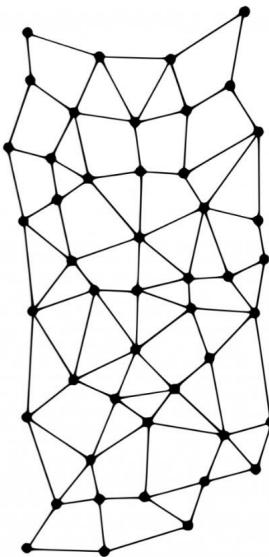


GRAU DE TRANSPARÊNCIA

Observação

Transparência total pode ser um exagero:

- Existe latências de comunicação que não podem ser escondidas
- Esconder falhas por completo de redes e nós (teoricamente e praticamente) é impossível
 - Não dá para distinguir um computador lento de um que está falhando
 - Não dá para ter certeza que um servidor realizou uma operação antes de uma falha
- Transparência total degrada o desempenho (custo), expondo a distribuição do sistema
 - Manter replicas atualizadas de forma exata com a cópia mestre gasta tempo
 - Executar imediatamente operações de gravação no disco para tolerância a falhas



GRAU DE TRANSPARÊNCIA

Expor distribuição pode ser bom

- Fazer uso de serviços locais (achar amigos próximos)
- Ao lidar com usuários em diferentes fusos horários
- Quando fica mais fácil para um usuário entender o que está acontecendo (quando por exemplo, um servidor não responde por um longo tempo, relate-o como falho).

Conclusão

- Transparência na distribuição é um objetivo legal, mas conseguir é uma história diferente, e muitas vezes não deve ser mirado ..



ABERTURA DE SISTEMAS DISTRÍBUIDOS

Do que estamos falando ?

- Ter a capacidade de interagir com serviços e outros sistemas abertos, independente do ambiente de baixo
 - Sistemas devem estar em conformidade com **interfaces** bem definidas
 - Sistemas devem **interoperar** facilmente
 - Sistemas devem suportar **portabilidade** de aplicações
 - Sistemas devem ser facilmente **extensíveis**



ESCALA EM SISTEMAS DISTRIBUÍDOS

OBSERVAÇÃO

- Muitos desenvolvedores de sistemas distribuídos modernos facilmente usam o adjetivo **escalável** sem deixar claro **porque** seus sistemas escalam.

PELO MENOS TRÊS COMPONENTES

- Número máximo de usuários e/ou processos (**escalabilidade de tamanho**)
- Máxima distância entre nós (**escalabilidade geográfica**)
- Número de domínios administrativos (**escalabilidade administrativa**)

OBSERVAÇÃO

- Muitos sistemas contabilizam somente, até um certo ponto, escalabilidade de tamanho. Frequentemente usando a solução: múltiplos servidores poderosos operando independentemente em paralelo. Hoje, o desafio está na escalabilidade geográfica e administrativa.



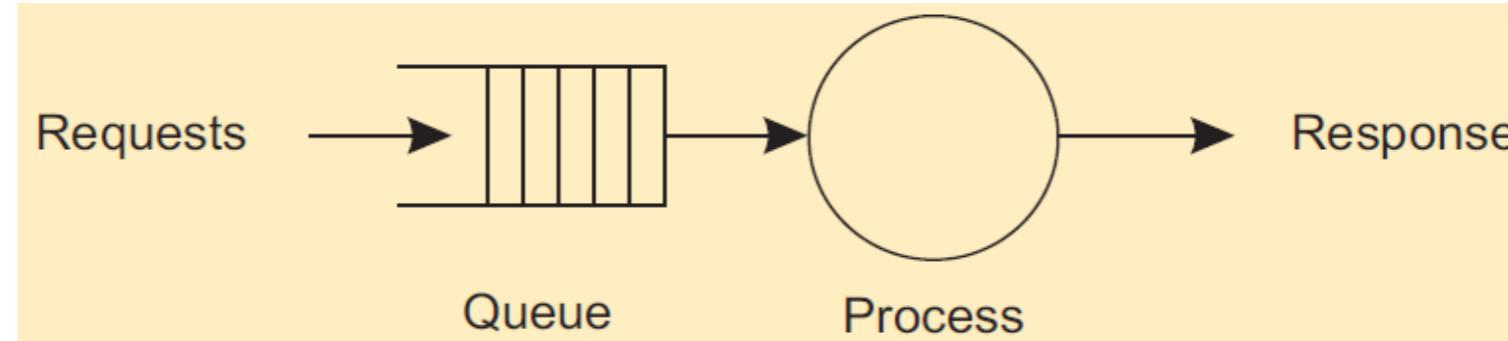
ESCALABILIDADE DE TAMANHO

CAUSAS RAIZ PARA PROBLEMAS DE ESCALABILIDADE COM SOLUÇÕES CENTRALIZADAS

- Capacidade computacional limitada por CPU's
- Capacidade de armazenamento incluindo taxa de transferência entre CPU's e discos
- A rede entre usuário e o serviço centralizado

ANÁLISE FORMAL

UM SERVIÇO
CENTRALIZADO
PODE SER
MODELADO COMO
UM SISTEMA DE FILA
SIMPLES



SUPOSIÇÕES E NOTAÇÕES

- A fila tem tamanho infinito → a taxa de chegada de requisições não é influenciada pelo tamanho atual da fila ou o que está sendo processado
- Taxa de chegada de requisições: λ requisições/seg
- Capacidade do processamento do serviço: μ requisições/seg

FRAÇÃO DE TEMPO NA QUAL HÁ k REQUISIÇÕES NO SISTEMA

$$p_k = \left(1 - \frac{\lambda}{\mu}\right) \left(\frac{\lambda}{\mu}\right)^k$$

ANÁLISE FORMAL

UTILIZAÇÃO U DE UM SERVIÇO É A FRAÇÃO DE TEMPO QUE ELE ESTÁ OCUPADO

$$U = \sum_{k \geq 0} p_k = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_k = (1 - U) U^k$$

NÚMERO MÉDIO DE REQUISIÇÕES EM UM SISTEMA

$$\bar{N} = \sum_{k \geq 0} k \cdot p_k = \sum_{k \geq 0} k \cdot (1 - U) U^k = (1 - U) \sum_{k \geq 0} k \cdot U^k = \frac{(1 - U)U}{(1 - U)^2} = \frac{U}{1 - U}$$

VAZÃO MÉDIA

$$X = \underbrace{U \cdot \mu}_{\text{server at work}} + \underbrace{(1 - U) \cdot 0}_{\text{server idle}} = \frac{\lambda}{\mu} \cdot \mu = \lambda$$



ANÁLISE FORMAL

**TEMPO DE RESPOSTA: TEMPO TOTAL PARA PROCESSAR
UMA REQUISIÇÃO DEPOIS DA SUBMISSÃO**

$$R = \frac{\bar{N}}{X} = \frac{S}{1 - U} \Rightarrow \frac{R}{S} = \frac{1}{1 - U}$$

- Onde $S = 1/\mu$ sendo o tempo do serviço

OBSERVAÇÕES

- Se U é pequeno, a resposta para o tempo de serviço é perto de 1: uma requisição é processada imediatamente
- Se U cresce até 1, o sistema vai para travamento. Solução: diminúa S (aumente a capacidade de processamento)

PROBLEMAS COM ESCALABILIDADE GEOGRÁFICA

ESCONDER LATÊNCIAS DE COMUNICAÇÃO

- Não pode simplesmente mudar de LAN para WAN: muitos sistemas distribuídos assumem **interações síncronas entre cliente e servidor**: cliente envia requisição e espera por resposta: **Latência** pode facilmente proibir este esquema.
- Links WAN frequentemente são inherentemente **não confiáveis**: mudar um stream de vídeos de LAN para WAN é propenso a falhar.
- **Falta de comunicação multiponto**, de forma que um broadcast de busca simples não pode ser disseminado. A solução é desenvolver de forma separada, **serviços de nomes e diretórios** (com seu próprio problema de escalabilidade)



PROBLEMAS COM ESCALABILIDADE GEOGRÁFICA

ESSÊNCIA

- Políticas conflitantes em relação ao uso (e sobre pagamento), gerenciamento e segurança.

EXEMPLOS

- **Grades (grids) computacionais:** compartilha recursos caros entre diferentes domínios
- **Equipamentos compartilhados:** como controlar, gerenciar, e usar um rádio telescópio compartilhado construído como uma de sensores de grande escala ?

EXCEÇÃO: MUITAS REDES PEER-TO-PEER

- Sistema de compartilhamento de arquivos (baseados p. ex. no BitTorrent)
- Telefonia peer-to-peer (Skype)
- Streaming de áudio assistido por “peer” (Spotify)

Nota: **usuários finais** colaboram e não **entidades administrativas**



TÉCNICAS PARA ESCALABILIDADE

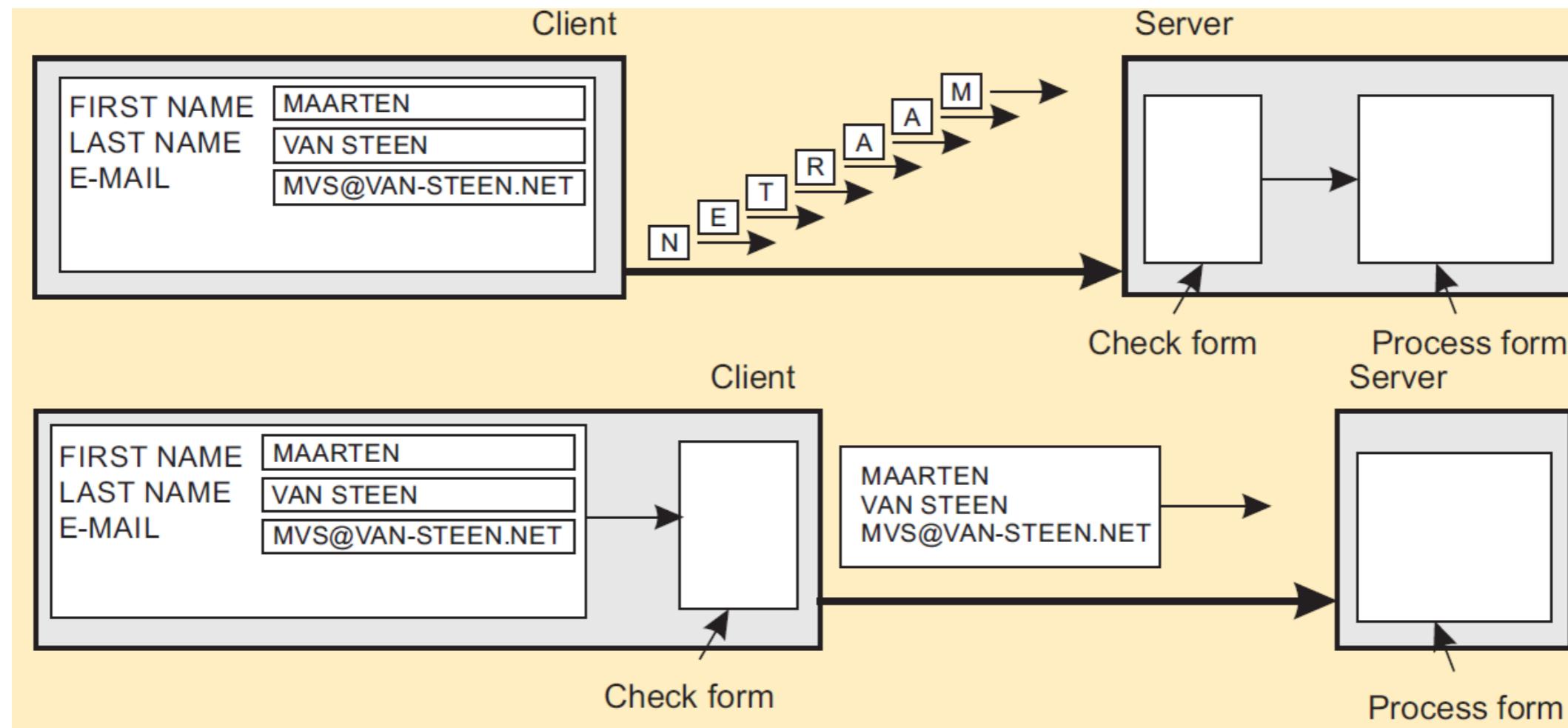
ESCONDER LATÊNCIAS DE COMUNICAÇÃO

- Fazer uso comunicação **assíncrona**
- Ter um manipulador (handler) separado para respostas que estão chegando
- **Problema:** nem sempre todas as aplicações cabem neste modelo



TÉCNICAS PARA ESCALABILIDADE

FACILITAR A SOLUÇÃO ATRAVÉS DA MUDANÇA DA COMPUTAÇÃO PARA O CLIENTE



TÉCNICAS PARA ESCALABILIDADE

PARTIÇÃO DE COMPUTAÇÃO E DADOS ATRAVÉS DE MULTIPLAS MÁQUINAS

- Mover a computação para clientes (Java applets)
- Descentralização de serviços de nomes (DNS)
- Descentralização de sistemas de informação (www)



TÉCNICAS PARA ESCALABILIDADE

REPLICAÇÃO E CACHING: DISPONIBILIZA CÓPIAS DE DADOS EM DIFERENTES MÁQUINAS

- Replicação de servidores de arquivos e base de dados
- Sites WEB espelhados
- Cache WEB (nos browsers e proxies)
- Cache de arquivos (no servidor e no cliente)



ESCALABILIDADE PROBLEMA DE REPLICAÇÃO

APLICAR REPLICAÇÃO É FÁCIL, EXCETO POR UMA COISA

- A existência de múltiplas cópias (cache ou replicadas), leva a **inconsistências**: modificar uma cópia torna-a diferente de todas as outras,
- Manter cópias consistentes sempre de uma forma geral demanda **sincronização global** em cada modificação.
- Sincronização global impede soluções de larga escala

OBSERVAÇÃO

- Se pudermos tolerar inconsistências, podemos reduzir a necessidade de sincronização global, porém a **tolerância de inconsistências é dependente da aplicação**.



DESENVOLVIMENTO DE SISTEMAS DISTRIBUÍDOS

PITFALLS

OBSERVAÇÃO

- Muitos sistemas distribuídos são desnecessariamente complexos devido a erros que demandaram correções (*patches* – remendos). Muitas **suposições falsas** são feitas com frequência.

SUPosições FALSAS (E FREQUENTEMENTE ESCONDIDAS)

- A rede é confiável
- A rede é segura
- A rede é homogênea
- A topologia não muda
- Latência é zero
- Largura de banda é infinita
- Custo de transporte é zero
- Existe um administrador

TRÊS TIPOS DE SISTEMAS DISTRIBUÍDOS

- Sistemas Computacionais Distribuídos de Alto Desempenho
- Sistemas de Informações Distribuídos
- Sistemas Distribuídos para Computação Pervasiva

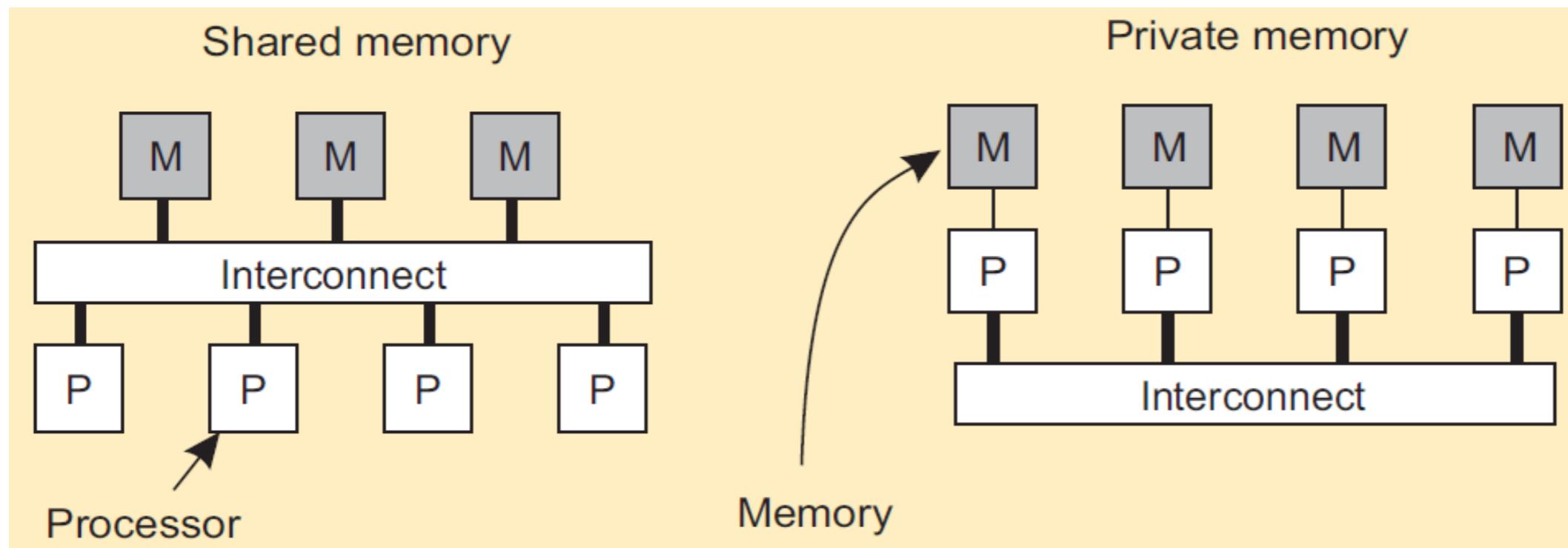


COMPUTAÇÃO PARALELA

OBSERVAÇÃO

- Computação Distribuída de Alto Desempenho começou com Computação Paralela

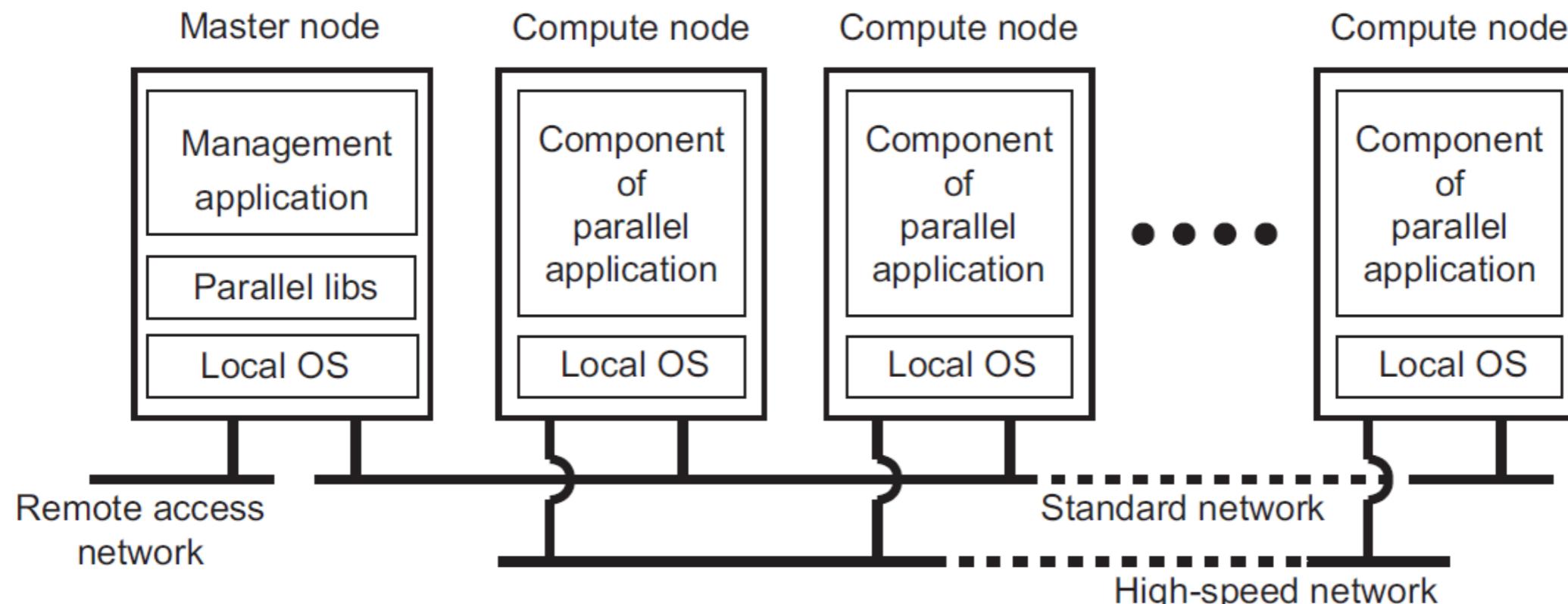
MULTIPROCESSADOR MULTINÚCLEO X MULTICOMPUTADOR



COMPUTAÇÃO EM CLUSTER

ESSENCIALMENTE UM GRUPO DE SISTEMAS HIGH-END CONECTADOS POR UMA LAN

- Homogêneo: mesmo SO, hardware quase idêntico
- Único nó de gerenciamento



COMPUTAÇÃO EM GRADE (GRID)

O PRÓXIMO PASSO: MUITOS NÓS DE TODOS LUGARES

- Heterogêneo
- Disperso através de diversas organizações
- Pode facilmente se espalhar por uma rede WAN

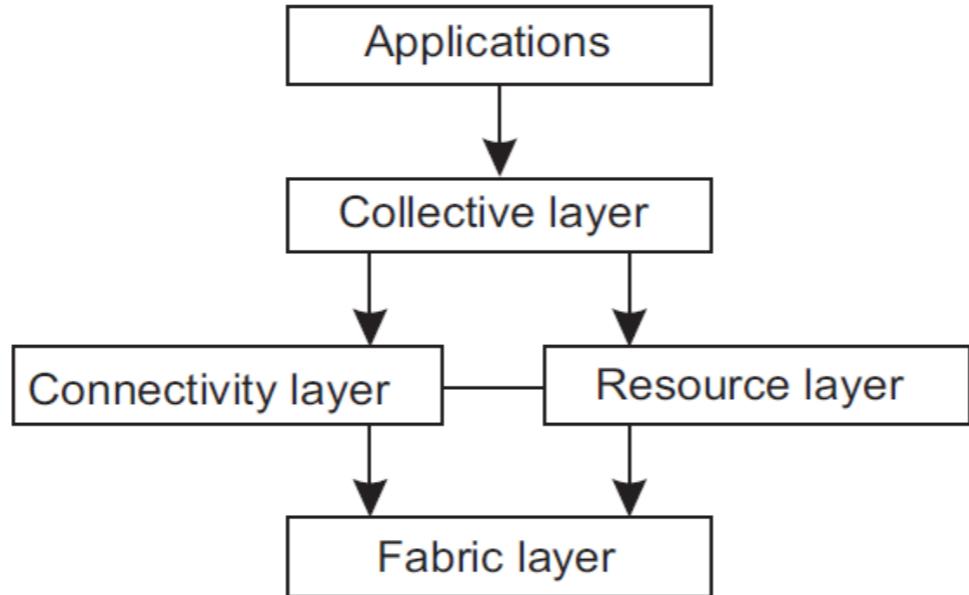
NOTA

Para permitir colaboração, grids geralmente usam organizações virtuais. Em essência, isto é um agrupamento de usuários (ou melhor: seus ID's) que irão permitir a autorização na alocação de recursos.



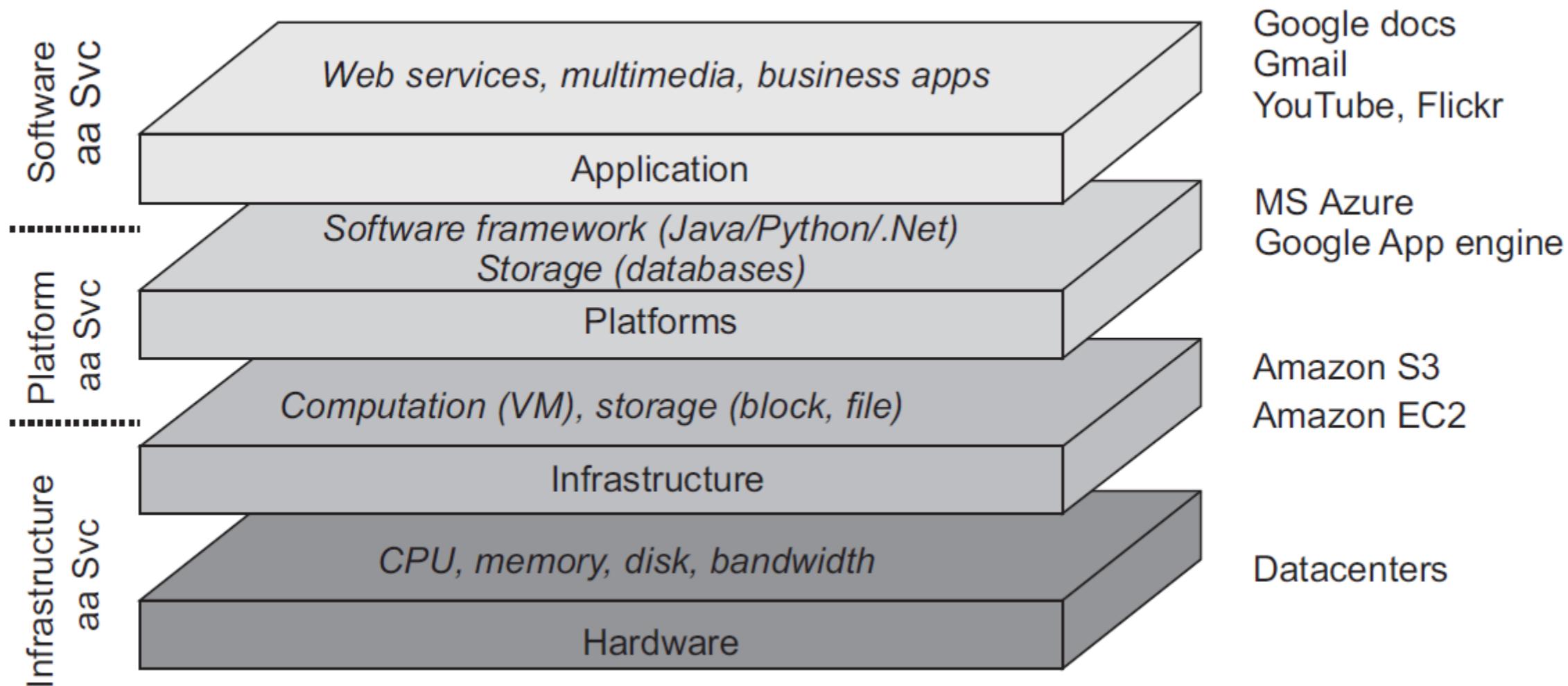
ARQUITETURA GRID COMPUTING

AS CAMADAS



- Fabrica: provê interfaces para recursos locais (para “querying” do estado e capacidades, locking etc..)
- Conectividade: protocolos de comunicação / transação, p.ex., para mover dados entre recursos. Presença de vários protocolos de autenticação.
- Recursos: Gerencia um único recurso, tal como a criação de processo ou leitura de dados.
- Coletivo: manuseia o acesso a múltiplos recursos: descoberta, escalonamento e replicação.
- Aplicação: contém as aplicações do grid em uma única organização

COMPUTAÇÃO EM NÚVEM (CLOUD)



INTEGRANDO APLICAÇÕES

SITUAÇÃO

Organizações usam muitas aplicações em rede, porém interoperabilidade é difícil de ser realizada

ABORDAGEM BÁSICA

Uma aplicação em rede que roda em um servidor disponibilizando serviços para clientes remotos: Integração: clientes combinam requisições para diferentes aplicações, despacham as requisições; coletam respostas e apresentam um resultado coerente para os usuários.

PRÓXIMO PASSO

Permitir comunicação direta aplicação-para-aplicação, levando a EAI – Enterprise Application Integration

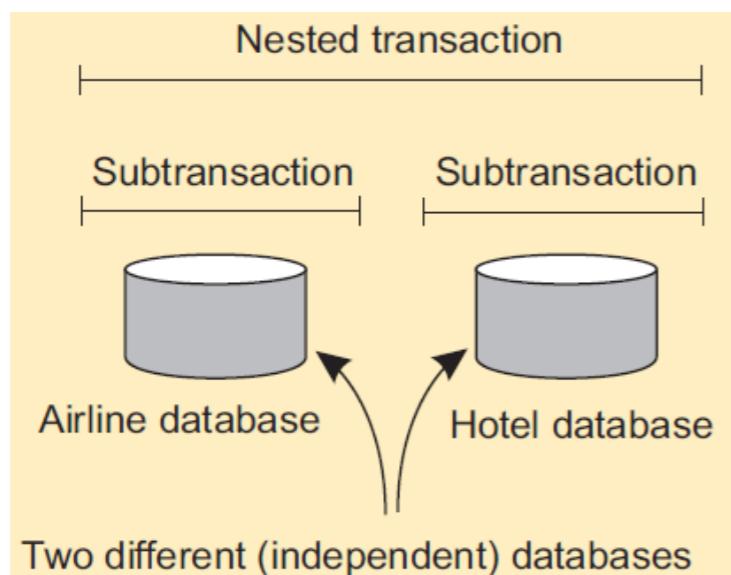


EXEMPLO EAI: TRANSAÇÕES (ANINHADAS)

TRANSAÇÃO

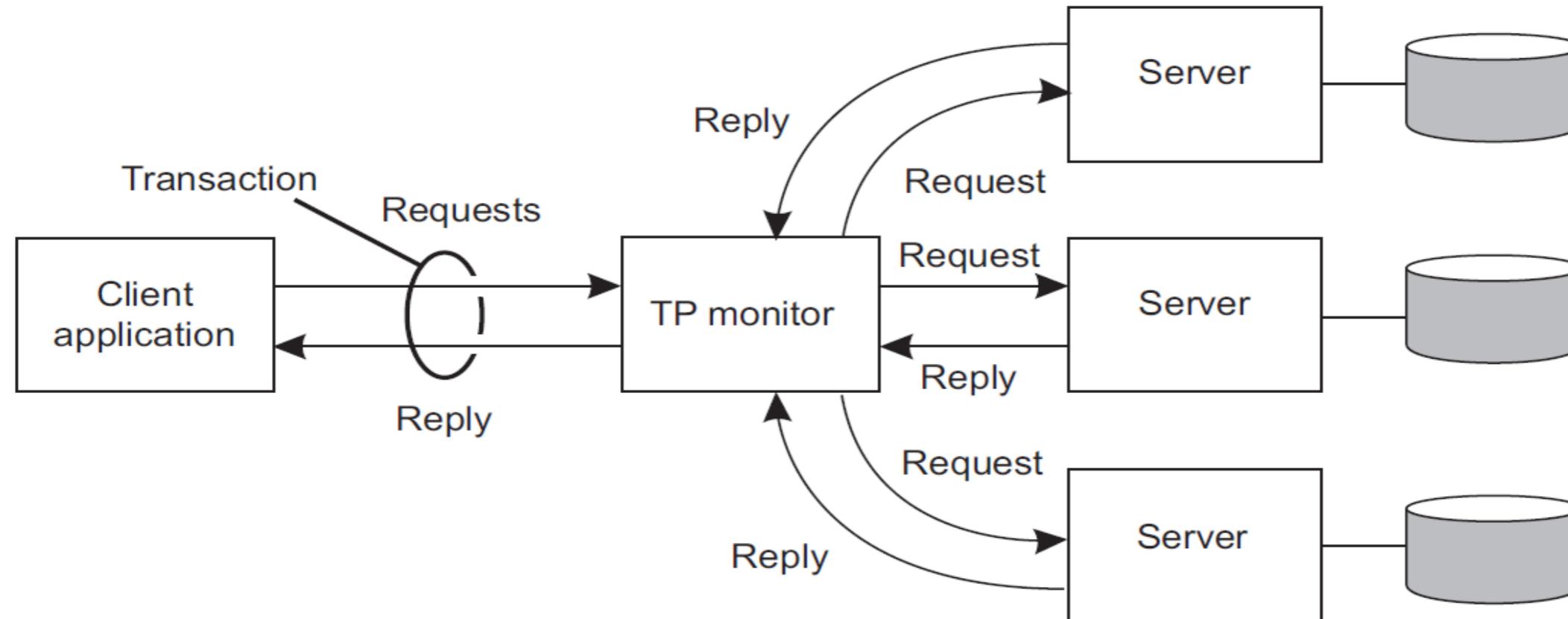
Primitive	Description
<i>BEGIN_TRANSACTION</i>	Mark the start of a transaction
<i>END_TRANSACTION</i>	Terminate the transaction and try to commit
<i>ABORT_TRANSACTION</i>	Kill the transaction and restore the old values
<i>READ</i>	Read data from a file, a table, or otherwise
<i>WRITE</i>	Write data to a file, a table, or otherwise

TRANSAÇÃO : TUDO OU NADA



- **Atômica:** acontece de forma indivisível (parecendo)
- **Consistente:** não viola invariantes do sistema
- **Isolada:** sem interferência mútua
- **Durável:** “**commit**” significa que mudanças são permanentes

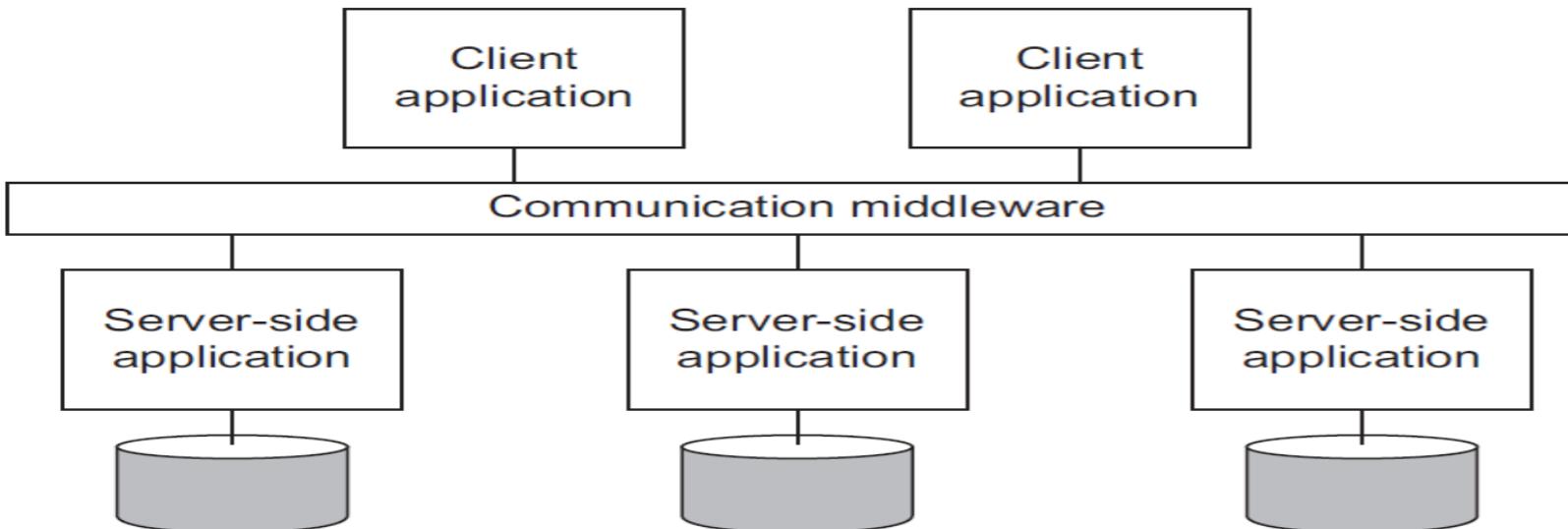
TPM: MONITOR DE PROCESSAMENTO DE TRANSAÇÕES



OBSERVAÇÃO

Em muitos casos, os dados envolvidos em uma transação é distribuído através de vários servidores. Um **monitor de TP** é responsável por coordenar a execução da transação

EAI E MIDDLEWARE



Middleware oferece facilidades de comunicação para integração

- RPC – Chamada Remota de Procedimento (**remote procedure call**): Requisições são enviadas através de uma chamada local a procedimento, empacotada como uma mensagem, processada, respondida através de mensagem e o resultado é retornado como um “return” de uma chamada
- MOM – Message Oriented Middleware: Mensagens são enviadas (**Published**) a um roteador de mensagens (**Broker**), e repassadas a aplicações assinantes do tópico (**Subscribers**)

COMO INTEGRAR APLICAÇÕES

- **Transferência de arquivos:** tecnicamente simples, mas sem flexibilidade:
 - Definir formato de arquivo comum de layout
 - Definir modelo de gerenciamento de arquivo
 - Atualizar propagação e atualizar notificações
- **Base de dados compartilhada:**
 - Mais flexível, mas ainda demanda um esquema comum de dados, tem risco de “bottleneck” (gargalo)
- **RPC:**
 - eficiente quando é necessário a execução de uma série de ações
- **Mensagens:**
 - RPC demanda que o chamante e chamado estejam em estado “rodando” ao mesmo tempo
 - Mensagens permitem o “descasamento” em tempo e espaço

REDES DE SENSORES

Características

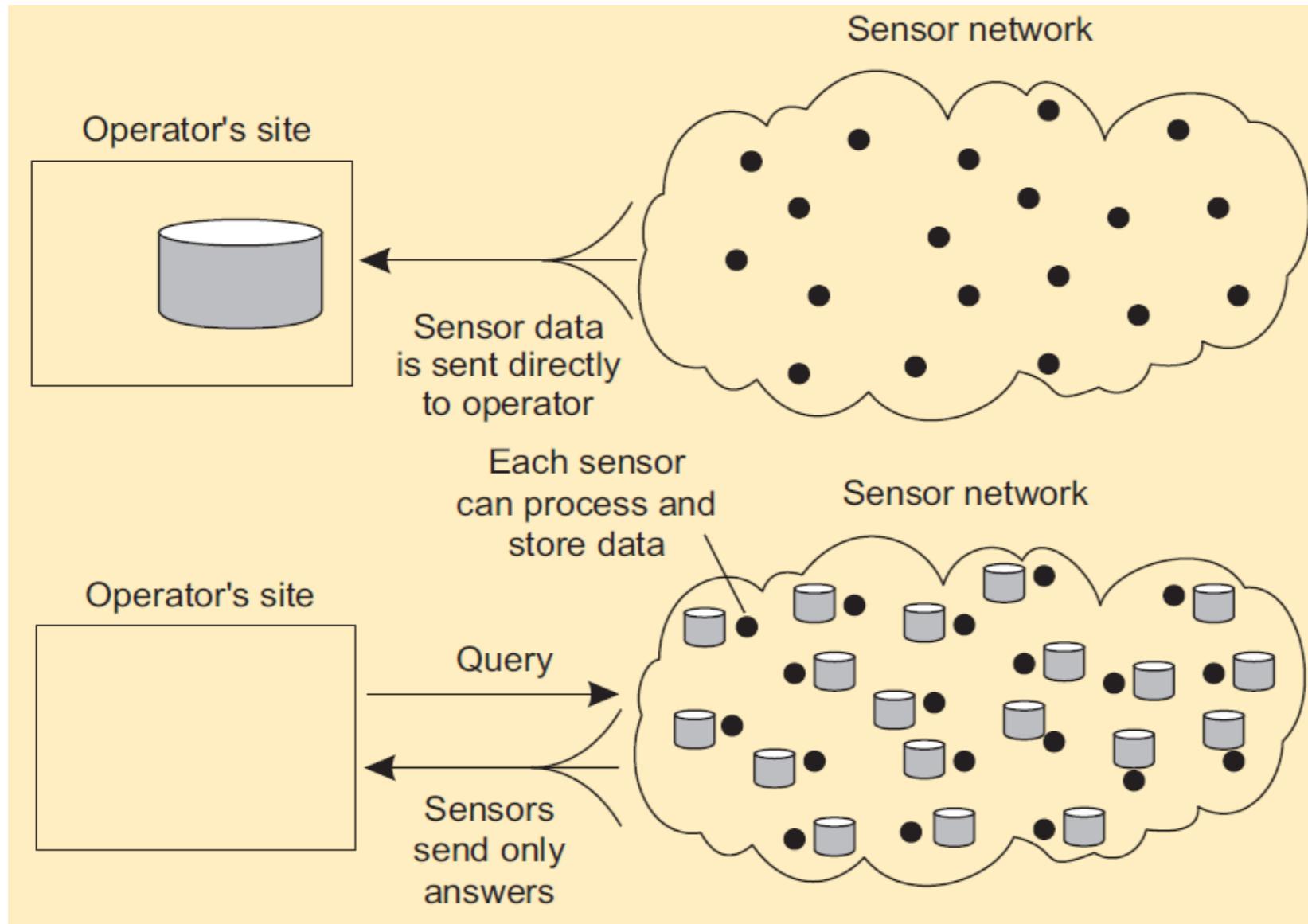
Os nós aos quais sensores estão conectados são:

- Muitos (10s a 1000s)
- Simples (baixa capacidade de memória / computação / comunicação)
- Frequentemente alimentados por bateria (ou mesmo sem bateria)

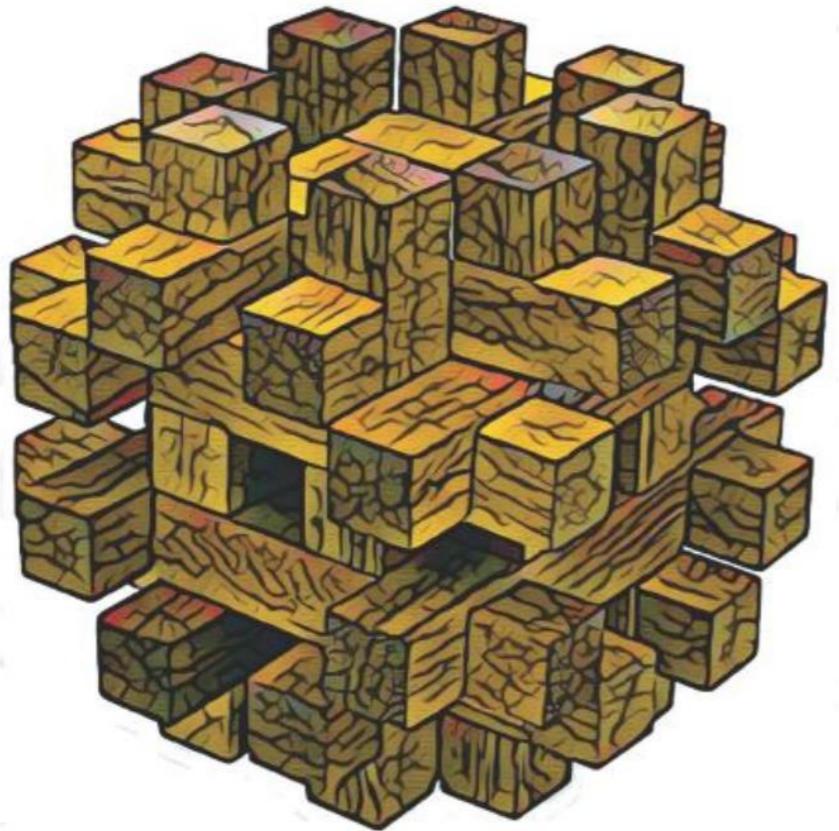


REDES DE SENsoRES COMO BASE DE DADOS DISTRIBuíDA

Dois extremos



CAPÍTULO 2 - ARQUITETURAS



Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum

3th Edition – Version 3.01 - 2017

ESTILOS ARQUITETURAIS

IDEIA BÁSICA

Um estilo é formulado em termos de:

- Componentes (trocáveis) com interfaces bem definidas
- A forma na qual componentes são conectados entre si
- Os dados trocados entre componentes
- Como estes componentes e conectores são agrupados e configurados em um sistema

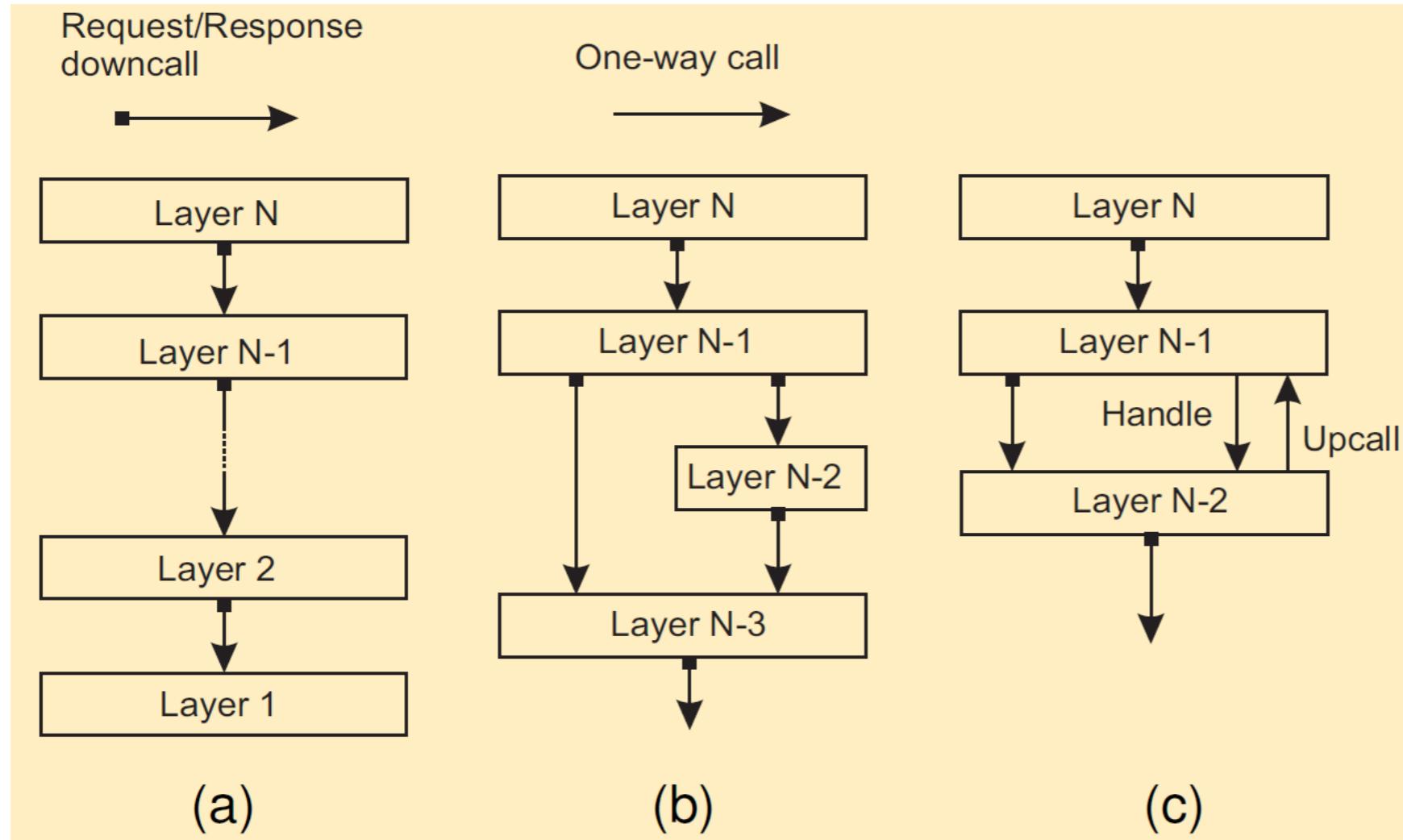
CONECTOR

Um mecanismo que intermedia a comunicação, coordenação, ou cooperação entre componentes. **Exemplos:** facilidades para RPC (remote procedure call), mensagens, ou streaming



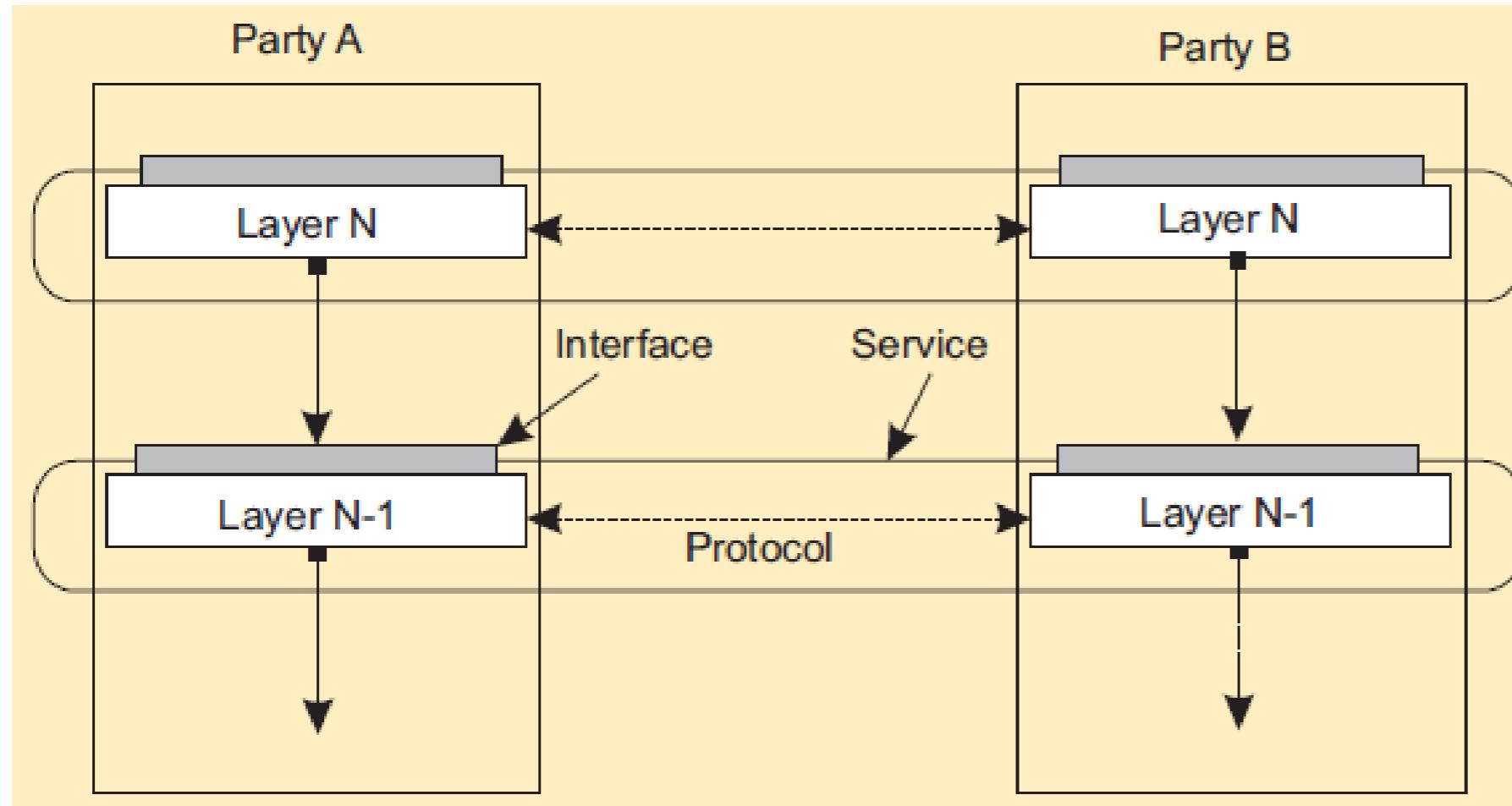
ARQUITETURA EM CAMADAS

DIFERENTES ORGANIZAÇÕES EM CAMADAS



EXEMPLO PROTOCOLOS DE COMUNICAÇÃO

PROTOCOLO, SERVIÇO, INTERFACE



COMUNICAÇÃO COM DOIS PARTICIPANTES

SERVIDOR

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept()      # returns new socket and addr. client
4 while True:                   # forever
5     data = conn.recv(1024)      # receive data from client
6     if not data: break         # stop if client stopped
7     conn.send(str(data)+"*")   # return sent data plus an "*"
8 conn.close()                  # close the connection
```

CLIENTE

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT))        # connect to server (block until accepted)
4 s.send('Hello, world')         # send some data
5 data = s.recv(1024)            # receive the response
6 print data                    # print the result
7 s.close()                     # close the connection
```



CAMADAS DE APLICAÇÃO

VISÃO TRADICIONAL EM TRÊS CAMADAS

- **Camada Interface de aplicação:** contém unidades para interfaceamento de usuários ou aplicações externas
- **Camada de processamento:** contém as funções da aplicação, i.e., sem dados específicos
- **Camada de Dados:** contém os dados que o cliente deseja para manipular através de componentes de aplicação

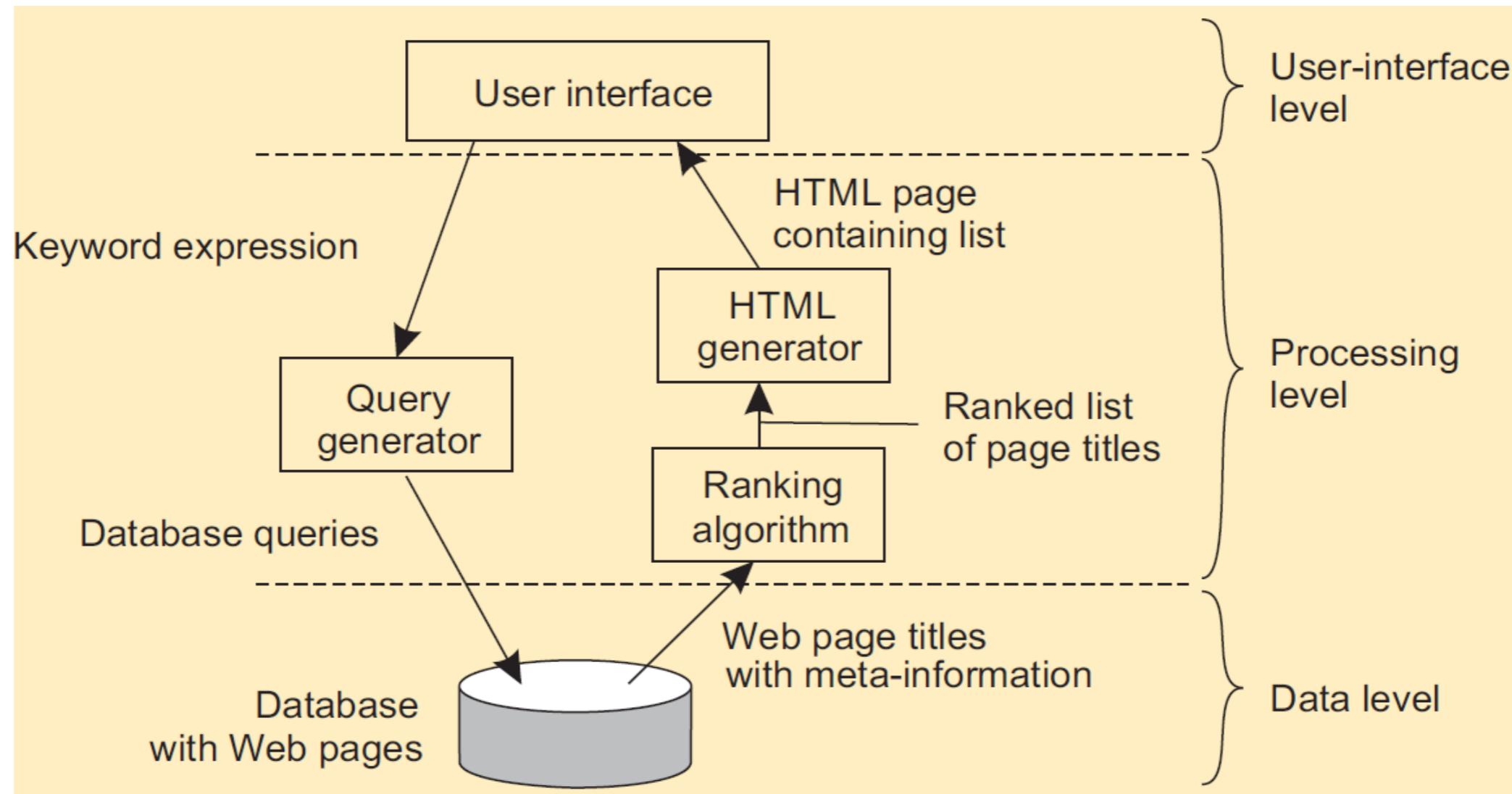
OBSERVAÇÃO

Estas camadas são encontradas em muitos sistemas de informação distribuídos, usando tecnologias de base de dados e aplicações de suporte



CAMADAS APLICAÇÃO

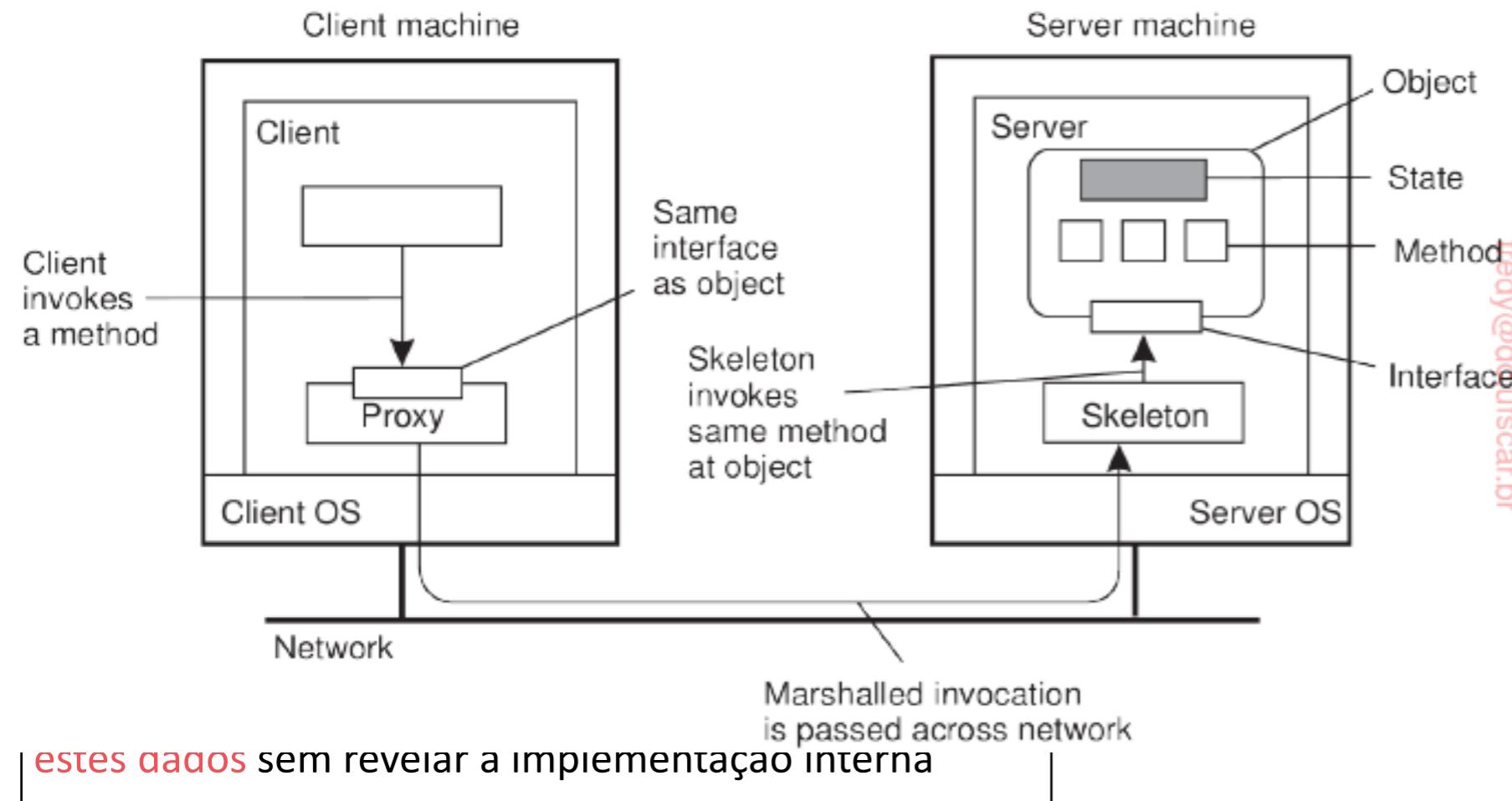
EXEMPLO: UM MOTOR DE BUSCAS SIMPLES



ESTILO BASEADO EM OBJETOS

ESSÊNCIA

Componentes são objetos conectados uns aos outros através de chamadas de procedimento. Objetos podem estar em diferentes máquinas; chamada pode então ser executada através da rede



ARQUITETURAS RESTful (REST – Representational State Transfer)

ESSÊNCIA (web based resources)

Visualize um sistema distribuído como uma coleção de recursos gerenciados individualmente por componentes. Recursos podem ser adicionados, removidos, recuperados e modificados por aplicações (remotas).

- 1) Recursos são identificados através de um único esquema de nomes
- 2) Todos serviços oferecem a mesma interface
- 3) Mensagens enviadas para ou provenientes de um serviço são totalmente auto-descritivas
- 4) Depois de executar uma operação em um serviço, o componente esquece tudo sobre o chamante ([Stateless execution](#))

ENCAPSULAMENTO

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

EXEMPLO AMAZON SIMPLE STORAGE SERVICE

ESSÊNCIA

Objetos (i.e.arquivos) são colocados em buckets (i.e. diretórios). Buckets não podem ser colocados dentro de buckets. Operações ObjectName no bucket BucketName requerem o seguinte identificador:

`http://BucketName.s3.amazonaws.com/ObjectName`

OPERAÇÕES TÍPICAS

Todas operações são realizadas enviando requisições HTTP:

- Criar um bucket/object: PUT, junto com o URI
- Listar objetos: GET em um nome de bucket
- Ler de um objeto: GET em um URI completo



SOBRE INTERFACES

QUESTÃO

Muitos usuários gostam da abordagem RESTful porque a interface de acesso ao serviço é bem simples. O ponto é que há a necessidade de trabalhar no [espaço de parametrização](#).

Amazon S3 Interface SOAP (Simple Object Access Protocol)

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy



SOBRE INTERFACES

SIMPLIFICAÇÕES

Assuma uma interface bucket oferecendo uma operação create, que demanda como entrada um string tal como mybucket, para criação do bucket “mybucket”.

SOAP

```
import bucket  
bucket.create("mybucket")
```

RESTful

```
PUT "http://mybucket.s3.amazonaws.com/"
```

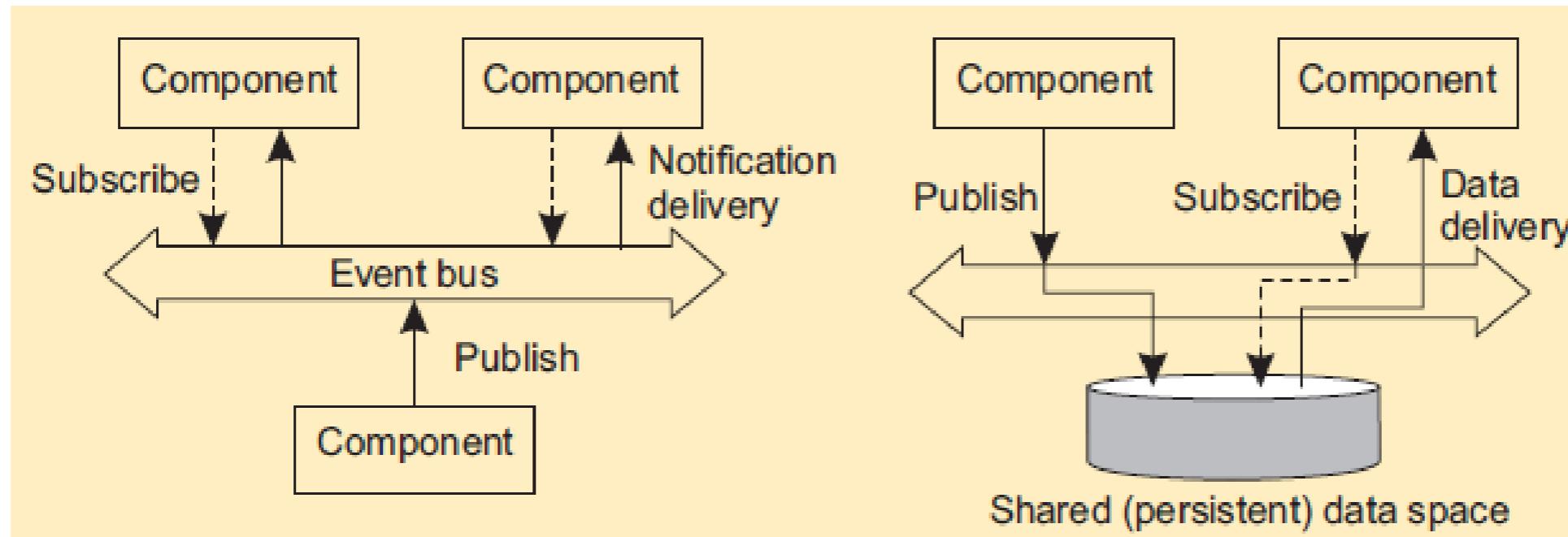


COORDENAÇÃO PUBLISH-SUBSCRIBE

CASAMENTO TEMPORAL E REFERENCIAL

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

ESPAÇO DE DADOS – BASEADO EM EVENTOS E COMPARTILHADO



LINDA TUPLE SPACE (seq obj não mutáveis)

TRÊS OPERAÇÕES SIMPLES

- $in(t)$: remove o template de acoplamento da tuple t
- $rd(t)$: obtém uma cópia do template da tuple t
- $out(t)$: adiciona a tuple t ao espaço de tuple

MAIS DETALHES

- Chamar $out(t)$ duas vezes seguidas, leva ao armazenamento de duas cópias da tuple $t \Rightarrow a$ e a tuple space é modelada como um **multiset**.
- Ambos in e rd são operações **bloqueantes**: o chamante ficará bloqueado até que a tuple correspondente seja achada, ou se torne disponível



LINDA TUPLE SPACE (seq obj não mutáveis)

BOB

```
1 blog = linda.universe._rd( ("MicroBlog",linda.TupleSpace) ) [1]
2
3 blog._out( ("bob", "distsys", "I am studying chap 2") )
4 blog._out( ("bob", "distsys", "The linda example's pretty simple") )
5 blog._out( ("bob", "gtcn", "Cool book!") )
```

ALICE

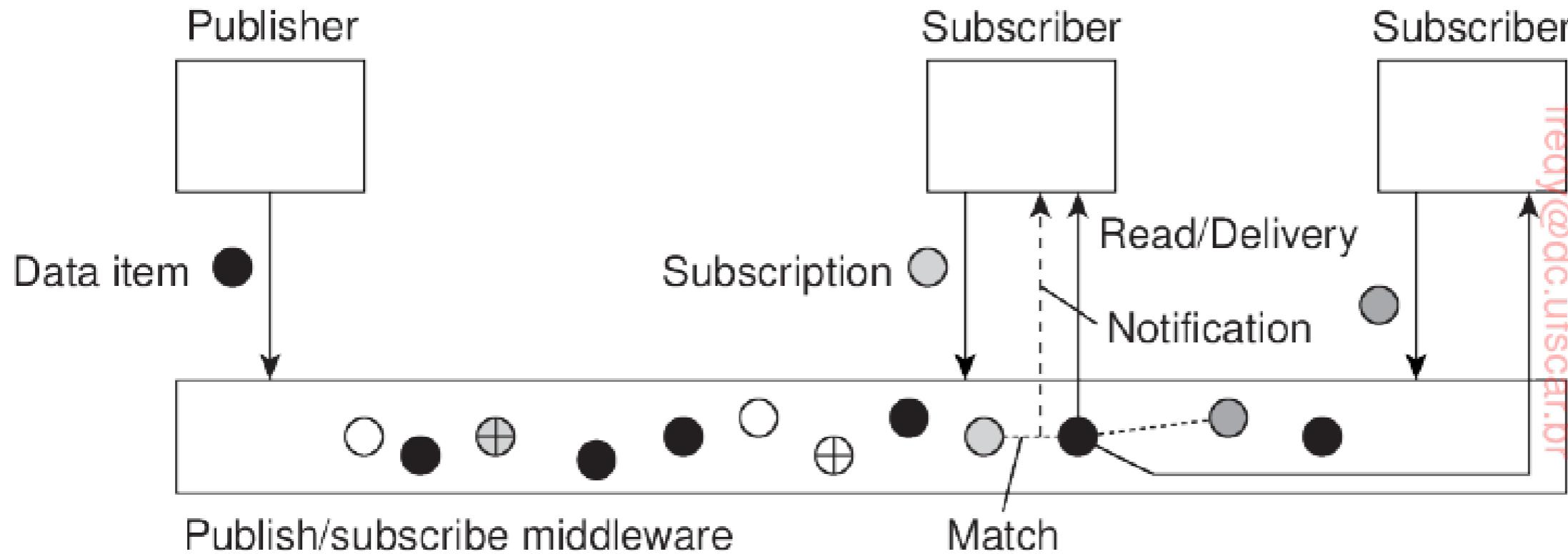
```
1 blog = linda.universe._rd( ("MicroBlog",linda.TupleSpace) ) [1]
2
3 blog._out( ("alice", "gtcn", "This graph theory stuff is not easy") )
4 blog._out( ("alice", "distsys", "I like systems more than graphs") )
```

CHUCK

```
1 blog = linda.universe._rd( ("MicroBlog",linda.TupleSpace) ) [1]
2
3 t1 = blog._rd( ("bob", "distsys", str) )
4 t2 = blog._rd( ("alice", "gtcn", str) )
5 t3 = blog._rd( ("bob", "gtcn", str) )
```



Troca de dados – publish/subscribe



USANDO LEGADO PARA CONSTRUIR MIDDLEWARE

PROBLEMA

A interface oferecida por um componente legado provavelmente não será adequada para todas aplicações.

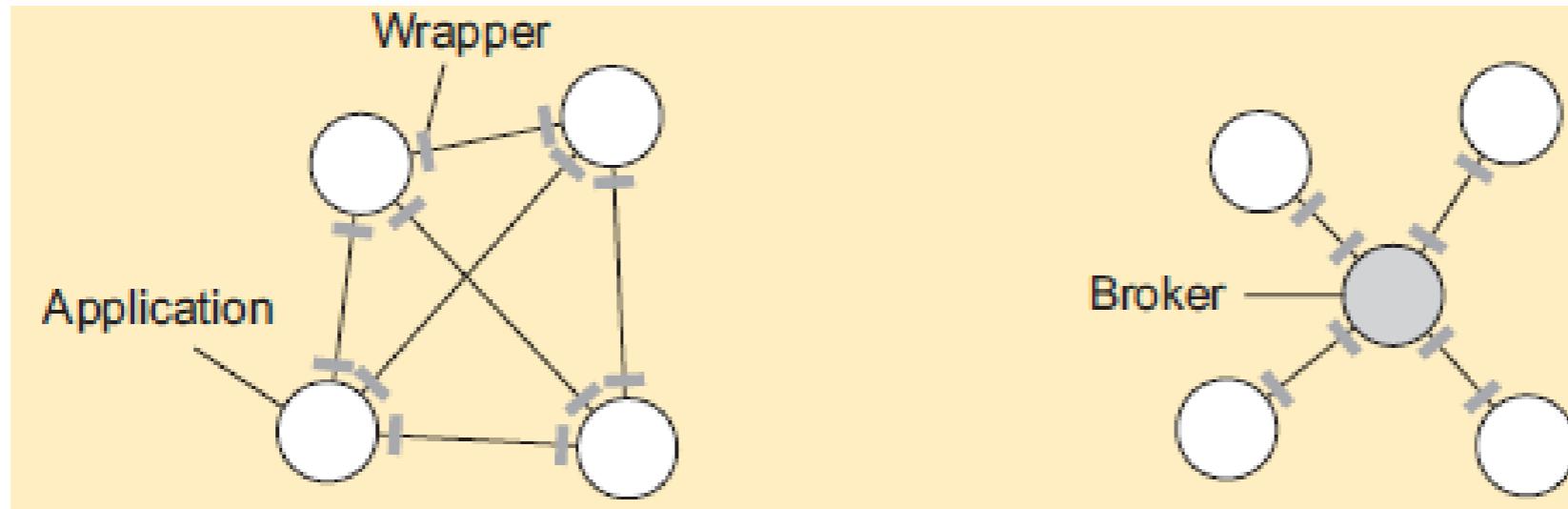
SOLUÇÃO

Um **wrapper** ou **adaptador** oferece uma interface aceitável para o cliente de aplicação. Suas funções são transformadas naquelas disponíveis no componente



ORGANIZANDO WRAPPERS

DUAS SOLUÇÕES: 1 EM 1 OU ATRAVÉS DE UM **BROKER**



COMPLEXIDADE COM N APLICAÇÕES

- Um em um: requer $N \times (N - 1) = \mathcal{O}(N^2)$ wrappers
- Broker: requer $2N = \mathcal{O}(N)$ wrappers

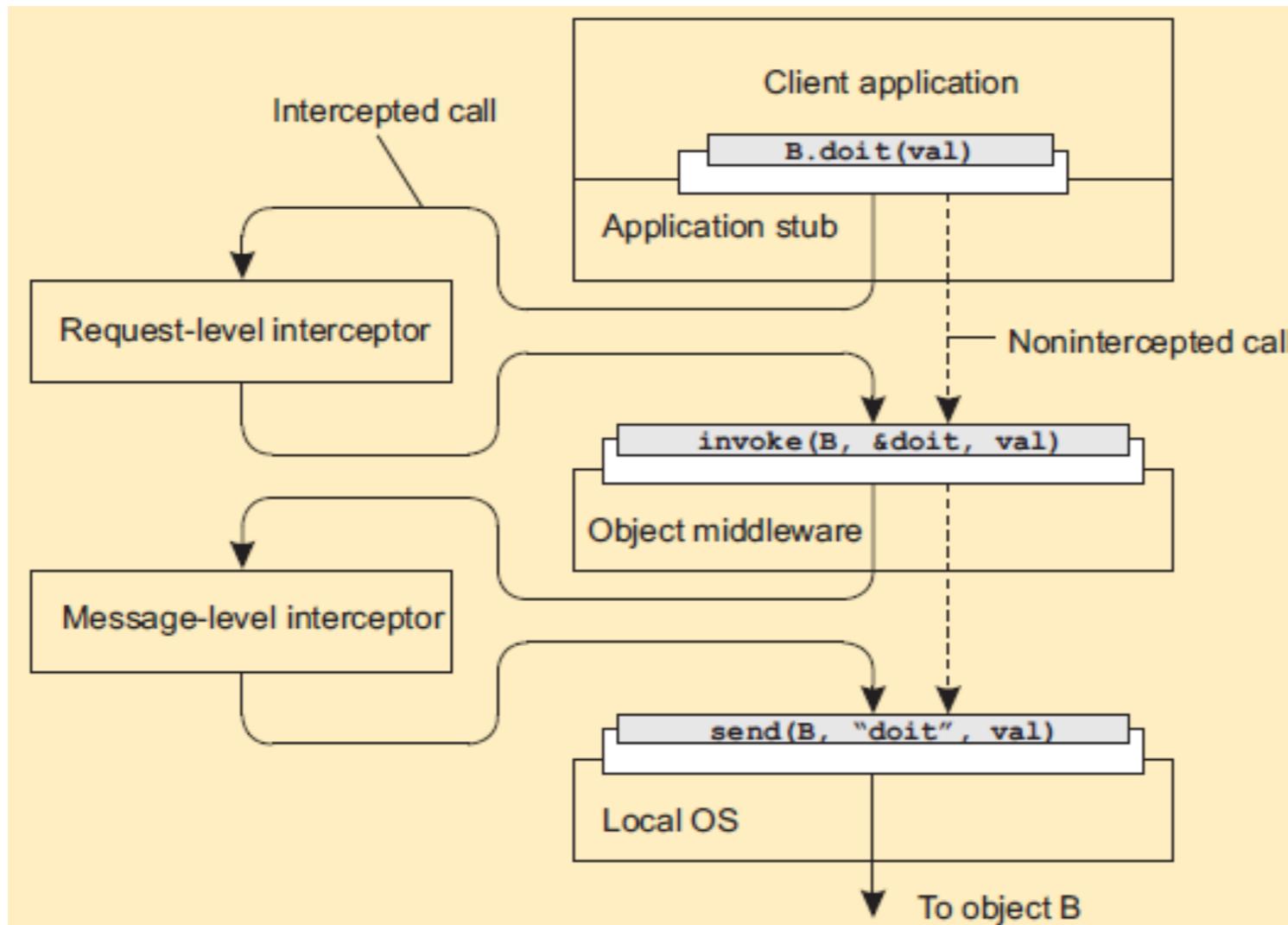
DESENVOLVENDO MIDDLEWARE ADAPTÁVEL

PROBLEMA

- Middleware contém soluções que são boas para a **maioria** das aplicações: possivelmente deseja-se que o comportamento seja adaptável para aplicações específicas.



INTERCEPTANDO O CONTROLE DE FLUXO USUAL

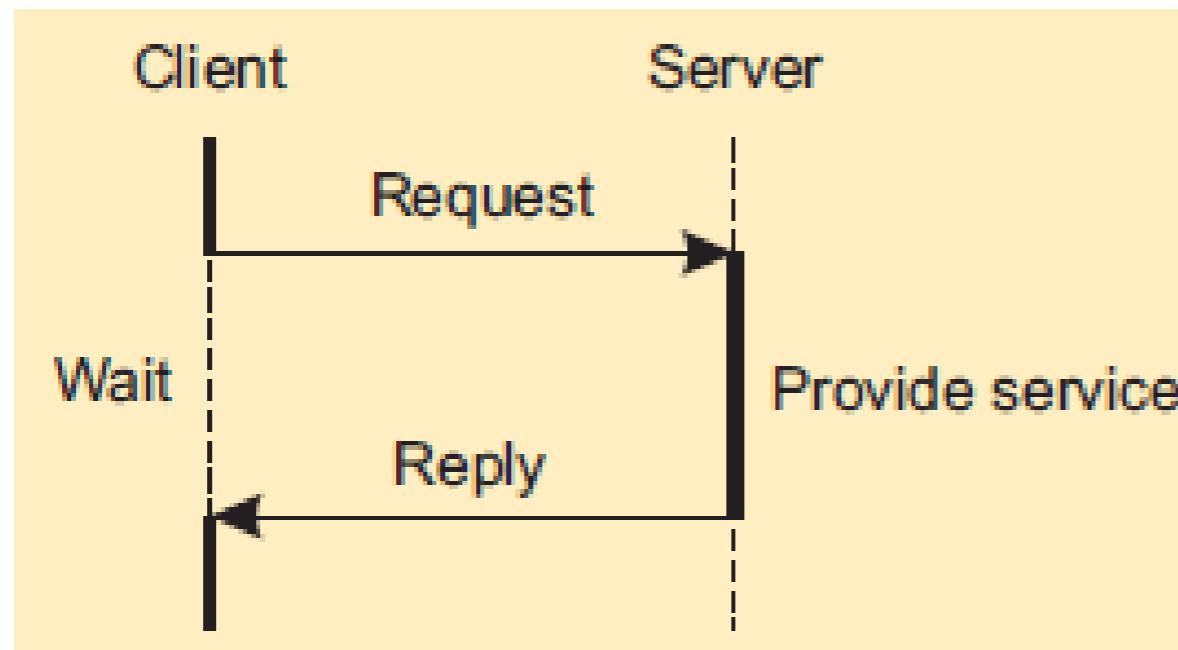


ARQUITETURA CENTRALIZADA DE SISTEMAS

MODELO BÁSICO CLIENTE SERVIDOR

Características

- Existe processos oferecendo serviços (**servidores**)
- Existe processos que usam serviços (**clientes**)
- Clientes e servidores podem estar em diferente máquinas
- Clientes seguem o modelos request/reply com respeito ao uso de serviços

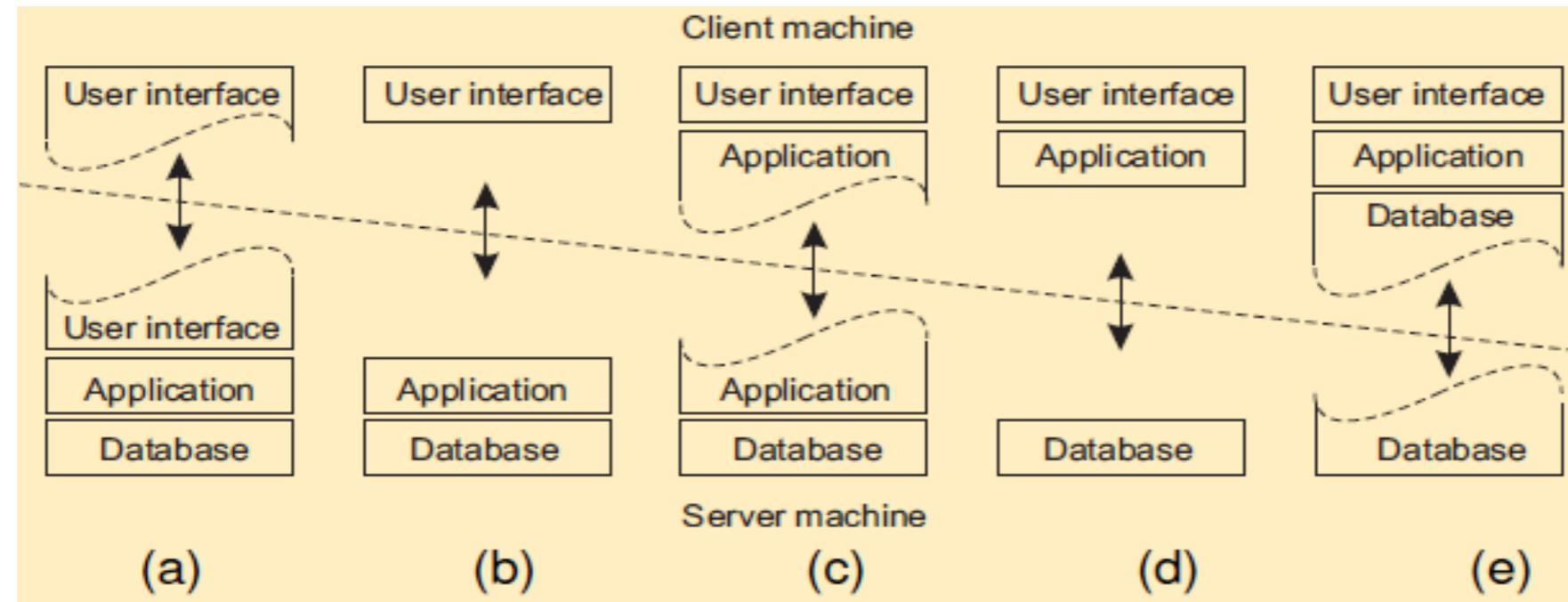


ARQUITETURA CENTRALIZADA DE SISTEMAS MULTI-TIERED

ALGUMAS ORGANIZAÇÕES TRADICIONAIS

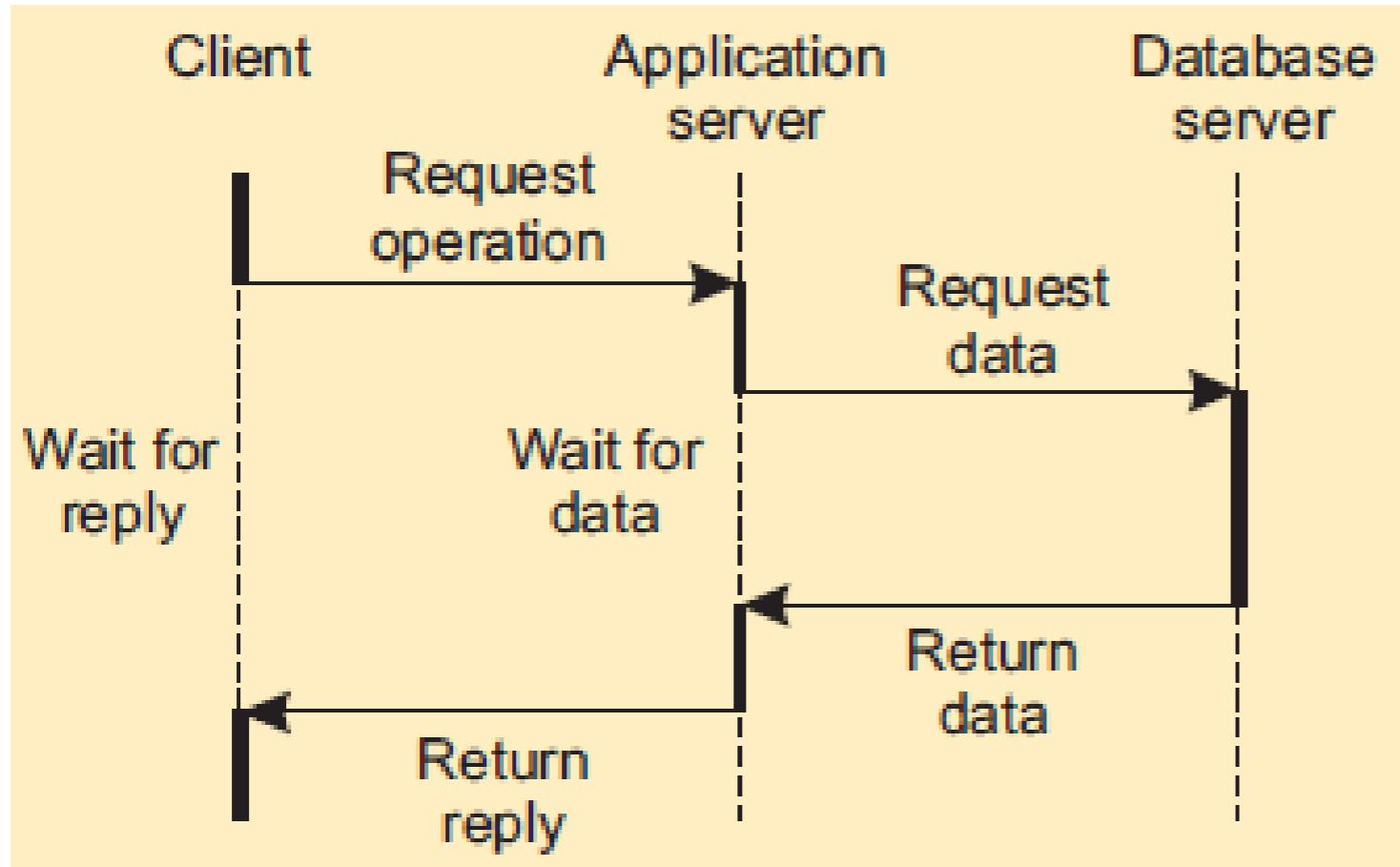
- Single-tiered: terminal burro / configuração mainframe
- Two-tiered: configuração cliente servidor de servidor único)
- Three-tiered: cada camada em uma máquina separada

CONFIGURAÇÃO TRADICIONAL EM DUAS CAMADAS (TWO TIERED)



SENDO CLIENTE E SERVIDOR AO MESMO TEMPO

ARQUITETURA EM TRÊS CAMADAS (THREE TIERED)



ORGANIZAÇÕES ALTERNATIVAS

DISTRIBUIÇÃO VERTICAL

Vem da divisão de aplicações distribuídas em três camadas lógicas, e os componentes rodam em cada camada em uma máquina (servidor) diferente.

DISTRIBUIÇÃO HORIZONTAL

Um cliente ou servidor pode ser fisicamente dividido em partes logicamente equivalentes, mas cada parte esta operando em sua porção compartilhada de dados de um conjunto completo de dados

ARQUITETURAS PEER-TO-PEER

Processos são todos iguais: as funções que precisam ser executadas são representadas por todos processos -> cada processo vai se comportar como cliente e como servidor ao mesmo tempo (i.e. agindo como um servo)



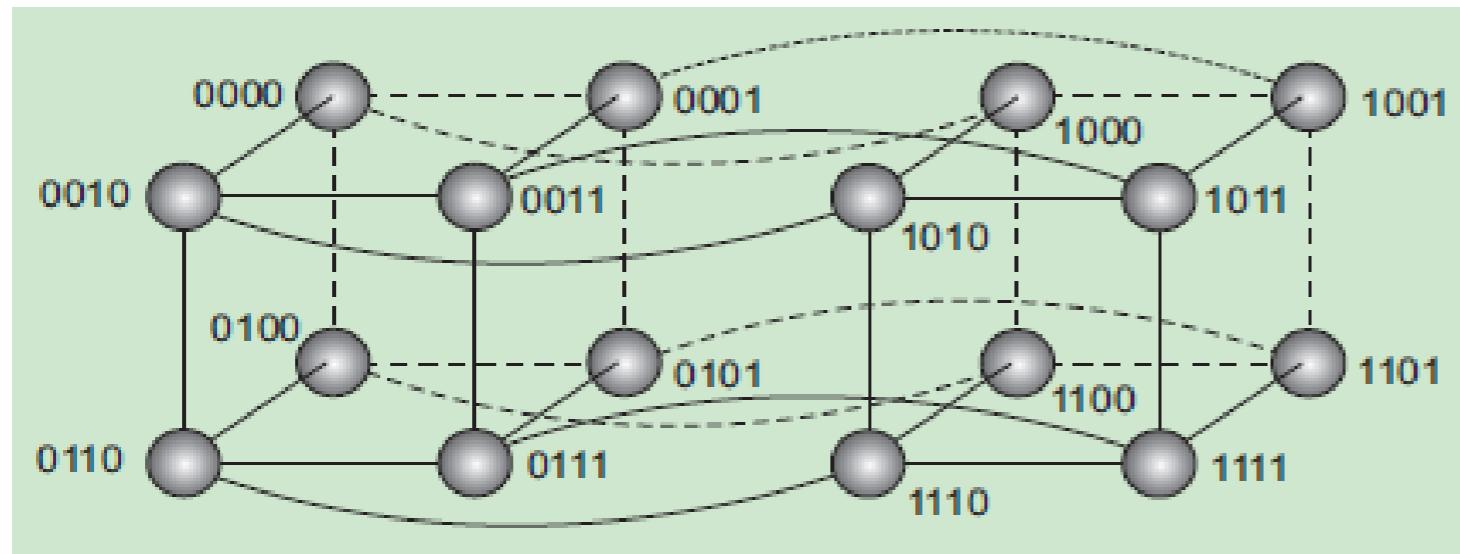
P2P ESTRUTURADO

ESSÊNCIA

Faça uso de um índice livre de semântica: cada item de dado é associado unicamente com uma chave, que será usada como índice. Prática comum: uso de uma **função hash** $key(data\ item) = hash(valor\ do\ item\ de\ dado)$

Sistema P2P agora é responsável por armazenar pares (chave, valor)

EXEMPLO SIMPLES: HIPERCUBO



Procurando (looking up) d com **chave** $k \in \{0, 1, 2, \dots, 2^4 - 1\}$ significa o **roteamento** da requisição para o nó com **identificador** k

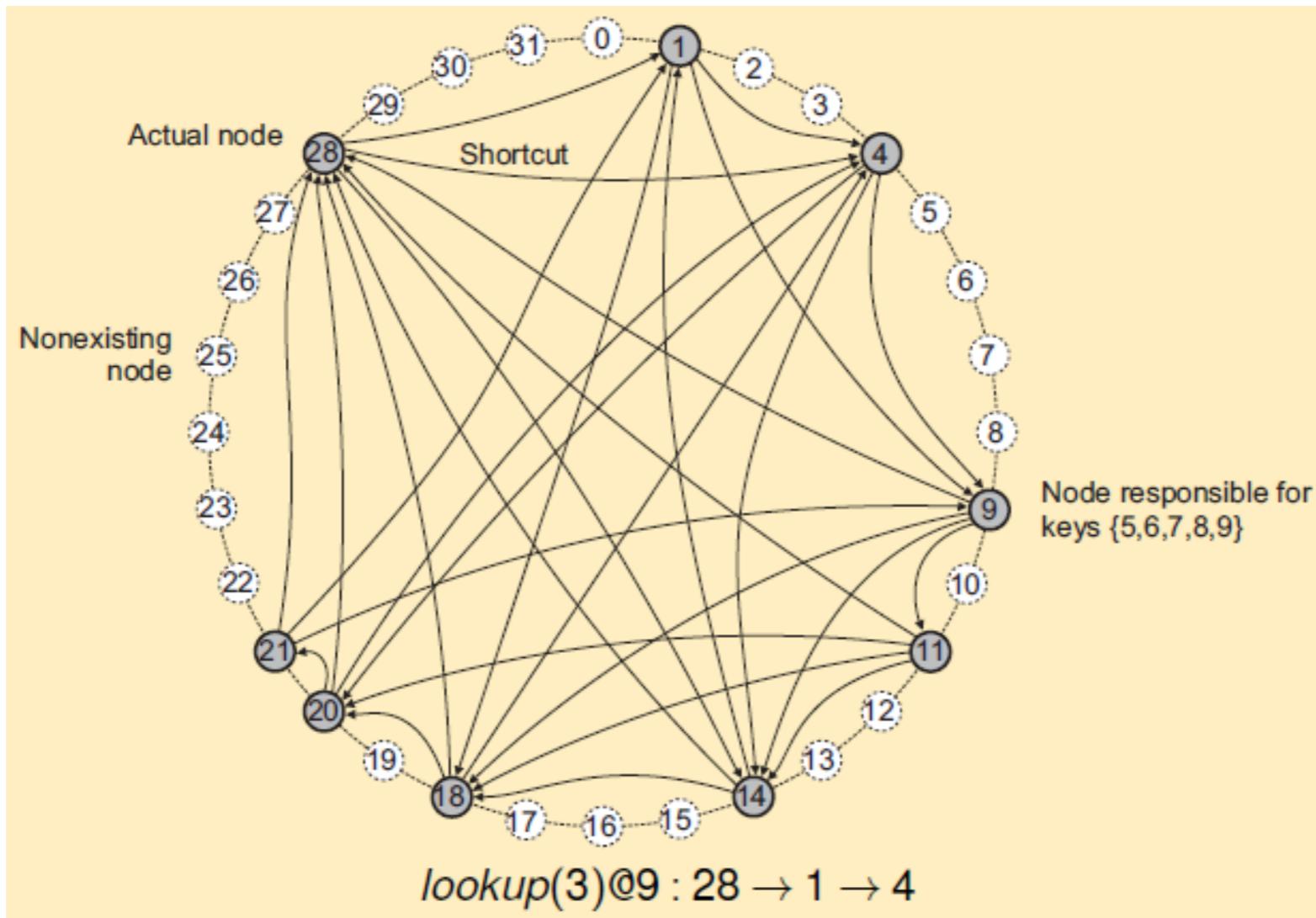
EXEMPLO CHORD

PRINCÍPIO

- Nós são logicamente organizados como um anel. Cada nó tem um **identificador** m -bit
- Cada item de dados é “*hashed*” para uma **chave** m -bit
- Item de dados com chave k é armazenado em um nó com o menor identificador $id \geq k$, chamado de **sucessor** da chave k .
- O anel é extendido com vários **links atalho** para outros nós.



EXEMPLO CHORD



P2P DESESTRUTURADO

ESSÊNCIA

Cada nó mantém uma lista ad hoc de vizinhos. O overlay resultante se parece com um **grafo randômico**: uma borda $\langle u, v \rangle$ existe somente com uma certa probabilidade $P[\langle u, v \rangle]$.

BUSCANDO

- **Flooding**: nó emissor u passa uma requisição de d para todos vizinhos. A requisição é ignorada quando um nó recebedor já tenha a visto anteriormente. De outra forma, v procura localmente por d (recursivamente). Pode ser limitado por **Time-To-Live**: um número máximo de hops.
- **Caminhada (walk) randômica**: o nó emissor u passa a requisição d para um vizinho escolhido randomicamente. Se v não tem d , ele repassa a requisição para um de seus vizinhos escolhido randomicamente, e assim por diante.

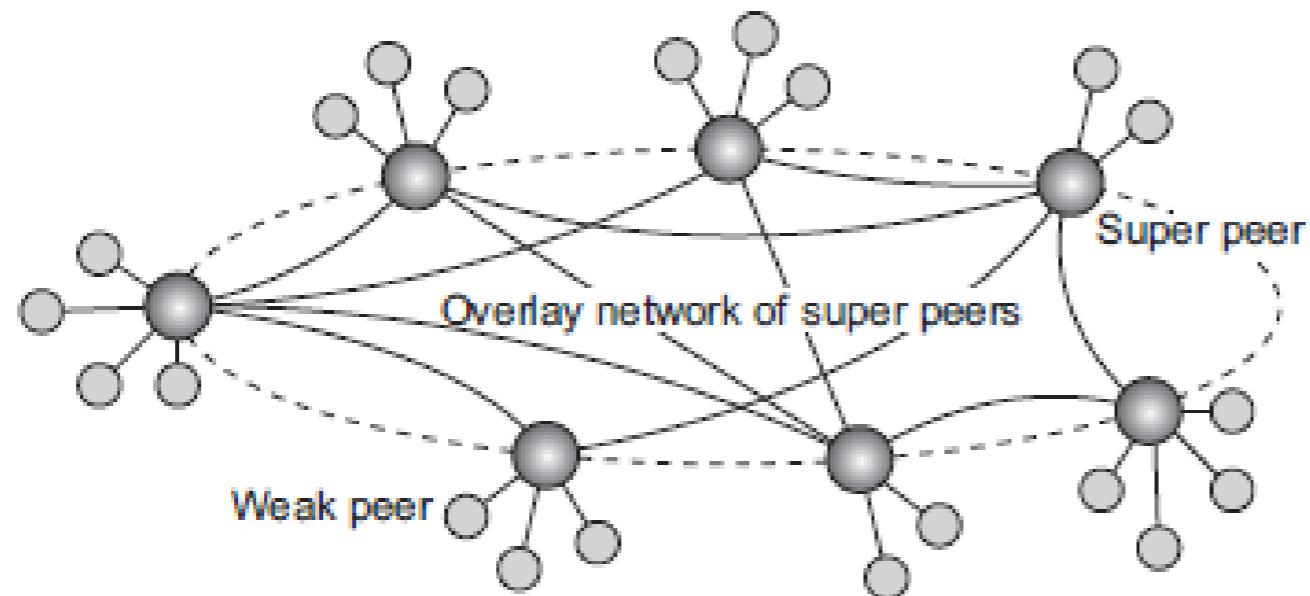


REDES SUPER PEERS

ESSÊNCIA

As vezes é sensato quebrar a simetria em redes puras peer-to-peer

- Quando buscando em sistemas P2P não estruturados, a existênciade **servidores de índice** melhora o desempenho
- Decidir onde armazenar dados pode ser frequentemente feito de forma mais eficiente através de **brokers**.



OPERAÇÃO PRINCIPAL DO SKYPE: A QUER CONTATAR B

AMBOS A E B ESTÃO EM DOMÍNIO PÚBLICO NA INTERNET

- Uma conexão TCP é feita entre A e B para controlar pacotes
- A chamada atual ocorre usando pacotes UDP entre portos negociados

A OPERA ATRÁS DE UM FIREWALL, ENQUANTO B ESTÁ PÚBLICO

- A faz uma conexão TCP (para controlar pacotes) com um super peer S
- S faz uma conexão TCP (para repassar pacotes de controle) para B
- A chamada atual acontece através de UDP e diretamente entre A e B

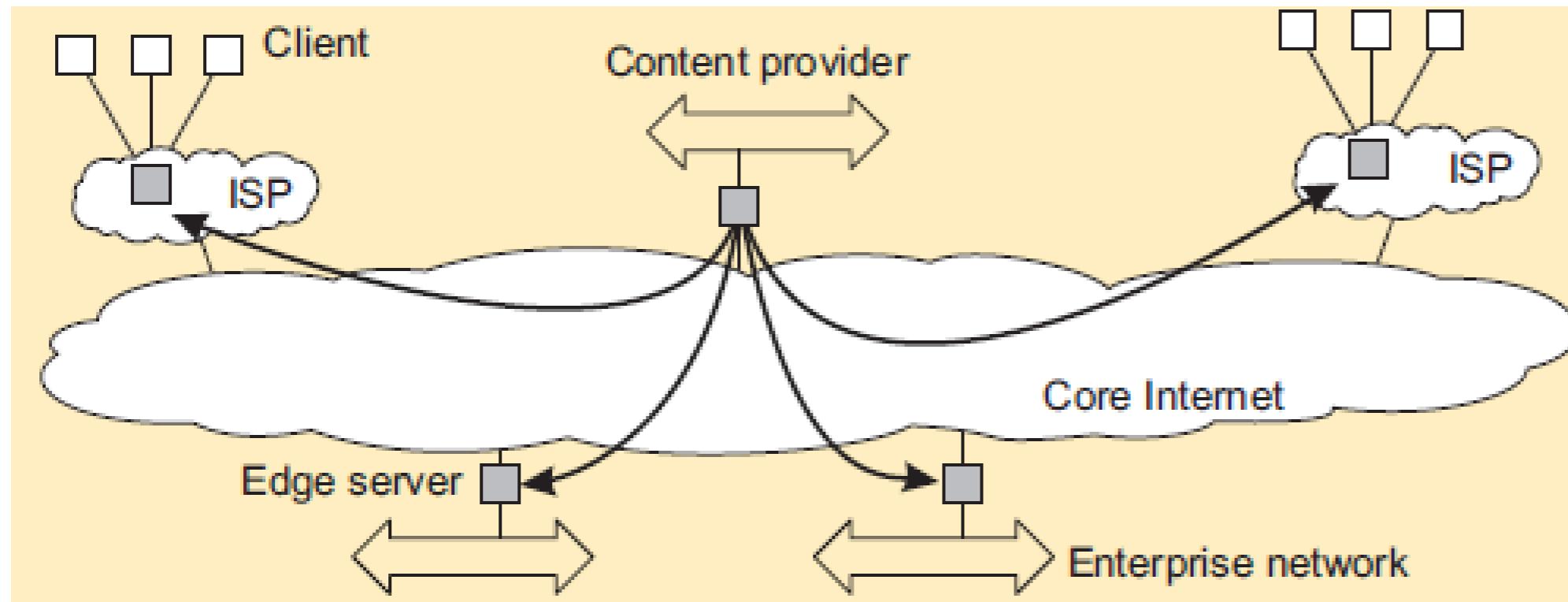
AMBOS A E B OPERAM ATRÁS DE UM FIREWALL

- A conecta com um super peer S usando TCP
- S faz uma conexão TCP com B
- Para a chamada atual, outro super peer é contatado para agir como repassador (**relay**) R: A faz uma conexão com R e B também.
- Todo tráfego de voz é repassado sobre duas conexões TCP, e através de R

ARQUITETURA EDGE-SERVER

ESSÊNCIA

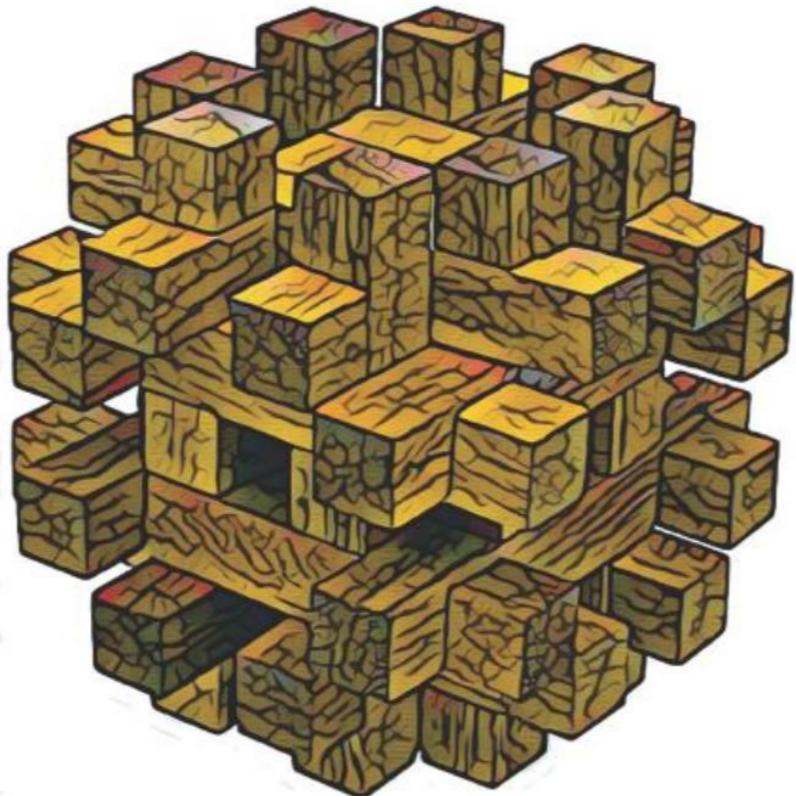
Sistemas disseminados na Internet onde os servidores são colocados na borda (*at the edge*) da rede: a fronteira entre redes empresariais e a Internet atual



CAPÍTULO 3 - PROCESSOS

Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



3th Edition – Version 3.01 - 2017

INTRODUÇÃO AOS THREADS

IDÉIA BÁSICA

Construção de processadores virtuais em software, no topo de processadores físicos:

- **Processador:** Provê um conjunto de instruções junto com a capacidade de executar automaticamente uma série destas instruções.
- **Thread:** Um processador de software mínimo em cujo contexto uma série de instruções podem ser executadas. Salvar o contexto de uma **thread** implica em parar a execução atual e salvar todos os dados em uso, de forma que seja possível retornar ao mesmo estágio em um momento seguinte.
- **Processo:** Um processador de software em cujo contexto um ou mais **threads** podem ser executadas. Executar uma **thread** significa executar uma série de instruções em um contexto daquela **thread**.



CHAVEAMENTO DE CONTEXTO

CONTEXTOS

- **Contexto de Processador:** Uma coleção mínima de valores armazenados nos registradores de um processador usados para execução de uma série de instruções (p.e., stack pointer, endereço de registradores, contador de programa, etc).
- **Contexto de Thread :** Uma coleção de valores armazenados em registradores e memória, usados na execução de uma série de instruções (isto é, contexto de processador, estados).
- **Contexto de Processo:** Uma coleção mínima de valores armazenados em registradores e memória, usados para execução de uma thread (isto é., contexto de thread, mas também pelo menos os valores dos registradores da MMU).



CHAVEAMENTO DE CONTEXTO

OBSERVAÇÕES

1. Threads dividem o mesmo espaço de endereçamento. O chaveamento de contexto de **Thread** pode ser feito de forma totalmente independente do sistema operacional.
2. Chavamento de processo é geralmente (de alguma forma) mais custoso pois envolve o SO, **trapping** para o **kernel** ...
3. A criação e destruição de uma thread é muito menos custosa do que fazer o mesmo para um processo.

POR QUE USAR THREADS

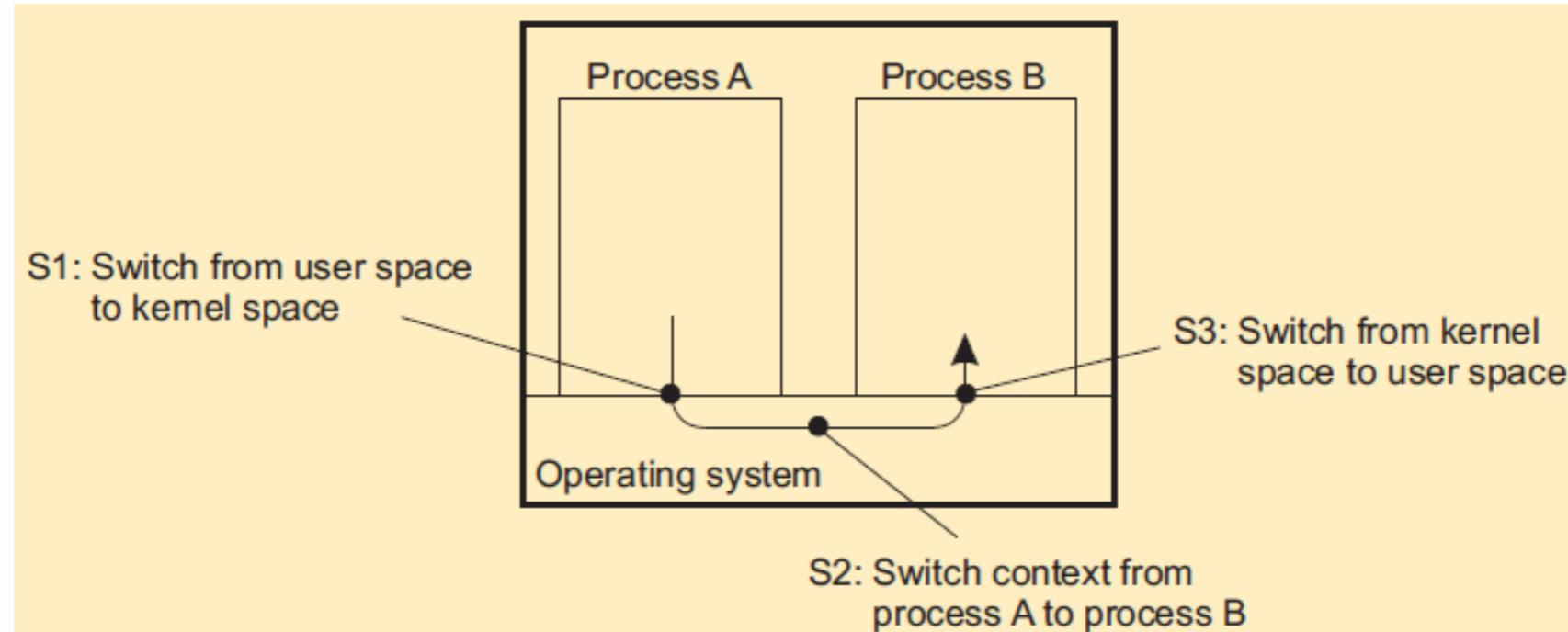
ALGUMAS RAZÕES SIMPLES

- **Evitar bloqueamentos desnecessários:** um processo **single-threaded** vai bloquear quando estiver fazendo E/S; em um processo **multi-threaded**, o Sistema Operacional pode chavear a CPU para outro thread no processo.
- **Explorar paralelismo:** os threads de um processo **multi-threaded** podem ser escalonados para rodar em paralelo em um multi-processador ou processador **multi-core**.
- **Evitar chaveamento de processo:** estruture a aplicação não como uma coleção de processos, mas através de múltiplos **threads**.



EVITAR CHAVEAMENTO DE PROCESSOS

EVITAR TROCA DE CONTEXTO CUSTOSA ..



TRADE-OFFS (compromissos)

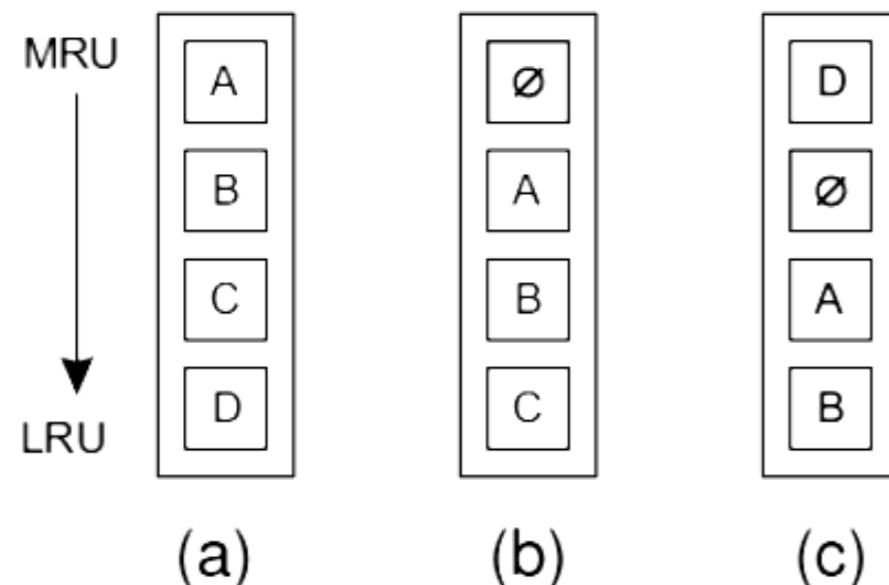
- **Threads** usam o mesmo espaço de endereçamento: mais propenso a erros
- Nenhum suporte do SO/HW para prevenir que as **threads** usem memória uma das outras
- Chaveamento de contexto de **thread** pode ser mais rápido que o chaveamento de contexto de processo.

CUSTO DE TROCA DE CONTEXTO

CONSIDERE UM HANDLER SIMPLES MANUSEADOR DE INTERRUPÇÃO DE RELÓGIO

- **Custos diretos:** chaveamento em si e a execução do código do **handler**
- **Custos indiretos:** outros custos, notadamente causados pela bagunça na **cache**

O QUE UMA TROCA DE CONTEXTO PODE CAUSAR: CUSTOS INDIRETOS



- Antes do chaveamento de contexto
- Depois do chaveamento de contexto
- Depois de acessar o bloco D

THREADS E SISTEMAS OPERACIONAIS

QUESTÃO PRINCIPAL

- Deveria o kernel de SO prover suporte a threads, ou este suporte deve ser implementado em pacotes de nível usuário?

O QUE UMA TROCA DE CONTEXTO PODE CAUSAR: CUSTOS INDIRETOS

- Todos processos podem ser completamente manuseados dentro de um único processo → implementações podem ser extremamente eficientes.
- Todos serviços providos pelo kernel são feitos em nome do processo no qual a thread reside → se um kernel decide bloquear uma thread, então o processo todos será bloqueado.
- Threads são usadas quando existe muitos eventos externos: threads bloqueiam com base em eventos → se um kernel não pode distinguir threads, como ele pode suportar a sinalização de eventos para eles ?



THREADS E SISTEMAS OPERACIONAIS

SOLUÇÃO NO KERNEL

A ideia completa é implementar o pacote de suporte a threads dentro do kernel. Isto significa que **todas** operações retornam como chamadas de sistema:

- Operações que bloqueiam uma thread não são mais um problema: o **kernel escalona outra thread disponível** dentro do mesmo processo.
- O manuseio de eventos externos é simples: o kernel (que pega todos eventos) escalona uma thread associada com o evento.
- O problema é (ou era) a falta de eficiência devido ao fato que cada operação na thread requer um trap para o kernel.

CONCLUSÃO – MAS ..

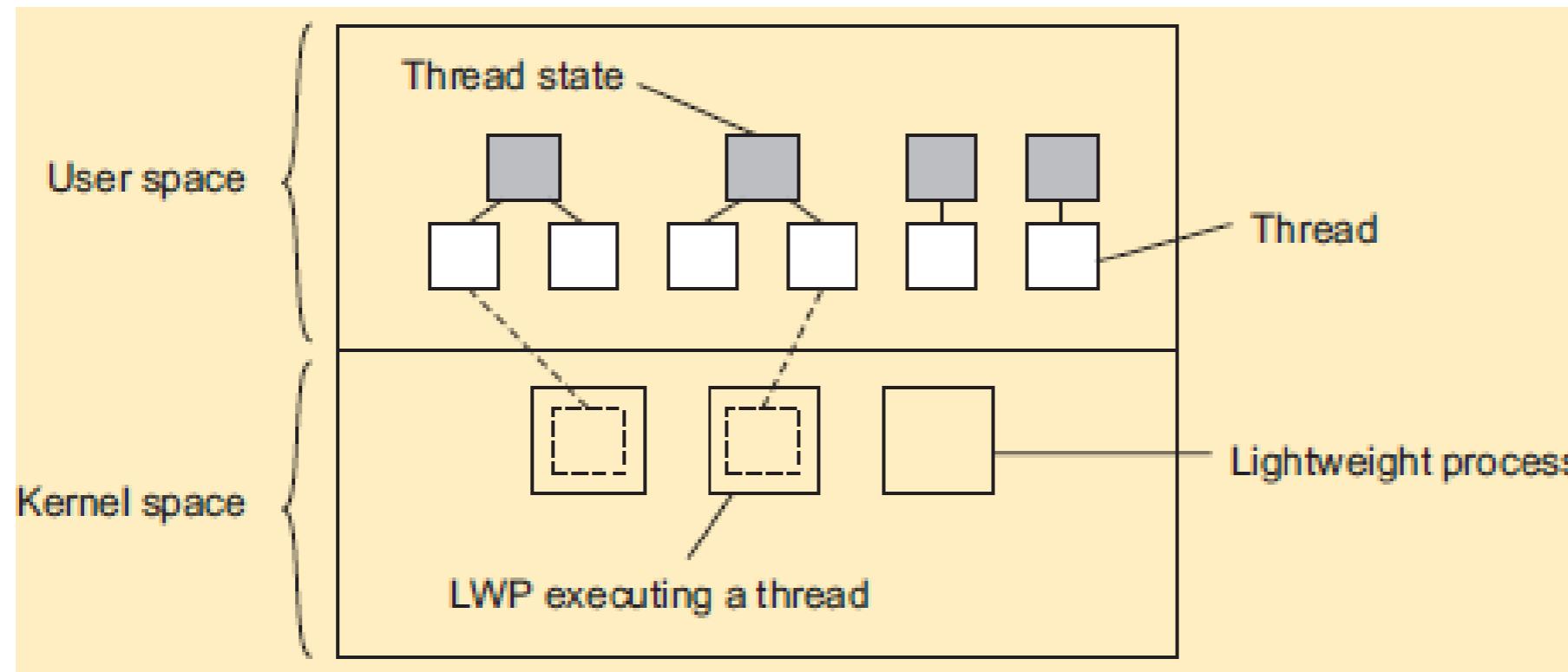
- Tente uma mistura de threads nível usuário e nível kernel em um mesmo conceito, contanto que o ganho de desempenho não implique em aumento da complexidade.



PROCESSOS LIGHTWEIGHT

IDEIA BÁSICA

Introduzir uma abordagem de threads de dois níveis: **lightweight processes** que podem executar threads nível usuário.



PROCESSOS LIGHTWEIGHT

PRINCÍPIO OPERACIONAL

- Uma thread nível usuário faz uma system call → o LWP que está executando esta thread, **bloqueia**. A thread se mantém presa (**bound**) ao LWP.
- O kernel pode **escalonar outro LWP mesmo tendo uma thread presa a ele**. Nota: esta **thread** pode chavear para **qualquer** outra **thread** rodável que esteja no espaço de usuário.
- Se uma thread realiza uma chamada de operação bloqueante em nível usuário → ocorre chaveamento de contexto para outra **thread** rodável, (presa ao mesmo LWP).
- Quando não há threads para serem escalonadas, o LWP pode permanecer desocupado, e pode até ser removido (destruído) pelo kernel.

NOTA

- Este conceito foi virtualmente abandonado – é somente thread nível usuário ou nível kernel.



MULTITHREADED WEB CLIENT

Escondendo latências da rede:

- Web browsers escaneiam páginas HTML no recebimento, e descobre que mais páginas (files) são necessárias de serem buscadas.
- **Cada arquivo é buscado por uma thread separada**, cada uma realizando uma requisição HTTP.
- Conforme os arquivos chegam, o browser vai mostrando (displays) eles.

MULTIPLAS CHAMADAS REQUISIÇÕES-RESPOSTA PARA OUTRAS MÁQUINAS (RPC)

- Um cliente realiza várias chamadas ao mesmo tempo, cada um por uma thread diferente.
- Ele então espera até que os resultados sejam retornados.
- Nota: se chamadas são feitas a diferentes servidores, Podemos ter **aceleração linear**.



CLIENTES MULTITHREADED – AJUDA ?

PARALELISMO EM NÍVEL THREAD: TLP

- Se c_i denota uma fração de tempo para que exatamente i threads estejam sendo executadas simultaneamente.
- com N sendo o máximo número de threads que (podem) executar ao mesmo tempo

$$TLP = \frac{\sum_{i=1}^N i \cdot c_i}{1 - c_0}$$

MEDIDAS NA PRÁTICA

Um Web browser típico tem um valor entre 1.5 e 2.5 → threads são primariamente usadas para a organização lógica de browsers.



USANDO THREADS LADO SERVIDOR

MELHORA DE DESEMPENHO

- Iniciar uma thread é mais custoso que iniciar um novo processo.
- Ter um servidor single-threaded proíbe o escalamento para um sistema **multiprocessor**.
- Como nos clientes: esconde a **latência da rede** reagindo a próxima requisição enquanto a anterior está sendo atendida.

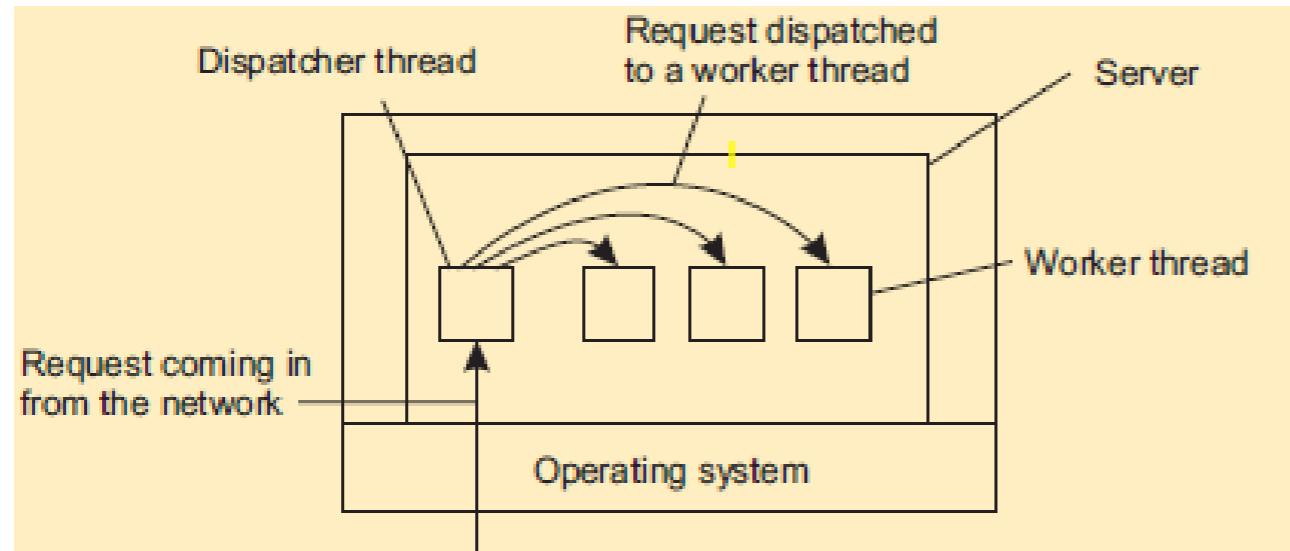
MELHOR ESTRUTURA

- Maioria dos servidores tem altas demandas de E/S. Usando simples, chamadas bloqueantes bem entendidas simplifica a estrutura toda.
- Programas multi-thread tendem a ser menores, mais simples de entender devido a fluxo de controle simplificado.



POR QUE MULTITHREAD É POPULAR: ORGANIZAÇÃO

MODELO DISPATCHER / WORKER



VISÃO GERAL

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

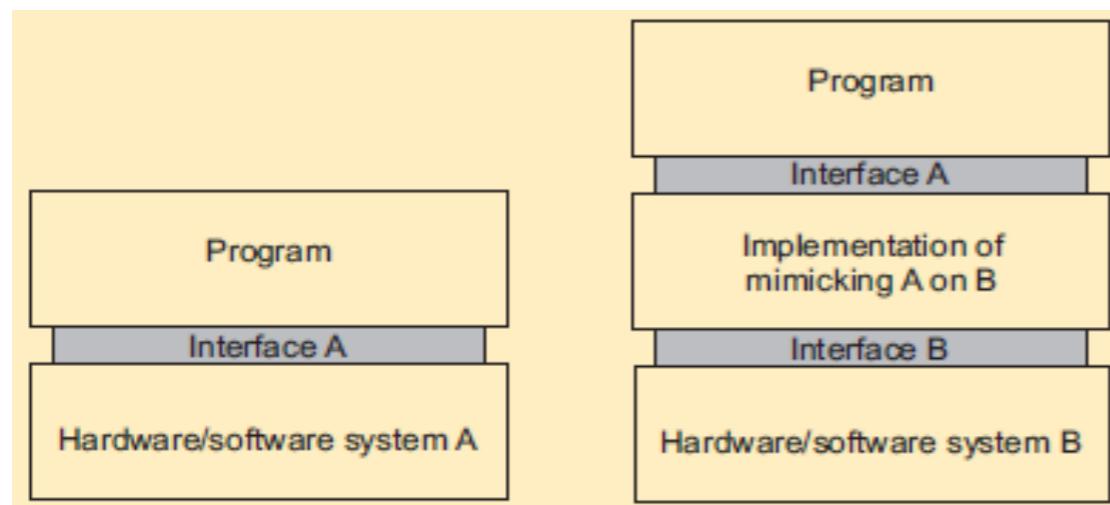
VIRTUALIZAÇÃO

OBSERVAÇÃO

Virtualização é importante:

- Hardware muda mais rápido que software
- Facilita a portabilidade e a migração de código
- Isolamento de components em falha ou atacados

PRINCÍPIO: IMITANDO INTERFACES



IMITANDO INTERFACES

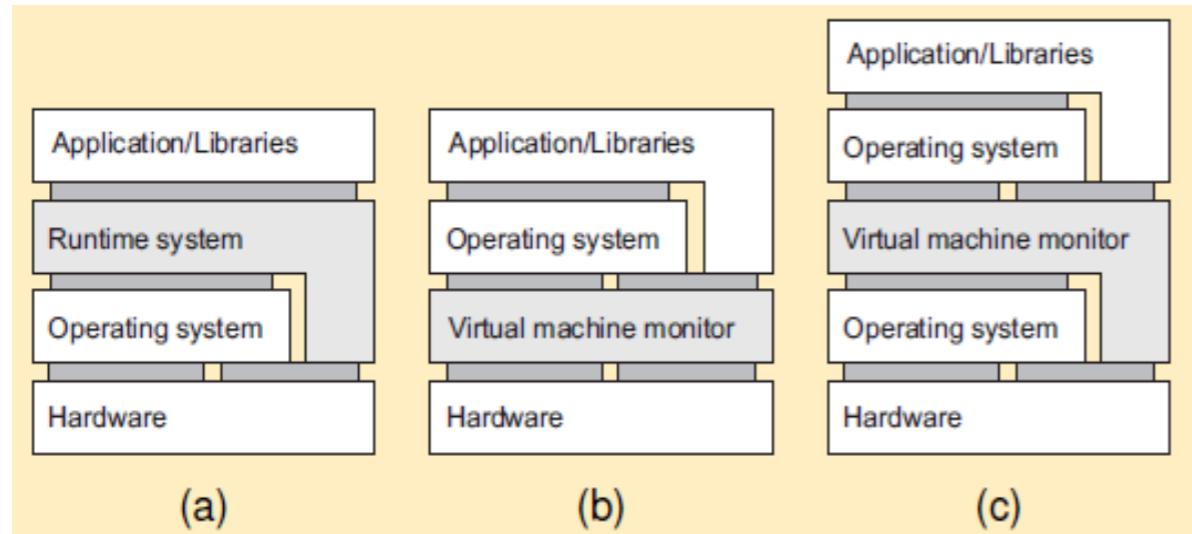
4 TIPOS DE INTERFACES EM 3 DIFERENTES NÍVEIS

1. **Instruction set architecture (i)**: o conjunto de instruções de máquina, com dois subconjuntos:
 - Instruções privilegiadas: permitidas para serem executadas somente pelo SO.
 - Instruções Gerais: podem ser executadas por qualquer programa.
2. **System calls (ii)** são oferecidas por um SO.
3. **Library calls (iii)**, conhecidas como API (**application programming interface (iv)**)



FORMAS DE VIRTUALIZAÇÃO

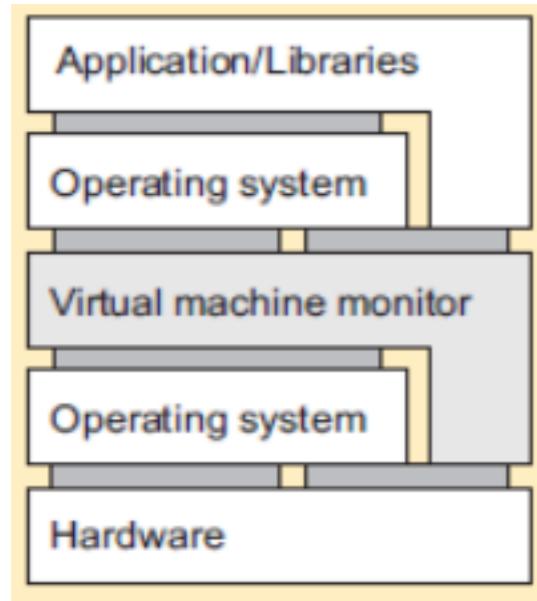
(a) PROCESSO VM, (b) VMM NATIVO, (c) VMM HOSTED



DIFERENÇAS

- (a) Conjunto separado de instruções, um interpretador / emulador rodando no topo do SO
- (b) Instruções de baixo nível (low-level), junto com um SO mínimo (bare-bone)
- (c) Instruções de baixo nível, as delegando o trabalho para um SO completo

VMs DESEMPENHO



- Instrução privilegiada: se e somente se executada em modo usuário –pode causar um trap para o SO
- Instrução não privilegiada: o restante

INSTRUÇÕES ESPECIAIS

- **Instrução sensível a controle:** pode afetar a configuração da máquina (p.ex. afetando o registro de relocação ou tabela de interrupção).
- **Instrução sensível a comportamento:** efeito é parcialmente determinado por contexto (p.ex. POPF seta uma flag que habilita interrupção, mas somente em modo Sistema)

CONDIÇÃO PARA VIRTUALIZAÇÃO

CONDIÇÃO NECESSÁRIA

- Para muitos Computadores convencionais, um monitor de máquina virtual pode ser construído se um conjunto de instruções sensíveis para aquela máquina é um subconjunto de instruções privilegiadas.

PROBLEMA: condição nem sempre é satisfeita

- Pode existir instruções sensíveis que são executadas em modo usuário sem causar um *trap* para o SO

SOLUÇÕES

- Emular todas instruções
- *Wrap* instruções sensíveis não privilegiadas para desviar o controle para a VMM (*virtual machine monitor*)
- Para-virtualização: modifique o SO impedindo instruções sensíveis não privilegiadas ou tornando elas não sensíveis (i.e. mudando o contexto)



VMs E CLOUD COMPUTING

3 TIPOS DE SERVIÇOS EM CLOUD

- **Infrastructure-as-a-service:** cobrindo a infrestrutura básica
- **Platform-as-a-service:** cobrindo serviços em nível Sistema
- **Software-as-a-service:** contendo a aplicação em sí

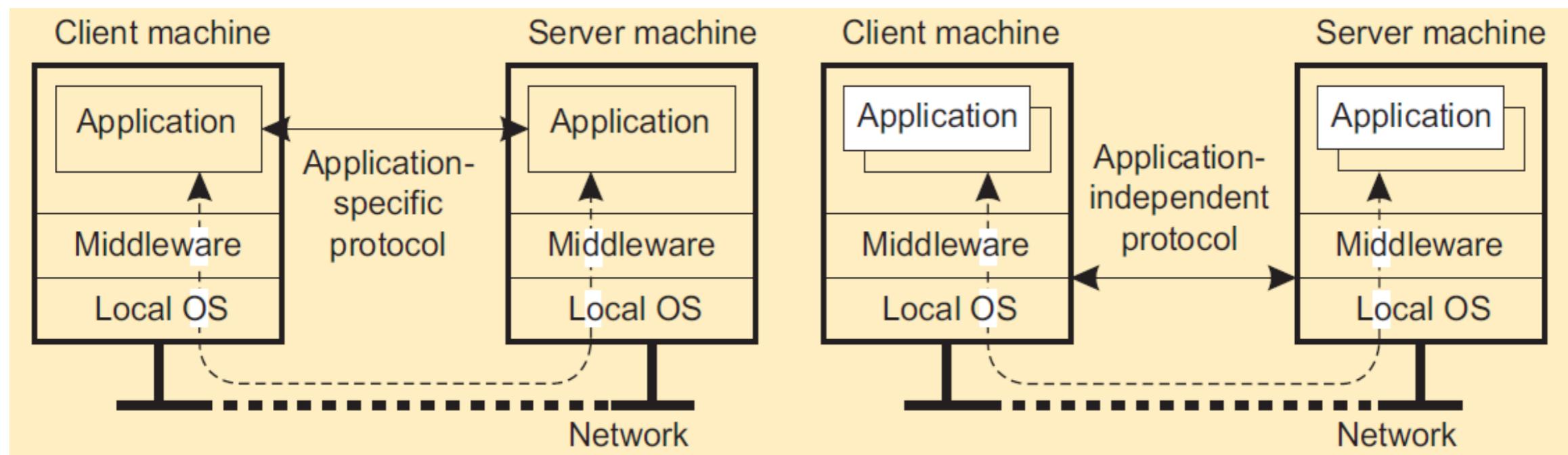
IaaS

- Ao invés de alugar uma máquina física, um provedor de serviços em núvem vai alugar uma VM (ou VMM) que pode possivelmente estar compartilhando uma máquina física com outros clientes → com quase isolamento total entre clientes (embora o isolamento de desempenho possa nunca ser atingido)



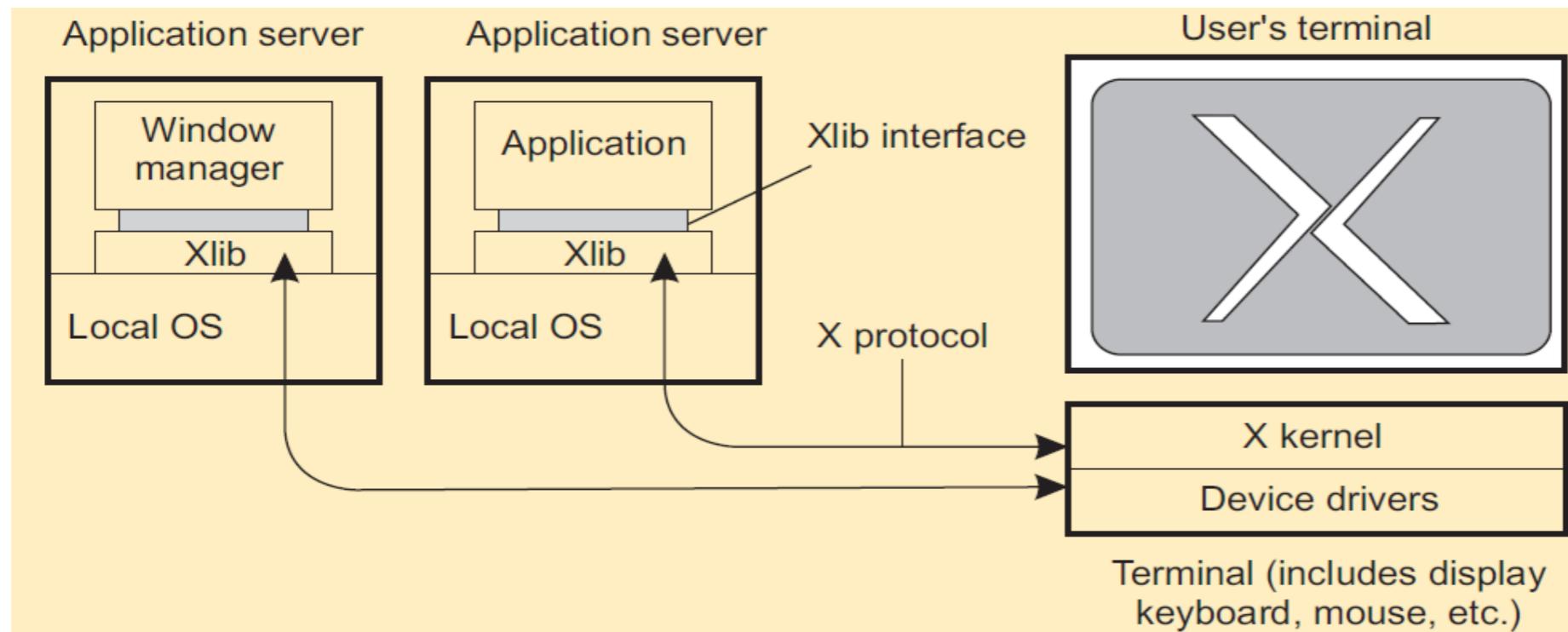
INTERAÇÃO CLIENTE-SERVIDOR

DISTINÇÃO ENTRE SOLUÇÕES EM NÍVEL APLICAÇÃO E EM NÍVEL MIDDLEWARE



EXEMPLO SISTEMA X-WINDOW

ORGANIZAÇÃO BÁSICA



X CLIENTE E SERVIDOR

- A aplicação age como um cliente para o X-kernel, que roda como um servidor na máquina do cliente

EXEMPLO: O SISTEMA X WINDOW

MELHORANDO X WINDOW

OBSERVAÇÕES PRÁTICAS

- Existe frequentemente uma separação entre a lógica de controle e os commandos na interface de usuário
- Aplicações tendem a operarem sincronismo apertado com o X-kernel

ABORDAGEM ALTERNATIVA

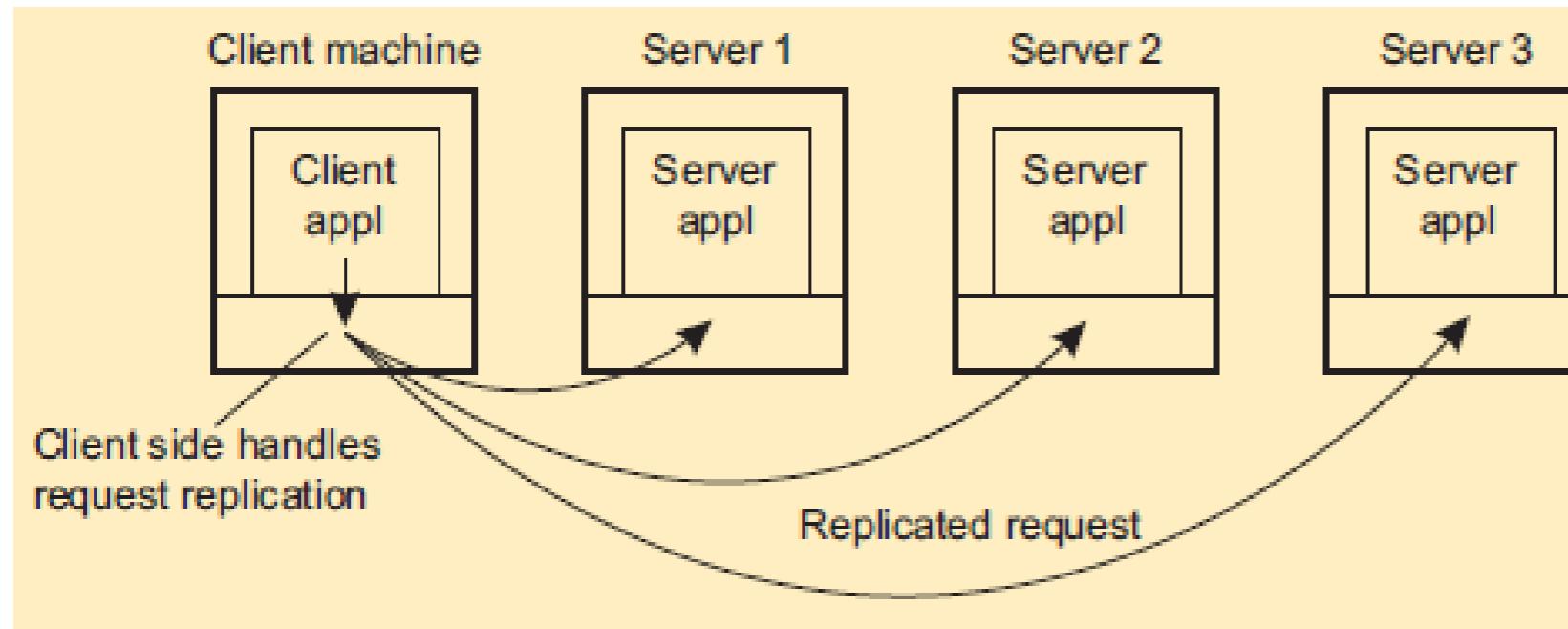
- Deixar as aplicações controlarem o display **completamente**, até o nível de pixel (p.ex. **VNC**)
- Prover somente poucas operações de display no alto nível (dependente de drivers de vídeo local), permitindo operações de display mais eficientes



SOFTWARE LADO CLIENTE

GERALMENTE CUSTOMIZADO PARA TRANSPARÊNCIA DISTRIBUÍDA

- Transparência no acesso: stubs no lado cliente para RPCs
- Transparência localidade / migração: deixa o software do lado cliente manter a localização atual
- Transparência replicação: múltiplas invocações manuseadas pelo stub cliente



- Transparência falha: pode frequentemente ser colocado somente do lado cliente (se tentarmos mascarar falhas no servidor e na comunicação)

SERVIDORES ORGANIZAÇÃO GERAL

MODELO BÁSICO

Um processo implementando um serviço específico em nome de uma coleção de clientes. Espera por requisições entrantes vindas de um cliente e subsequentemente garante que a requisição é atendida, e na sequência espera pela próxima requisição entrante.



SERVIDORES CONCORRENTES

DOIS TIPOS BÁSICOS

- Servidor interativo: servidor manipula uma requisição antes de atender a próxima requisição
- Servidor concorrente: usa um dispatcher, que pega uma requisição entrante e então passa para um thread/processo separado atender.

OBSERVAÇÃO

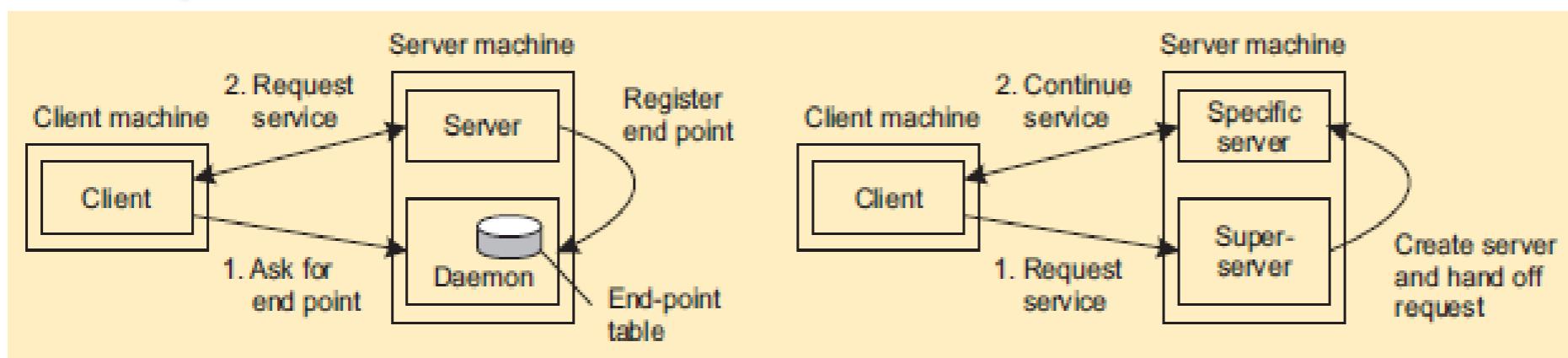
- Servidores concorrentes é a norma: eles podem facilmente manipular múltiplas requisições , notavelmente na presença de operações bloqueantes (disco ou outros servidores)

CONTATANDO UM SERVIDOR

OBSERVAÇÃO: A MAIORIA DOS SERVIÇOS SÃO ATRELADOS A UM PORTO ESPECÍFICO

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)

ATRIBUIÇÃO DINÂMICA DE UM “END POINT”



COMUNICAÇÃO OUT-OF-BAND (FORA DE ORDEM)

QUESTÃO

É possível **interromper** um servidor uma vez que ele tenha aceitado uma requisição de serviço (ou esteja em processo de aceitar)

SOLUÇÃO 1: USE UM PORTO SEPARADO PARA DADOS URGENTES

- Servidor com uma thread/processo separada para mensagens urgentes
- Mensagem urgente → **requisição associada é colocada em espera**
- Nota: necessário **suporte do SO para escalonamento baseado em prioridade**

SOLUÇÃO 2: USE AS FACILIDADES DA CAMADA DE TRANSPORTE

- Exemplo: TCP permite mensagens urgentes na mesma conexão
- Mensagens urgentes podem ser capturadas usando técnicas de sinalização dos SO's

SERVIDORES E ESTADO

SERVIDORES STATELESS

Nunca mantém informações **precisas** sobre o estado de um cliente depois de ter realizado o atendimento de uma requisição

- Não registra se um arquivo foi aberto (simplesmente fecha ele novamente após o acesso)
- Não promete invalidar a cache de um cliente
- Não mantém informações sobre seus clientes

CONSEQUÊNCIAS

- Clientes e servidores são **completamente independentes**
- **Inconsistências de estado** devido a quedas do servidor **são reduzidas**
- Possíveis **perdas de desempenho** por causa de coisas que não podem ser antecipadas (ex. um cliente que precisa antecipar prefetch de blocos)

QUESTÃO

Comunicação orientada a conexão serve para **stateless design** ?

SERVIDORES E ESTADO

SERVIDORES STATEFUL

Mantém o estado de seus clientes

- Registra que um arquivo foi aberto de forma que é possível fazer prefetch
- Sabe quais dados estão na cache do cliente, e permite a manutenção de cópias locais de dados compartilhados

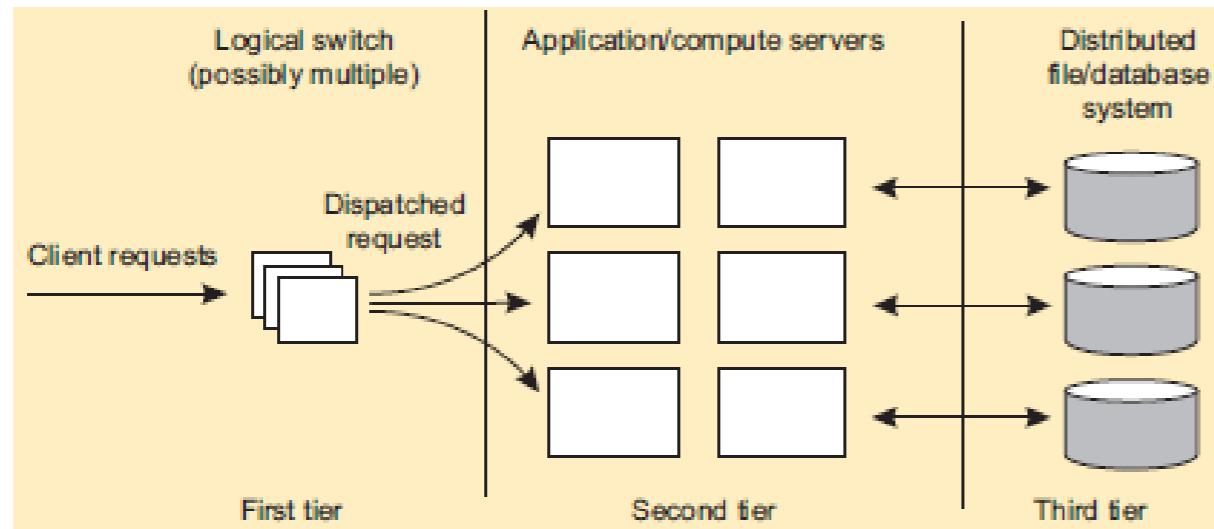
OBSERVAÇÃO

O desempenho de um servidor *stateful* pode ser bem alto, desde que os cliente permitam cópias locais. Na verdade, confiabilidade nem sempre é o maior problema



TRÊS DIFERENTES TIERS

ORGANIZAÇÃO COMUM



ELEMENTO CRUCIAL

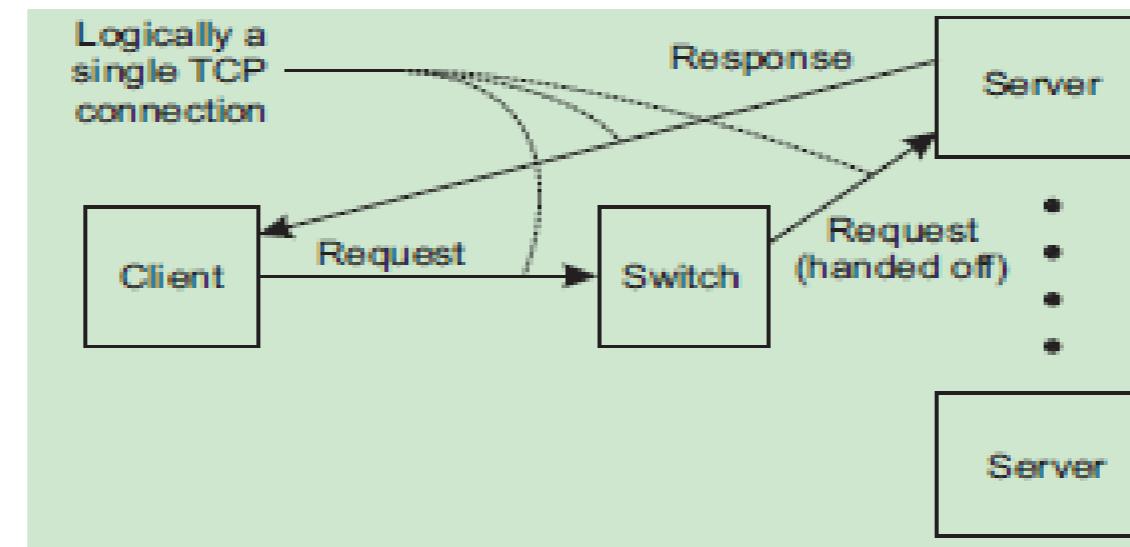
O primeiro tier é geralmente responsável por passar requisições para um servidor apropriado: **request dispatching**

HANDLING REQUISIÇÕES

OBSERVAÇÃO

- Ter a primeira camada (tier) manuseando toda comunicação de/para o cluster pode levar a **bottleneck**

SOLUÇÃO: TCP HANDOFF

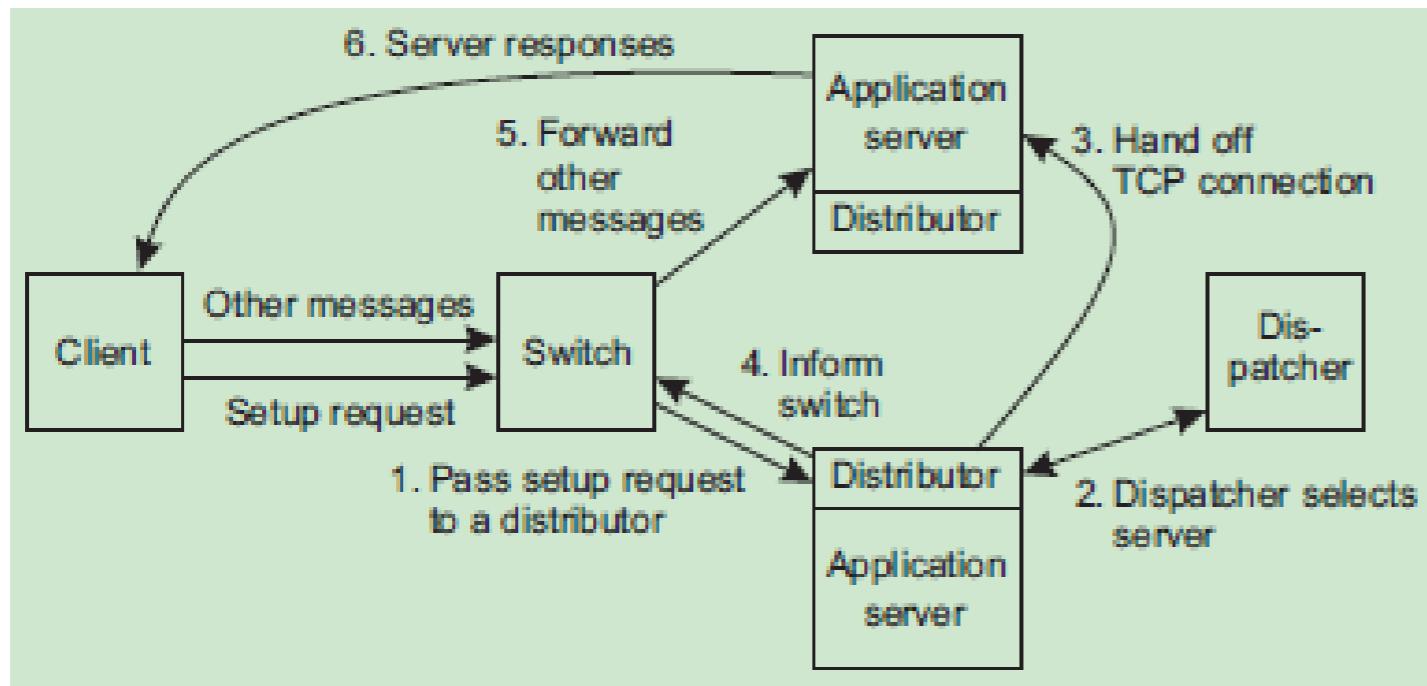


CLUSTER DE SERVIDORES

O FRONT-END PODE FACILMENTE FICAR SOBRECARREGADO: ATENÇÃO ESPECIAL PODE SER NECESSÁRIA

- Chaveamento na camada de transporte: o front-end passa as requisições TCP para um dos servidores, considerando algumas medidas de desempenho na decisão
- Distribuição baseada em conteúdo: front-end lê o conteúdo das requisições e seleciona o melhor servidor

SOLUÇÃO: TCP HANDOFF



SERVidores ESPALHADOS PELA INTERNET

OBSERVAÇÃO

Espalhar servidores na internet pode introduzir problemas administrativos. Isto pode ser minimizado usando data centers de um mesmo provedor de nuvem

DISPATCH DE REQUISIÇÃO: SE LOCALIDADE É IMPORTANTE

Abordagem comum: use DNS

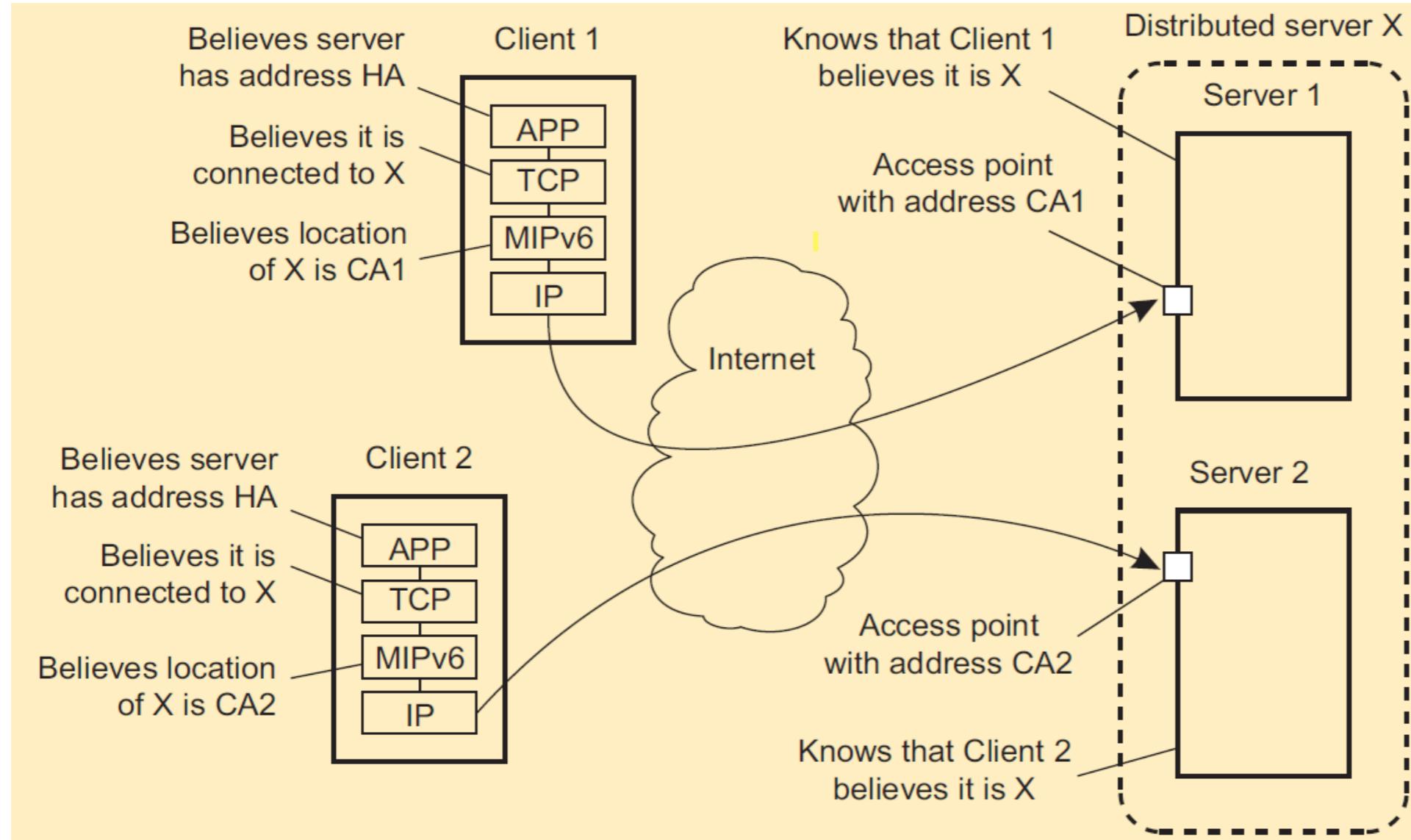
- Cliente procura um serviço específico através do DNS – o endereço do cliente é parte da requisição
- Servidor DNS mantém referência para servidores replica para requisitar serviços e retornar endereço do servidor mais local

TRANSPARÊNCIA NO CLIENTE

Para manter o cliente sem noção da distribuição, deixe o DNS resolver em nome do cliente. Problema é que o resolvedor pode na verdade estar **longe do local** do cliente atual



SERVIDORES DISTRIBUÍDOS COM ENDEREÇOS IPv6 ESTÁVEIS



SERVIDORES DISTRIBUÍDOS DETALHES DE ENDEREÇAMENTO

ESSÊNCIA: CLIENTES QUE POSSUEM UM IPv6 MÓVEL PODEM TRANSPARENTEMENTE SET UP UMA CONEXÃO PARA QUALQUER PEER

- Cliente C sets up uma conexão IPv6 *home address* (HA)
- HA é mantido por um agente local (*home agent*) (*network level*) que entrega a conexão para um registrador que toma conta do endereço CA (*Care of Address*)
- C pode então aplicar uma *otimização de rota* através do *forwarding* de pacotes para o endereço CA (i.e. sem o *hand off* através do agente local)

SISTEMAS DISTRIBUÍDOS COLABORATIVOS

- Servidor origem mantém um endereço local (*home*), mas *hands off* conexões para endereços de *peers* colaboradores → servidor origem e *peer* aparecem como um servidor

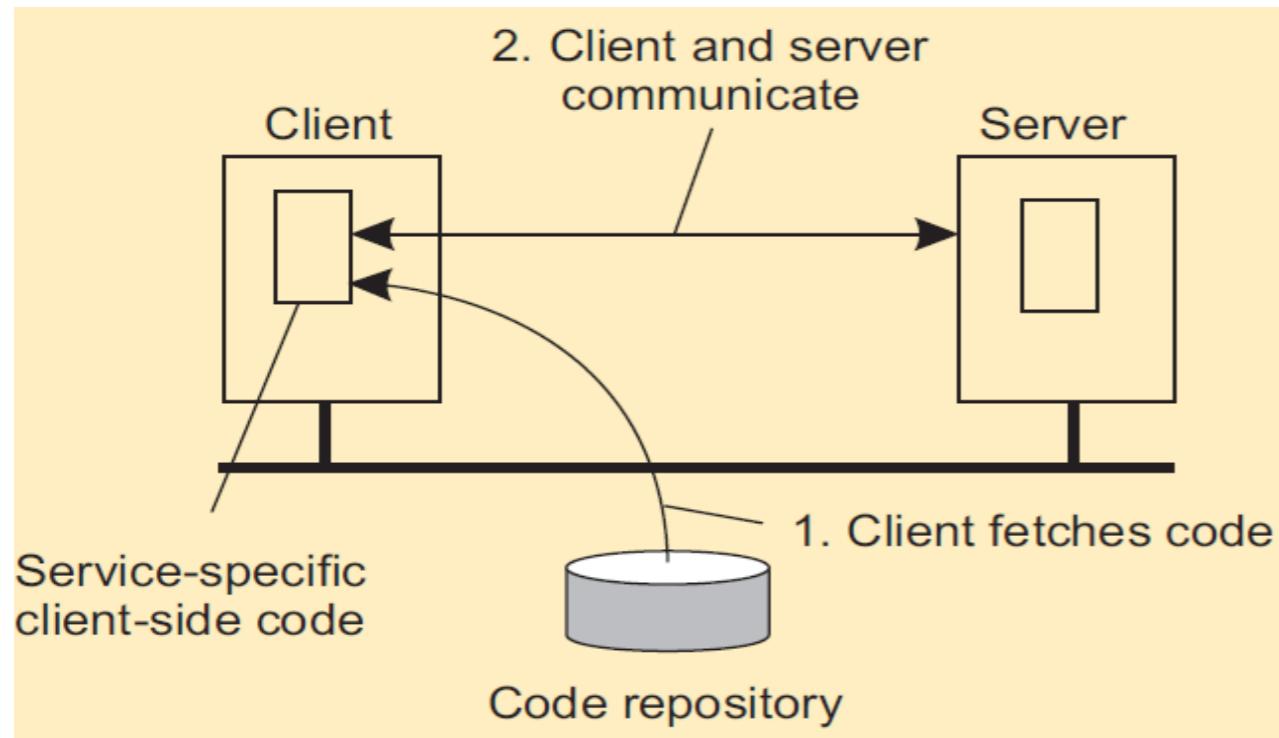


RAZÕES PARA MIGRAR CÓDIGO

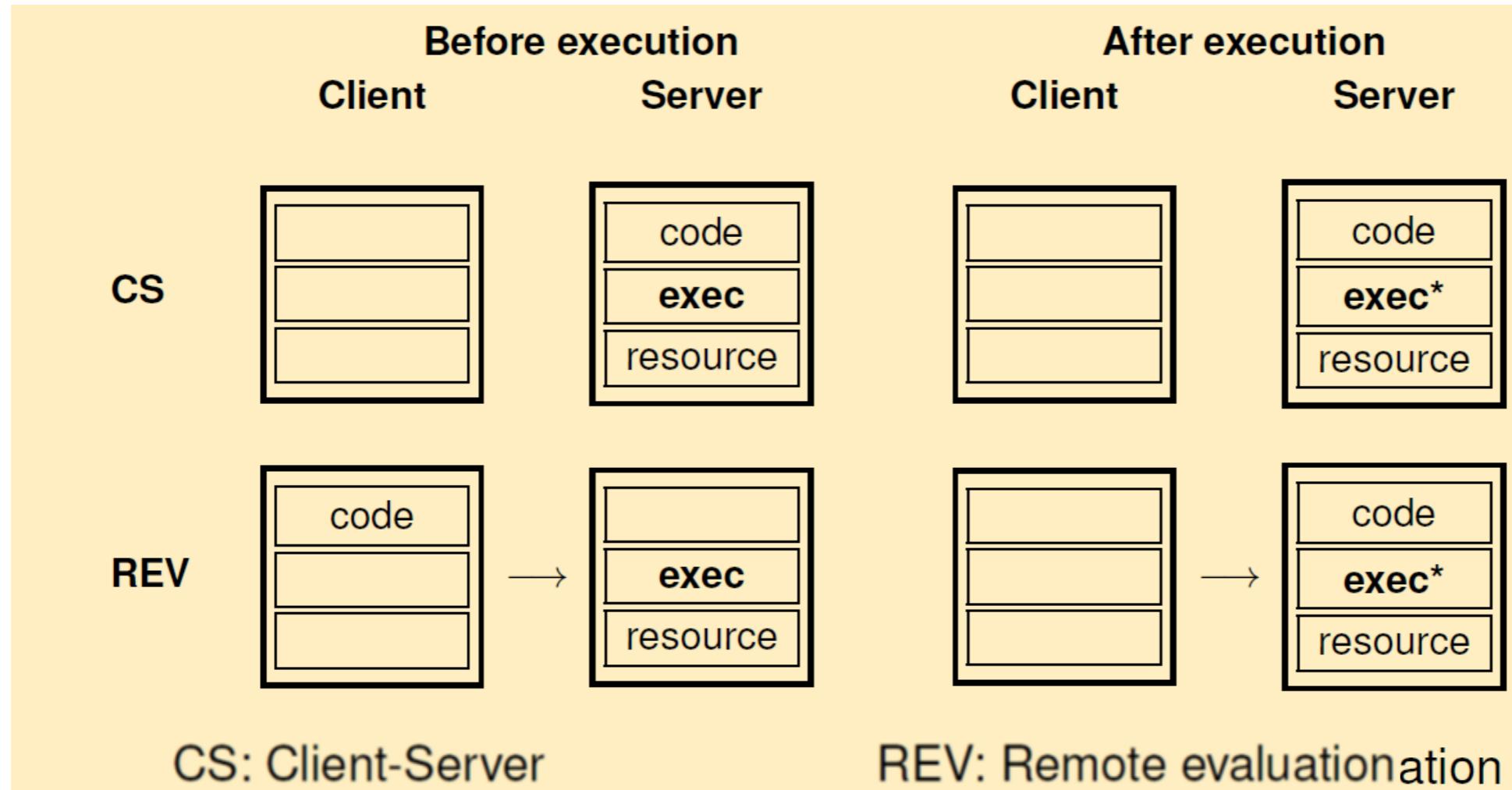
DISTRIBUIÇÃO DE CARGA

- Garantir que servidores em data centers estão **suficientemente** carregados (para minimizar desperdício de energia)
- Minimizar comunicação para garantir que computações estão próximas dos dados (pense em computação móvel)

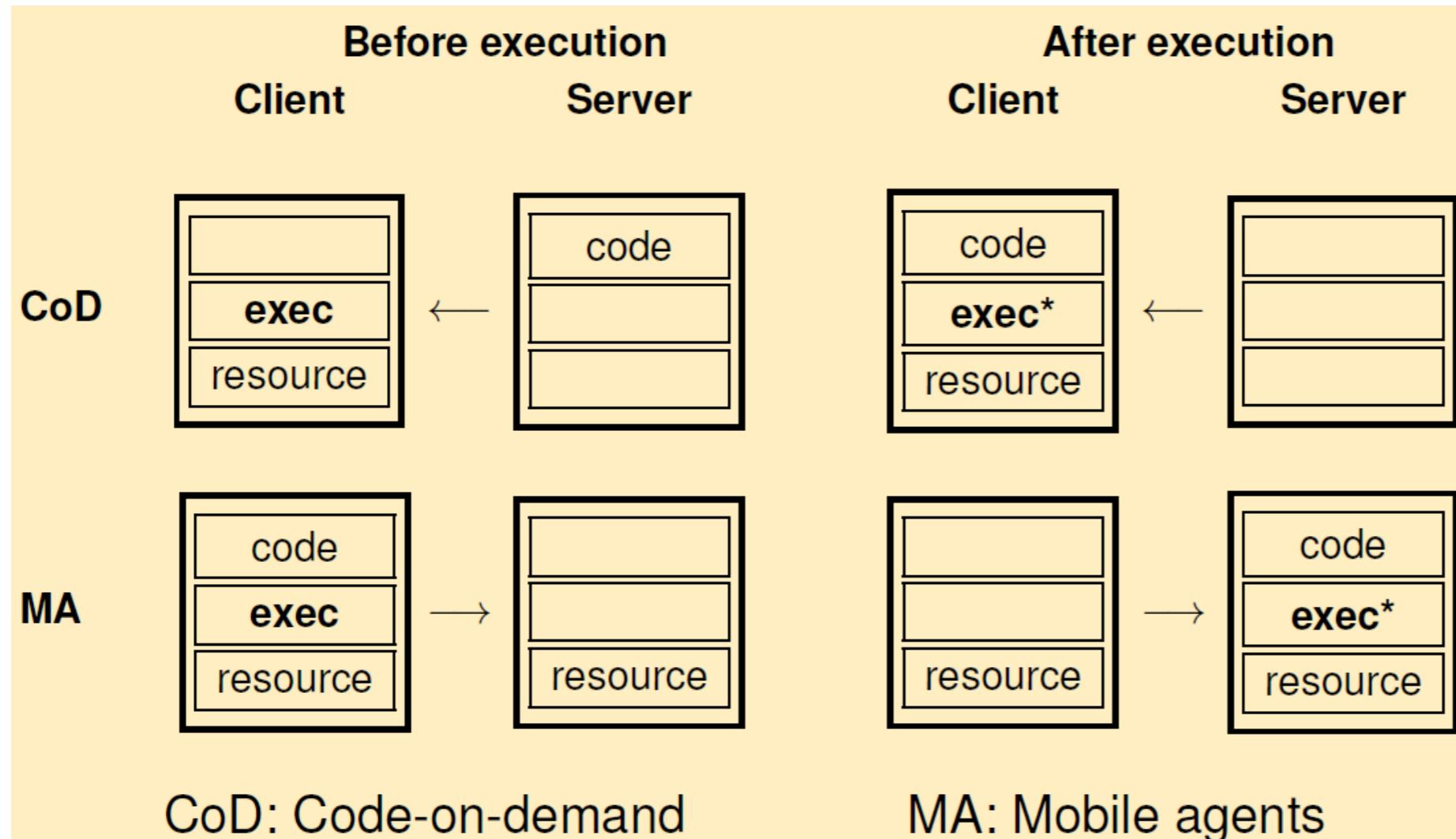
FLEXIBILIDADE: MOVER CÓDIGO PARA CLIENTE QUANDO NECESSÁRIO



MODELOS PARA MIGRAÇÃO DE CÓDIGO



MODELOS PARA MIGRAÇÃO DE CÓDIGO



MOBILIDADE FORTE E FRACA

COMPONENTES OBJETO

- Segmento de código: contém o código atual
- Segmento de dados: contém o estado
- Estado execução: contém contexto de thread executando o código do objeto

MOBILIDADE FRACA: MOVE SOMENTE O CÓDIGO E SEGMENTO DE DADOS (E REBOOT EXEC.)

- Relativamente simples, especialmente se código é portável
- Distingue **code shipping** (push) de **code fetching** (pull)

MOBILIDADE FORTE: MOVE COMPONENTE, INCLUINDO ESTADO DA EXECUÇÃO

- **Migração**: move o objeto todo de uma máquina pra outra
- **Cloning**: inicia um clone, e ajusta ele no mesmo estado de execução.



MIGRAÇÃO EM SISTEMAS HETEROGÊNEOS

PROBLEMA PRINCIPAL

- A máquina alvo pode não ser adequada para executar o código migrado
- A definição de contexto de processo / thread / processador é altamente dependente do hardware local, SO e runtime system

ÚNICA SOLUÇÃO: MÁQUINA ABSTRATA IMPLEMENTADA EM DIFERENTES PLATAFORMAS

- Linguagens interpretadas, efetivamente tendo sua própria VM
- VMM's



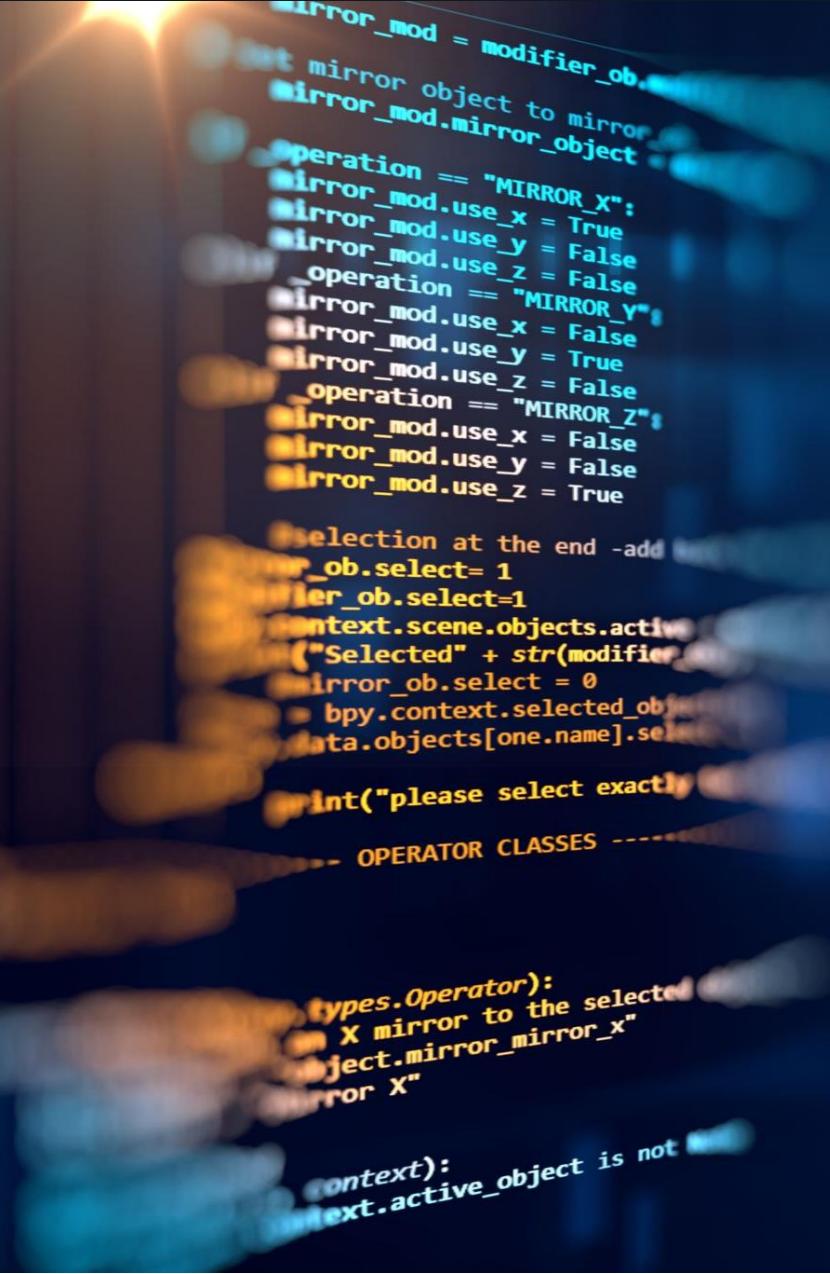
MIGRAÇÃO DE MÁQUINA VIRTUAL

MIGRANDO IMAGENS: TRÊS ALTERNATIVAS

1. Fazer push de páginas de memória para uma nova máquina e enviar as que forem modificadas mais tarde durante o processo de migração
2. Parar a máquina virtual; migrar memória e iniciar uma nova VM
3. Deixar uma nova VM puxar (pull in) novas páginas sob demanda: processos iniciam imediatamente na nova VM e copiam páginas de memória sob demanda.



Docker: o que é?

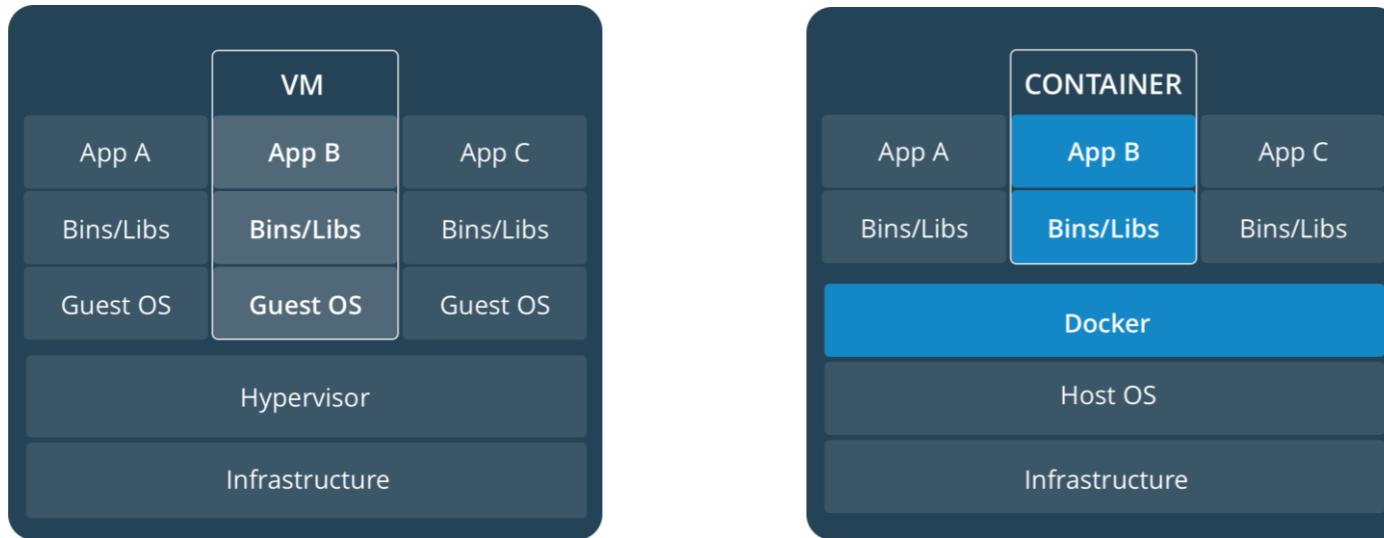


- *Um projeto de código aberto que automatiza a implantação de aplicativos de software dentro de contêineres, fornecendo uma camada adicional de abstração e automação de virtualização de nível de SO.*
- *Simplificando: o Docker é uma ferramenta que permite aos desenvolvedores, administradores de sistemas, etc., implantar facilmente seus aplicativos em uma caixa de areia (sandbox) (chamada de contêineres) para executar no sistema operacional hospedeiro.*
- *Docker permite empacotar um aplicativo com todas as suas dependências em uma unidade padronizada para desenvolvimento de software. Ao contrário das máquinas virtuais, os contêineres não têm alta sobrecarga e, portanto, permitem um uso mais eficiente do sistema e dos recursos subjacentes.*

Source: docs.docker.com

O que é um container?

- Container ≠ VM
- Isolado
- Compartilha o OS
- E as vezes bins/libs
- Consome somente uma pequena fração de tempo computacional



Source: docs.docker.com

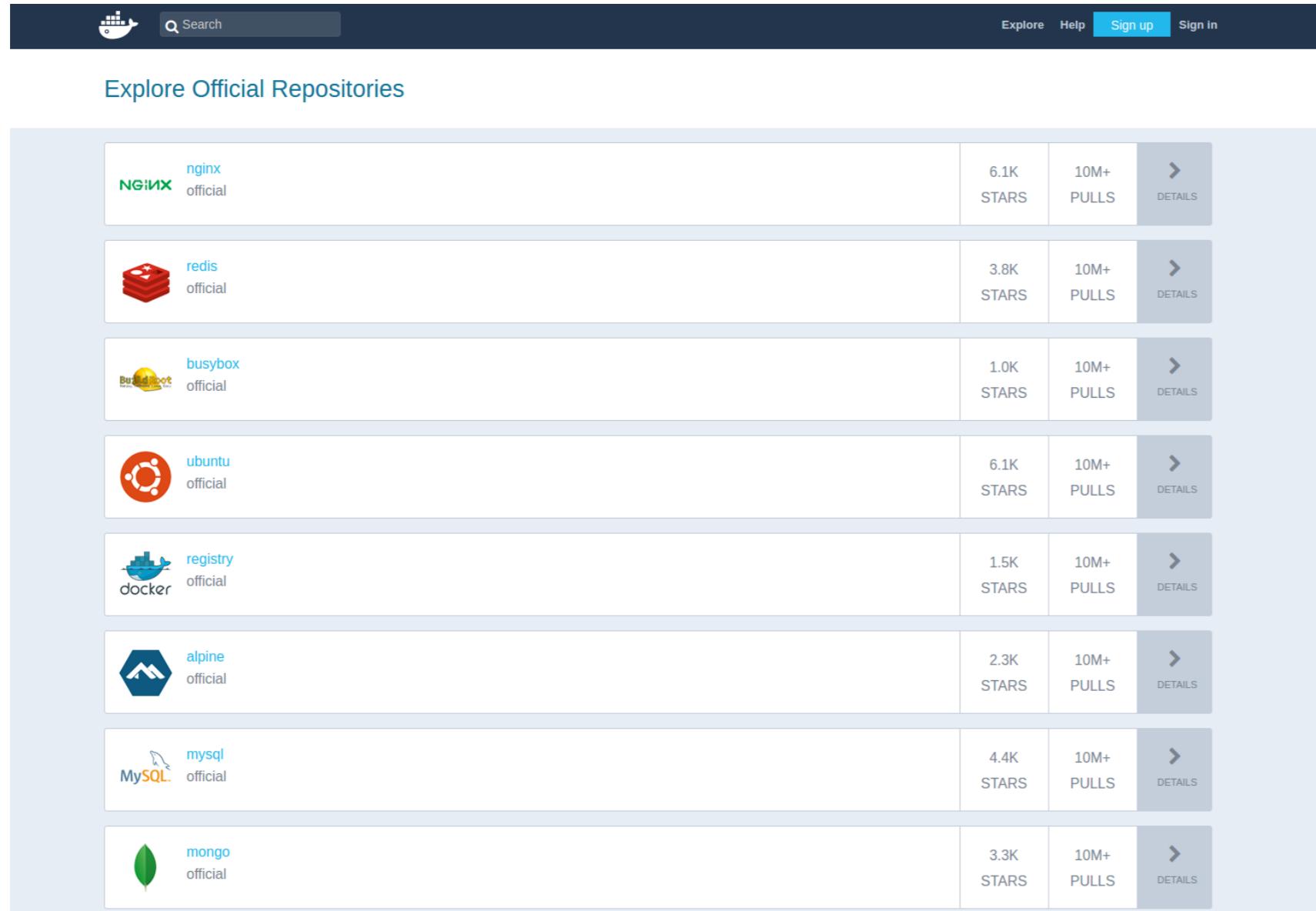


Por que usar containers ?

- Os contêineres oferecem um mecanismo de empacotamento lógico no qual os aplicativos podem ser abstraídos do ambiente em que realmente são executados.
- Esse desacoplamento permite que aplicativos baseados em contêiner sejam implantados de maneira fácil e consistente, independentemente de o ambiente de destino ser um data center privado, a nuvem pública ou até mesmo um laptop pessoal do desenvolvedor.
- Permite a capacidade de criar ambientes previsíveis que são isolados do resto dos aplicativos e podem ser executados em qualquer lugar.
- Além da portabilidade, os contêineres oferecem controle mais granular sobre os recursos, proporcionando maior eficiência à infraestrutura.
- Google, Facebook, Netflix e Salesforce utilizam contêineres



Docker Hub (hub.docker.com): crie uma conta



The screenshot shows the Docker Hub homepage with the title "Explore Official Repositories". It displays a list of eight official Docker repositories:

Repository	Owner	Stars	Pulls	Actions
nginx	NGINX	6.1K	10M+	DETAILS
redis	official	3.8K	10M+	DETAILS
busybox	Buildroot	1.0K	10M+	DETAILS
ubuntu	official	6.1K	10M+	DETAILS
registry	docker	1.5K	10M+	DETAILS
alpine	official	2.3K	10M+	DETAILS
mysql	MySQL	4.4K	10M+	DETAILS
mongo	official	3.3K	10M+	DETAILS

Source: docs.docker.com



Após instalar

- Teste o Docker rodando hello world!

```
$ sudo docker run hello-world
Hello from Docker. This message shows
that your installation appears to be
working correctly. . .
```



Docker + Kubernetes

- Como escalar containers?
- Como garantir o trabalho coordenado entre os diferentes containers de uma aplicação?
- Como detectar containers com falhas e corrigir isso automaticamente?

Utilizando uma orquestração



- Também conhecido como K8s ou kube
- Desenvolvido originalmente pela Google
- Mantido pela Cloud Native Computing Foundation
- Escrito em **Go**
- Open **sourceCluster** com máquina **Master** e **Nodes**
- Criação de objetos através de arquivos no formato **YAML**
- Diversas funcionalidades para gerenciamento
- **kubectl** → ferramenta de linha de comando
- **Minikube** → ambiente de testes



gerenciamento de containers

- Orquestração
- Auto recuperação
- Reinício
- Replicação
- Escalonamento





kubernetes

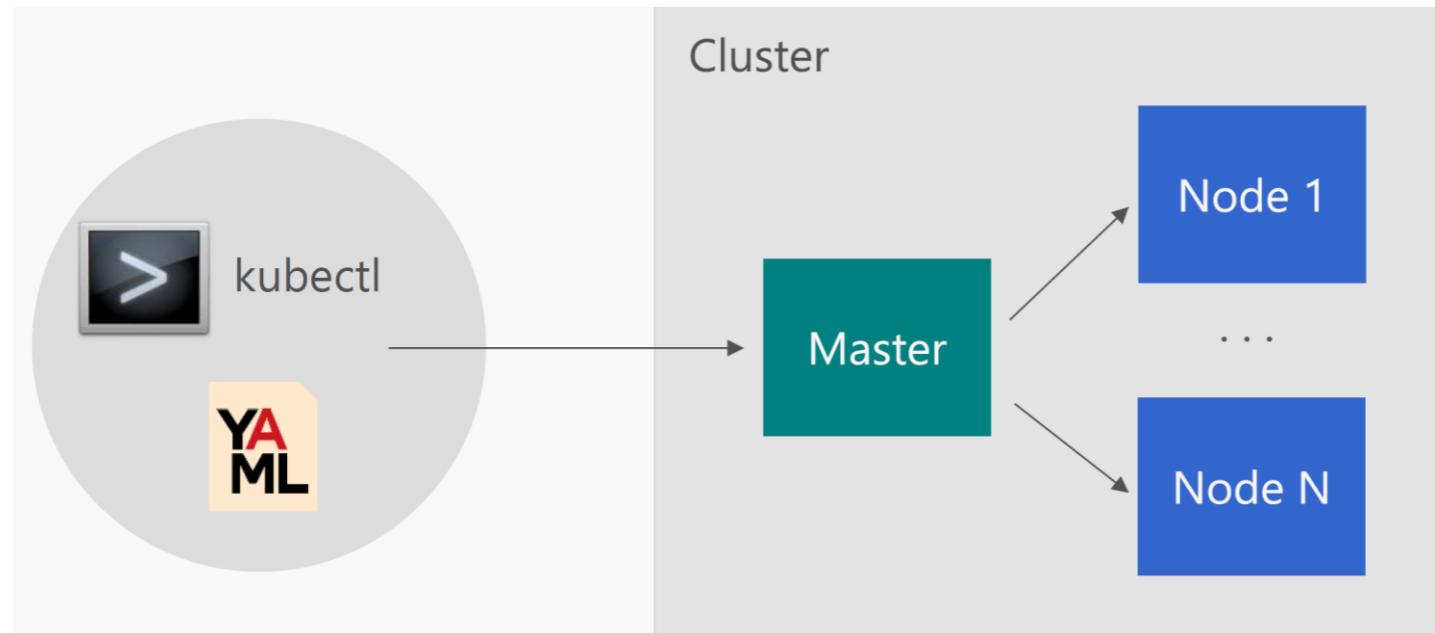
Arquitetura

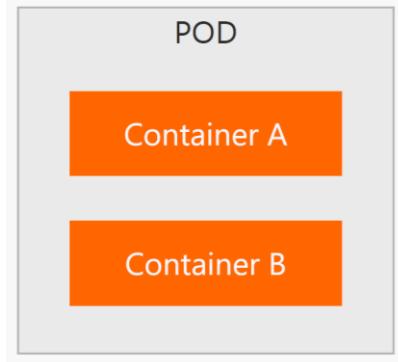
- Master

- Máquina que controla os **Nodes (Nós)**
- Responsável pelas atribuições de tarefas aos **Nodes**

- Nodes

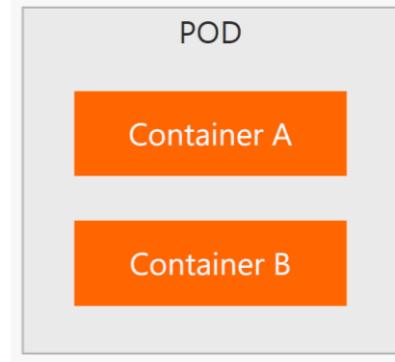
- Máquinas que realizam as tarefas atribuídas pelo **Master**





- Pod

- Grupo de um ou mais containers
- implantados em um **Node (Nó)**
- Compartilham o mesmo endereço IP,
- IPC, nome do host e outros recursos



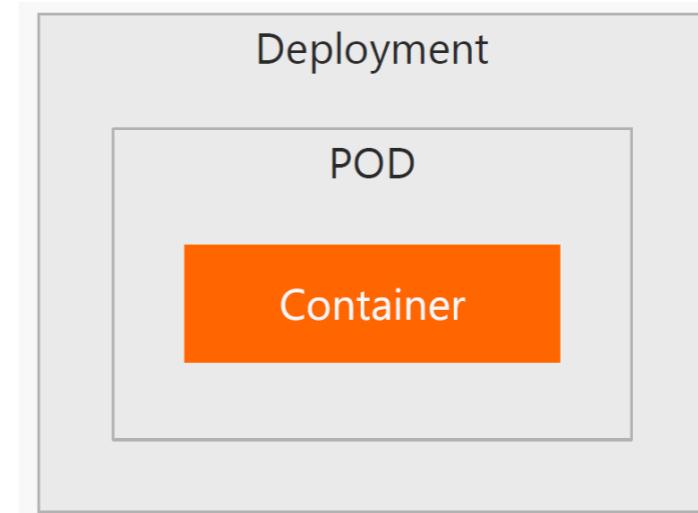


kubernetes

Arquitetura

- Deployment

- Abstração de um **Pod** com recursos adicionais
- Conta com gerenciamento de estados





kubernetes

Arquitetura

- Service

- Objeto mais estável (**Pods** são criados ou removidos continuamente)
- Cuidará do acesso aos **Pods**, funcionando como um **Load Balancer**



- Replication Controller

- Controla quantas cópias idênticas de um **Pod** serão executadas e em quais locais do **cluster**

- Kubelet

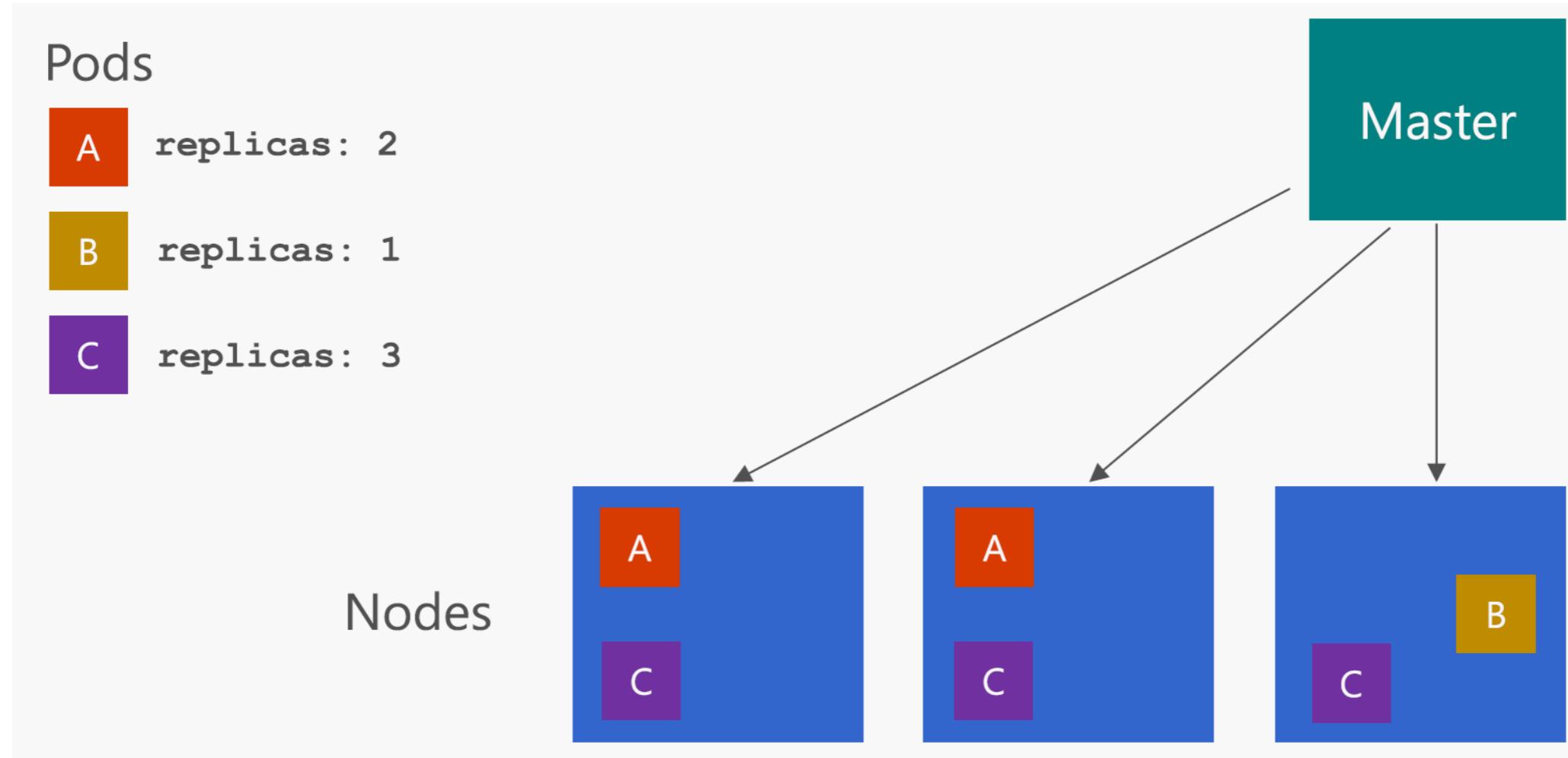
- Serviço que garante a inicialização e execução dos containers nos **Nodes**





kubernetes

Arquitetura



PROFESSOR DO CURSO



FREDY JOÃO VALENTE, PhD

**Engenheiro Elétrico pela EESC-USP / 1987, Mestre em Física Aplicada pelo IFSC-USP / 1991 e
Ph.D. Electronics & Computer Science - University of Southampton UK / 1995**

Prof. Dr. CNPq: ICMC USP 1996-97,
Sócio Diretor da S&V Consultoria 1997-2004: Soluções corporativas NFC/Mifare – Bilhetagem / Controle Acesso Inteligente.
Pós Doc 2005 na Universidade de Southampton: Computação pervasiva e Computação em GRID para Supply-Chain RFID/EPC Gen2.
Sócio Diretor da COSS Consulting 2005-2015: Cadeias de abastecimento inteligentes baseadas em RFID/EPC e Soluções IoT.
Coordenador do projeto [Mions: Mobile Interaction on Spot, Internet of Things \(IoT\) and Context Analysis](#):
FINEP Subvenção Econômica 2009-2012;
Coordenador de vários projetos FAPESP PIPE II e III
Atualmente é Prof. Dr. em Engenharia de Computação no Departamento de Computação - [UFSCar](#) – desde out / 2015.