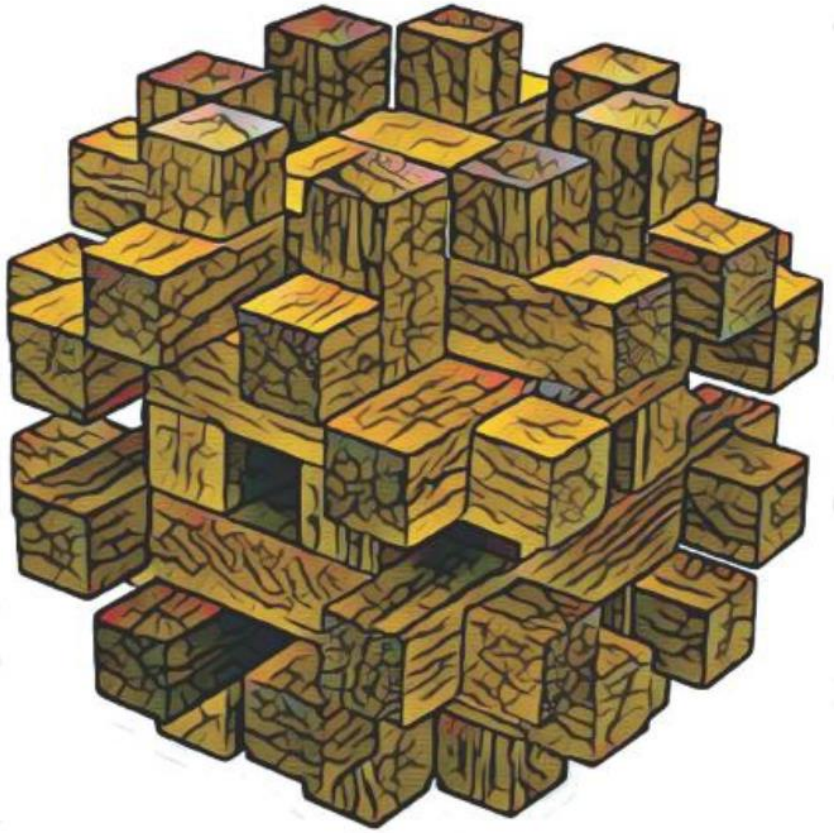
Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



Capítulo 2

Arquiteturas

Segunda-feira, 20 de Junho de 2022

3th Edition – Version 3.03 - 2020

ESTILOS ARQUITETURAIS

IDEIA BÁSICA

Um estilo é formulado em termos de:

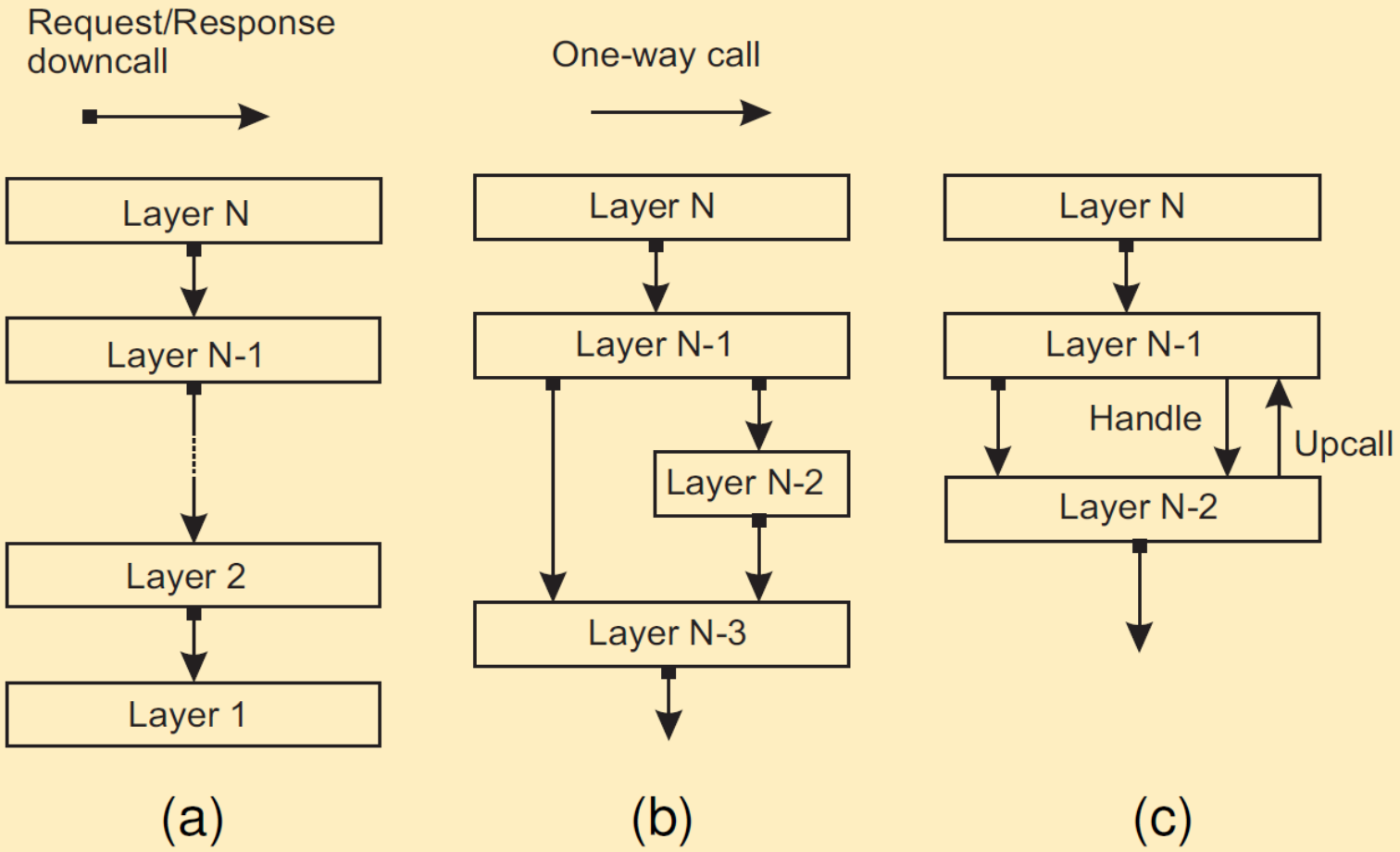
- Componentes (trocáveis) com interfaces bem definidas
- A forma na qual componentes são conectados entre si
- Os dados trocados entre componentes
- Como este componentes e conectores são ajuntados e configurados em um sistema

CONECTOR

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

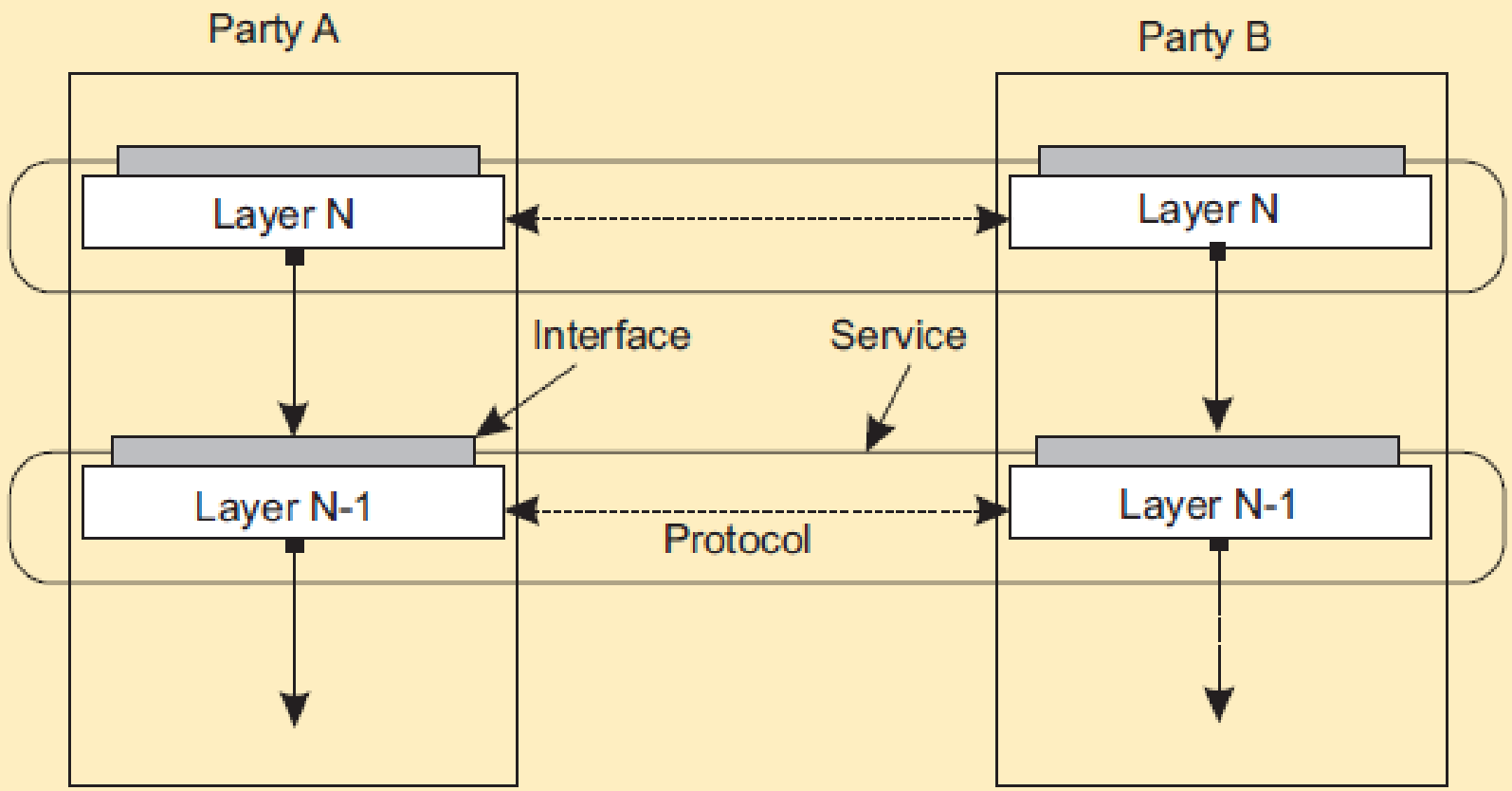
ARQUITETURA EM CAMADAS

DIFERENTES ORGANIZAÇÕES EM CAMADAS

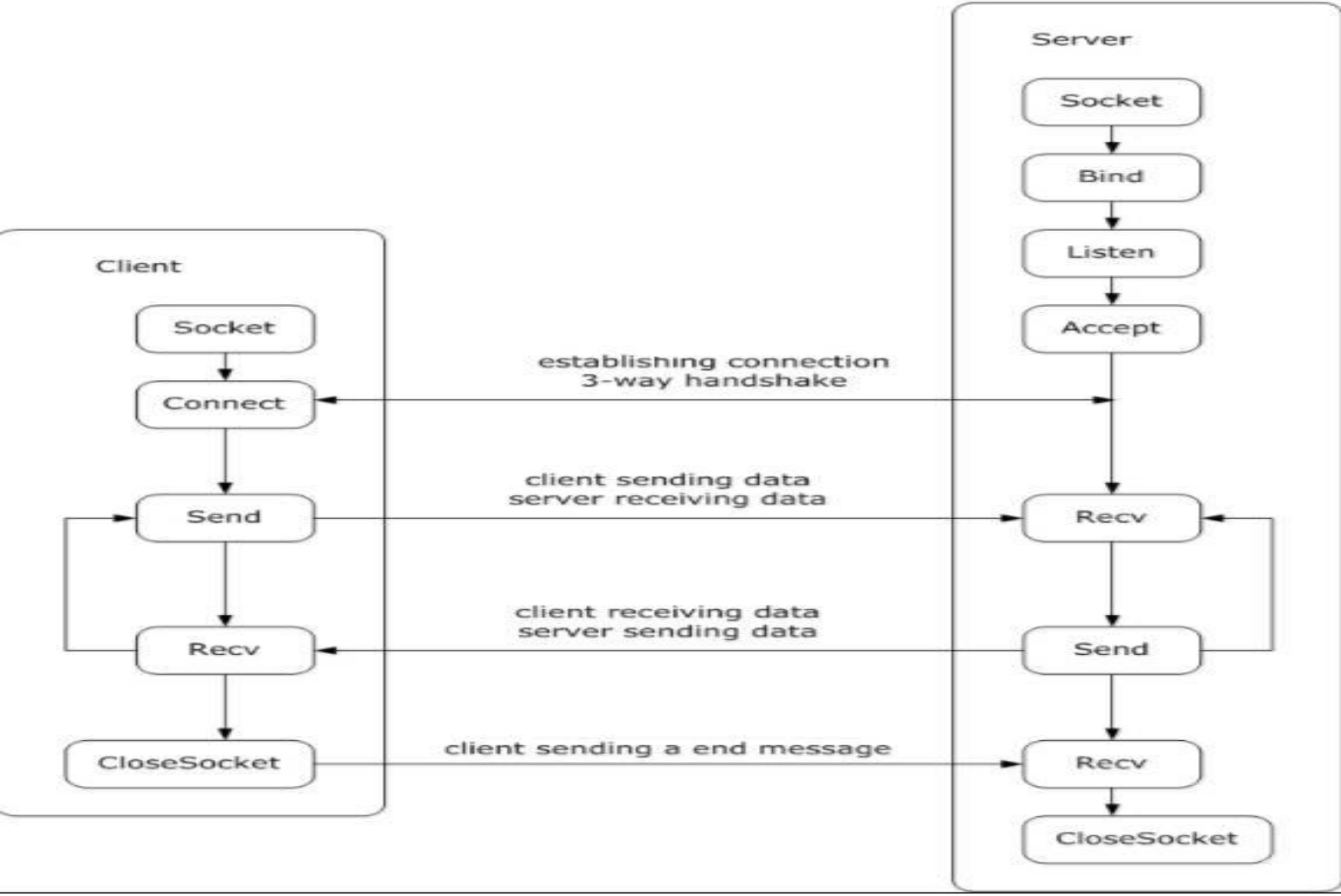


EXEMPLO PROTOCOLOS DE COMUNICAÇÃO

PROTOCOLO, SERVIÇO, INTERFACE



TCP Socket flow diagram



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

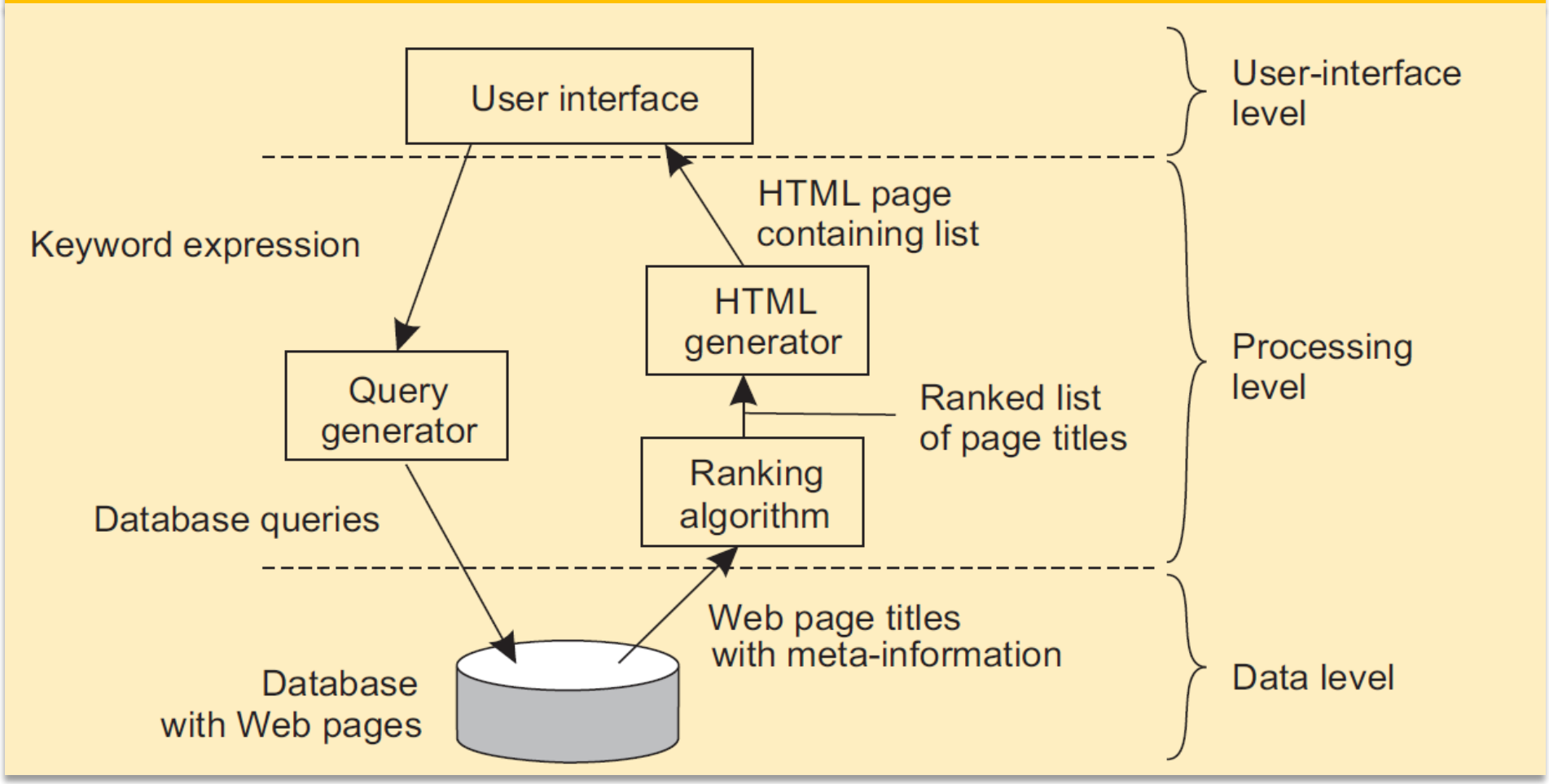
- Camada Interface de aplicação: contém unidades para interfaceamento de usuários ou aplicações externas
- Camada de processamento: contém as funções da aplicação, i.e., sem dados específicos
- Camada de Dados: contém os dados que o cliente deseja para manipular através de componentes de aplicação

OBSERVAÇÃO

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

CAMADAS APLICAÇÃO

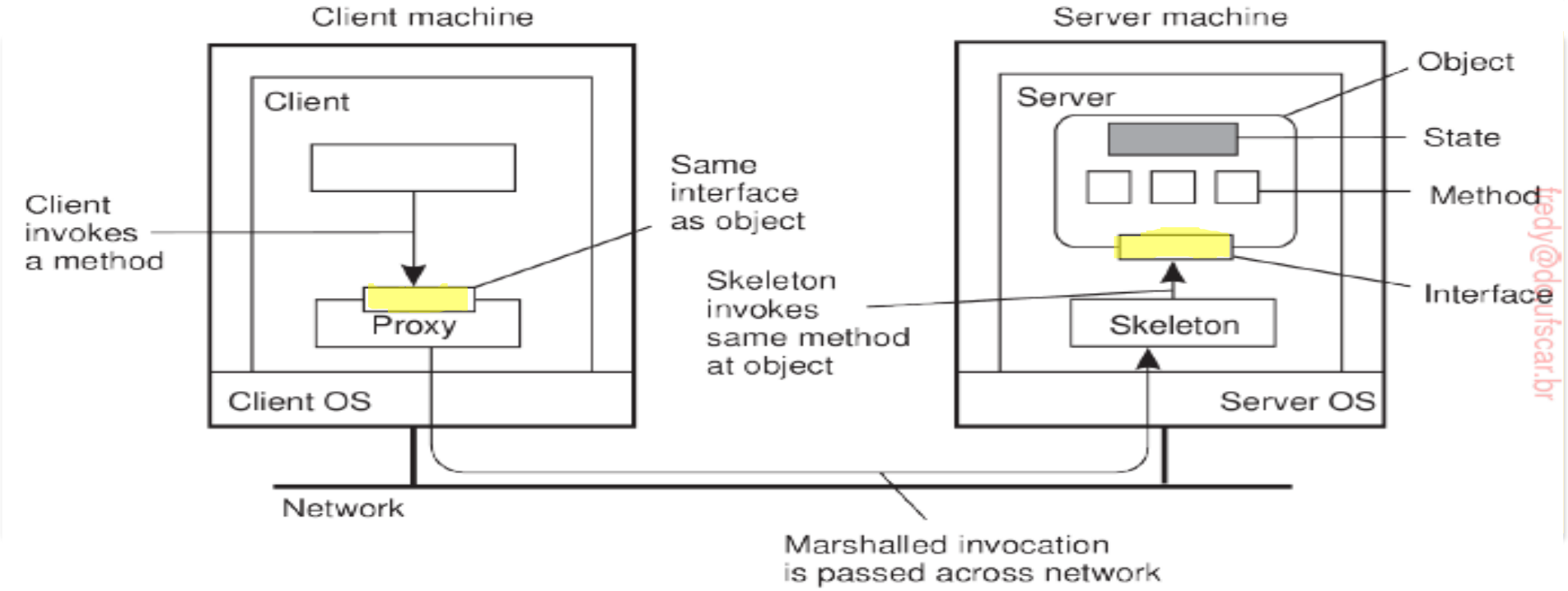
EXEMPLO: UM MOTOR DE BUSCAS SIMPLES



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



ENCAPSULAMENTO

Objetos encapsulam dados e oferecem métodos sobre estes dados sem revelar a implementação interna

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/"
```

CONCLUSÕES

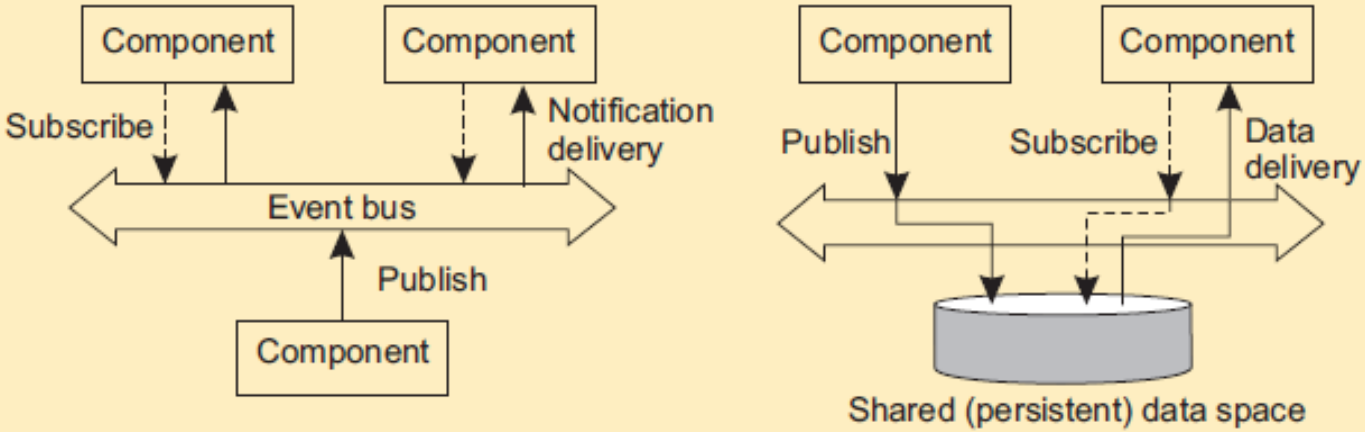
Há alguma a desenhar ?

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!"))
```

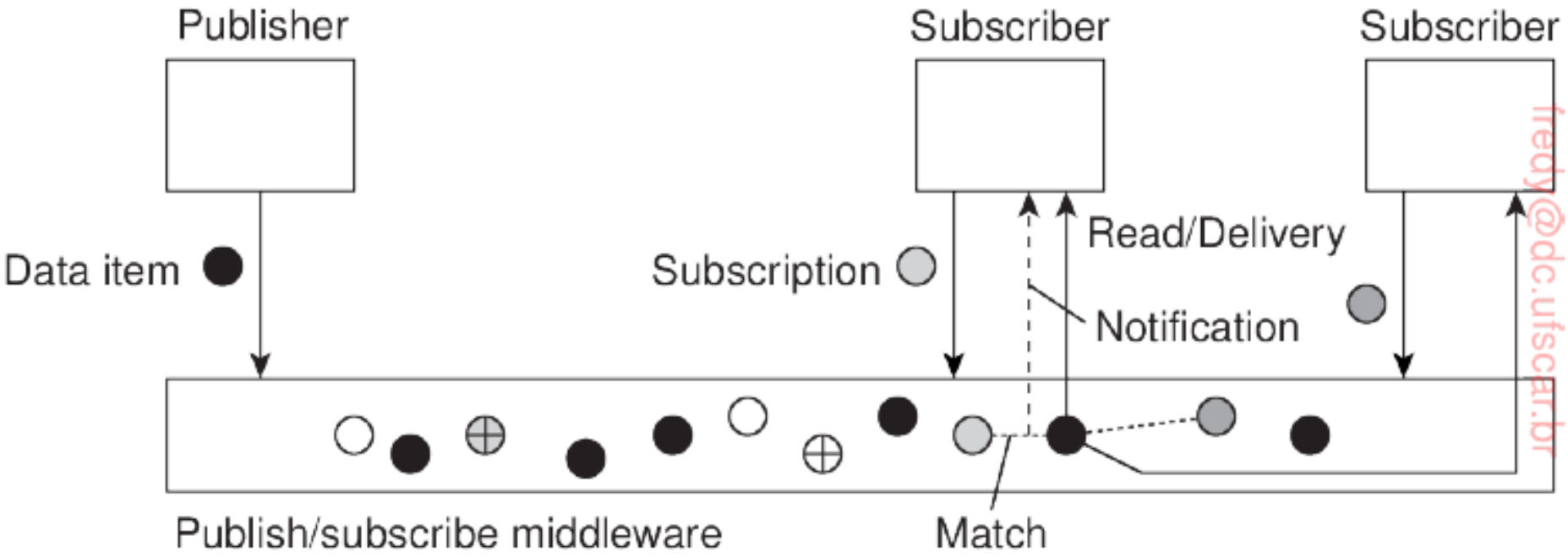
ALICE

```
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

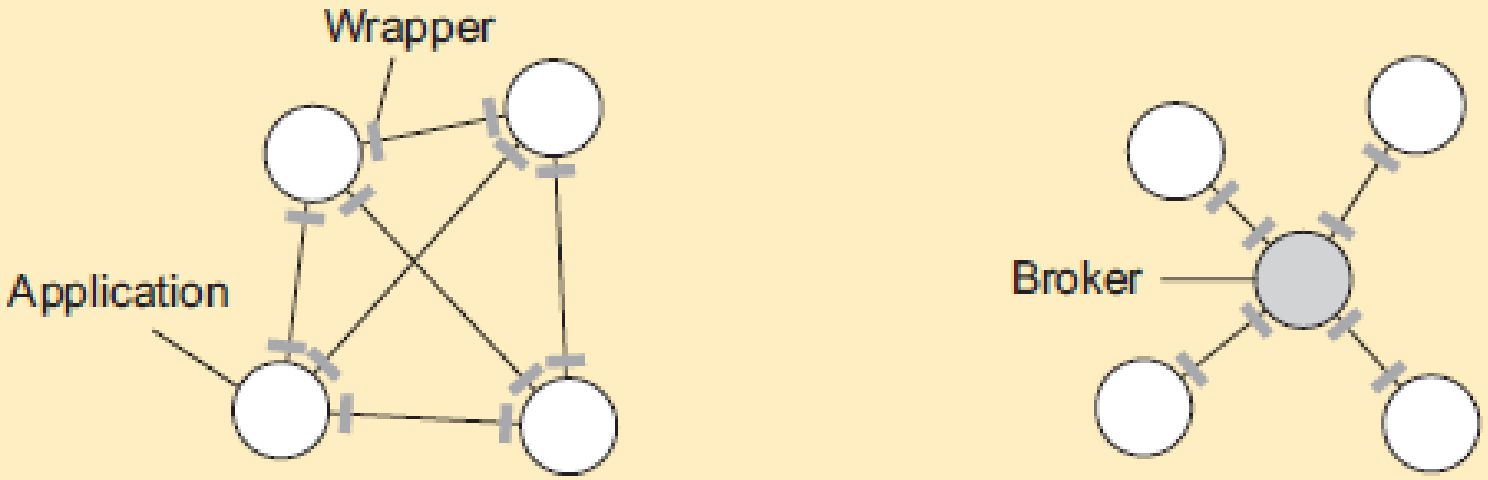
A interface oferecida por um componente legado provavelmente não será adequada para todas aplicações.

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

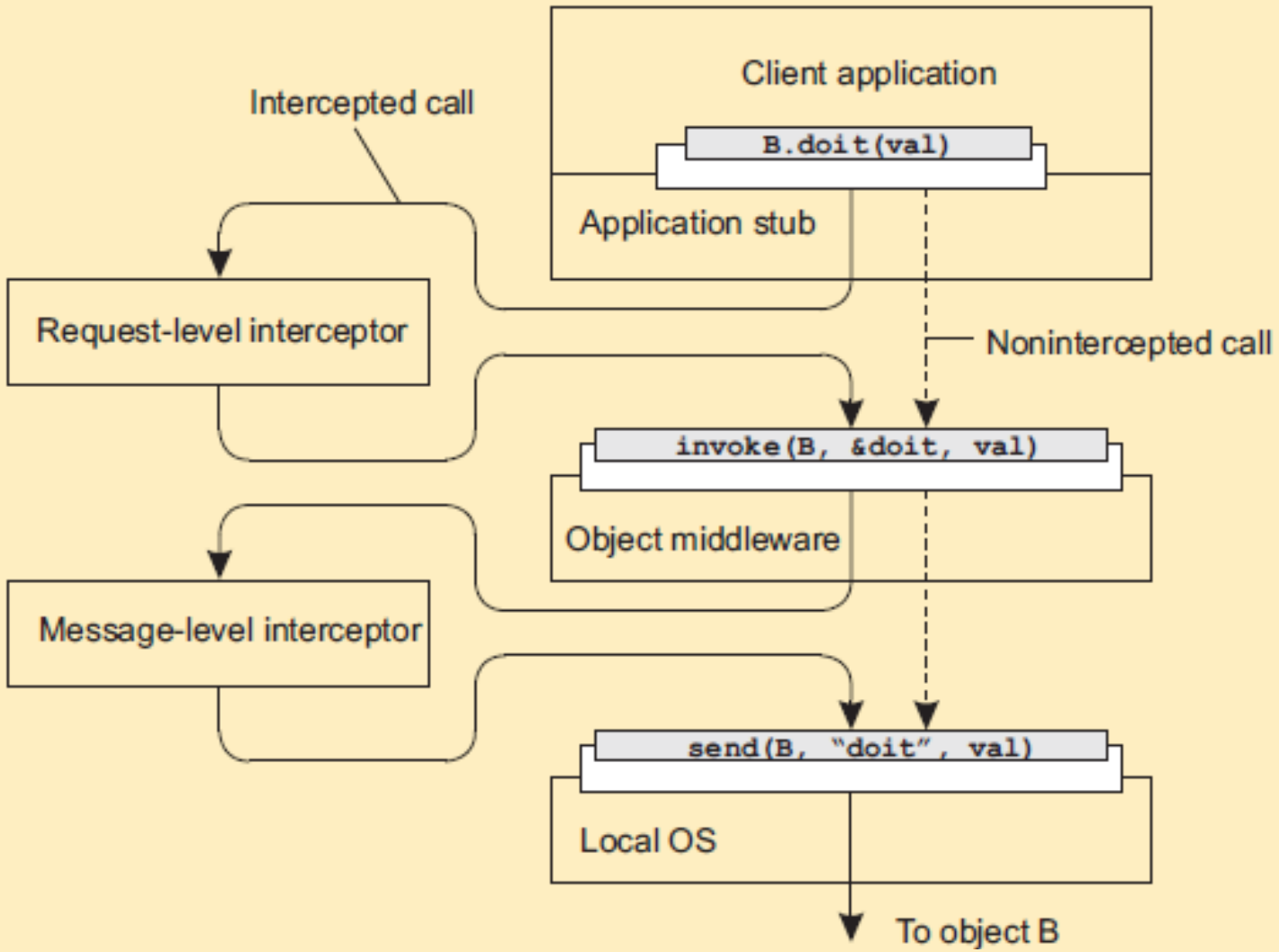
- Um pra um: requer $N \times (N-1) = \mathcal{O}(N^2)$ wrappers
- Broker: requer $2N = \mathcal{O}(N)$ wrappers

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

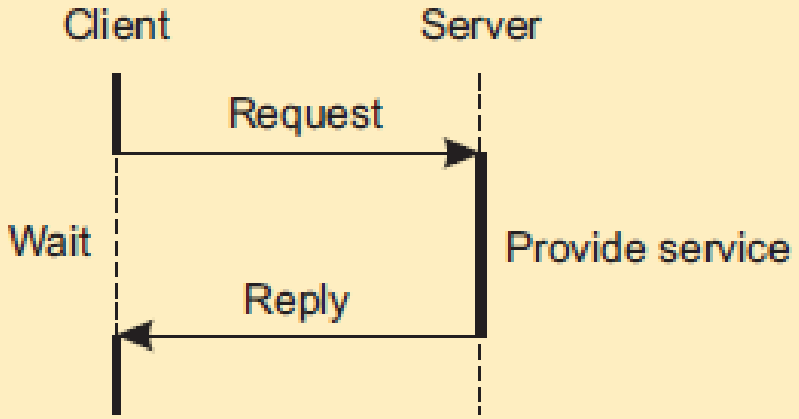


ARQUITETURA CENTRALIZADA DE SISTEMAS

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

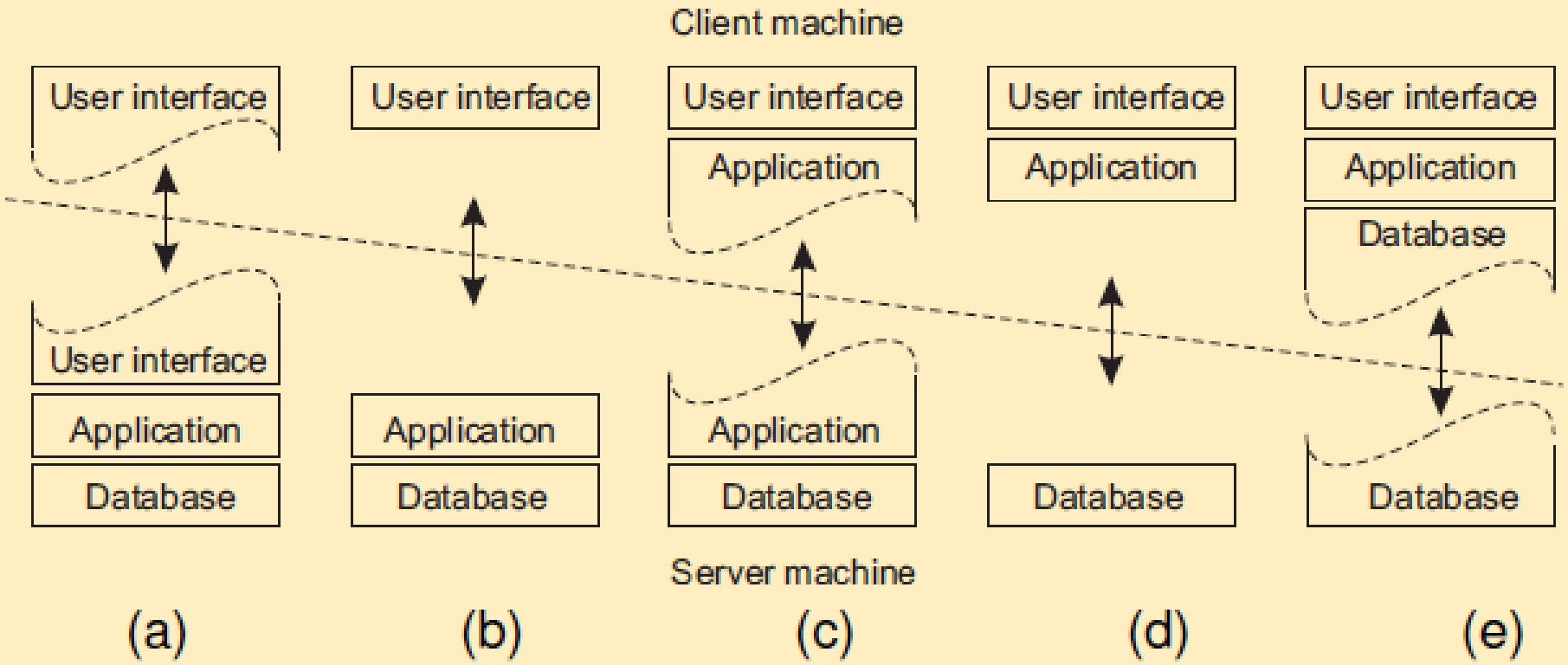


ARQUITETURA CENTRALIZADA DE SISTEMAS MULTI-TIERED

ALGUMAS ORGANIZAÇÕES TRADICIONAIS

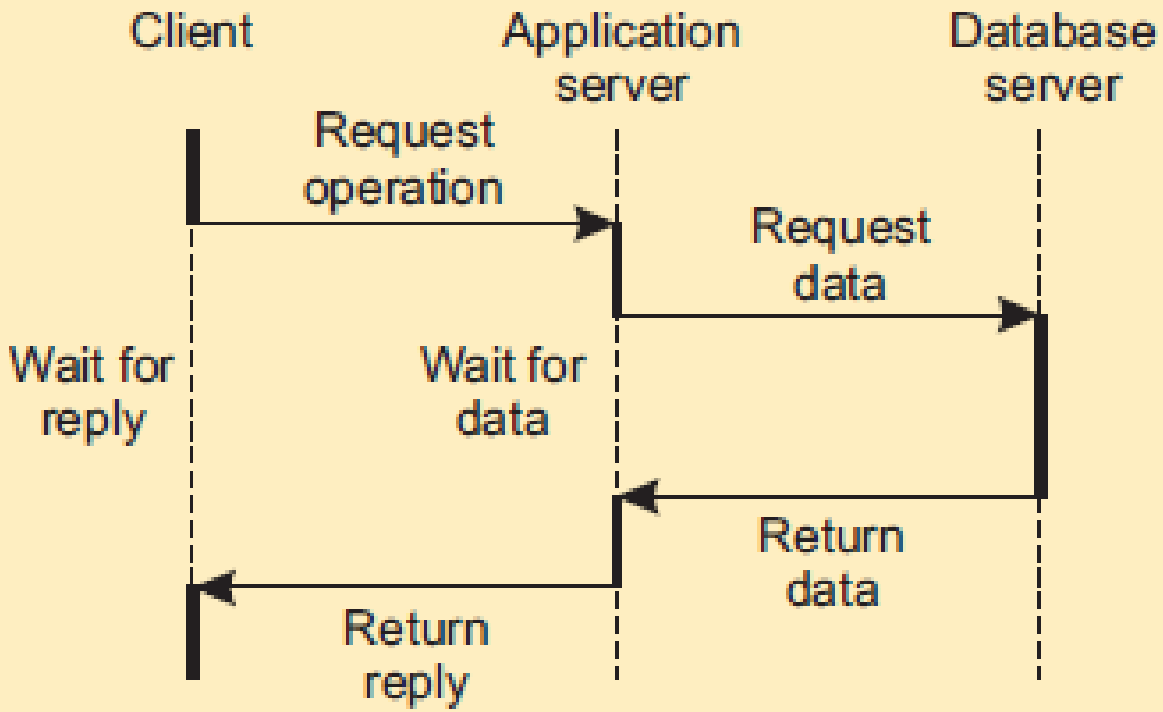
- Single-tiered: terminal burro / configuração mainframe
- Two-tiered: configuração cliente servidor de servidor único)
- Three-tiered: cada camada em uma máquina separada

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

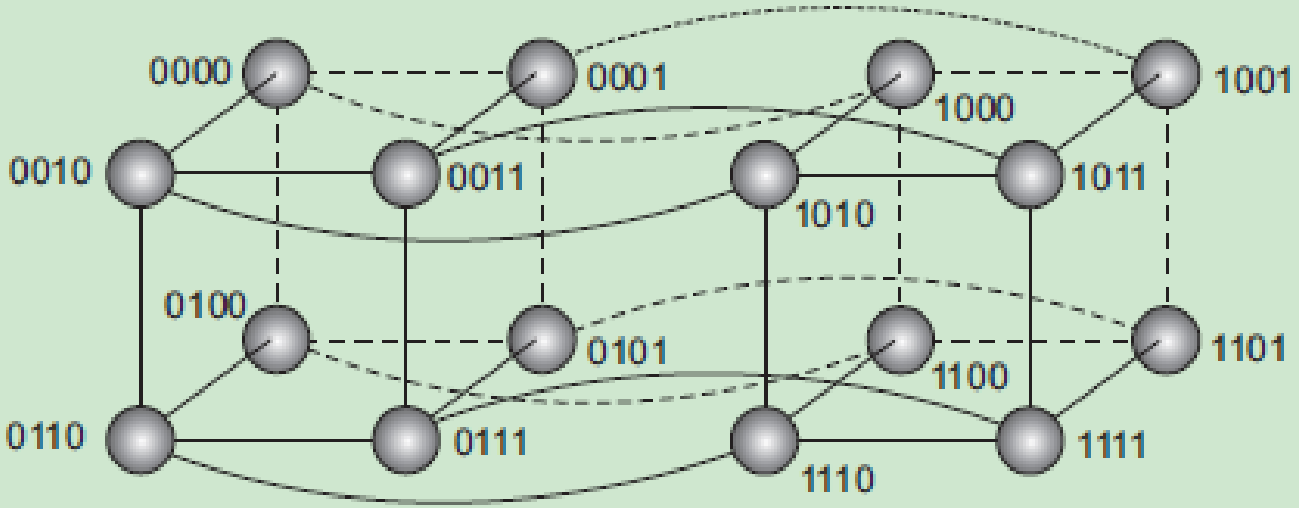
ESSÊNCIA

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



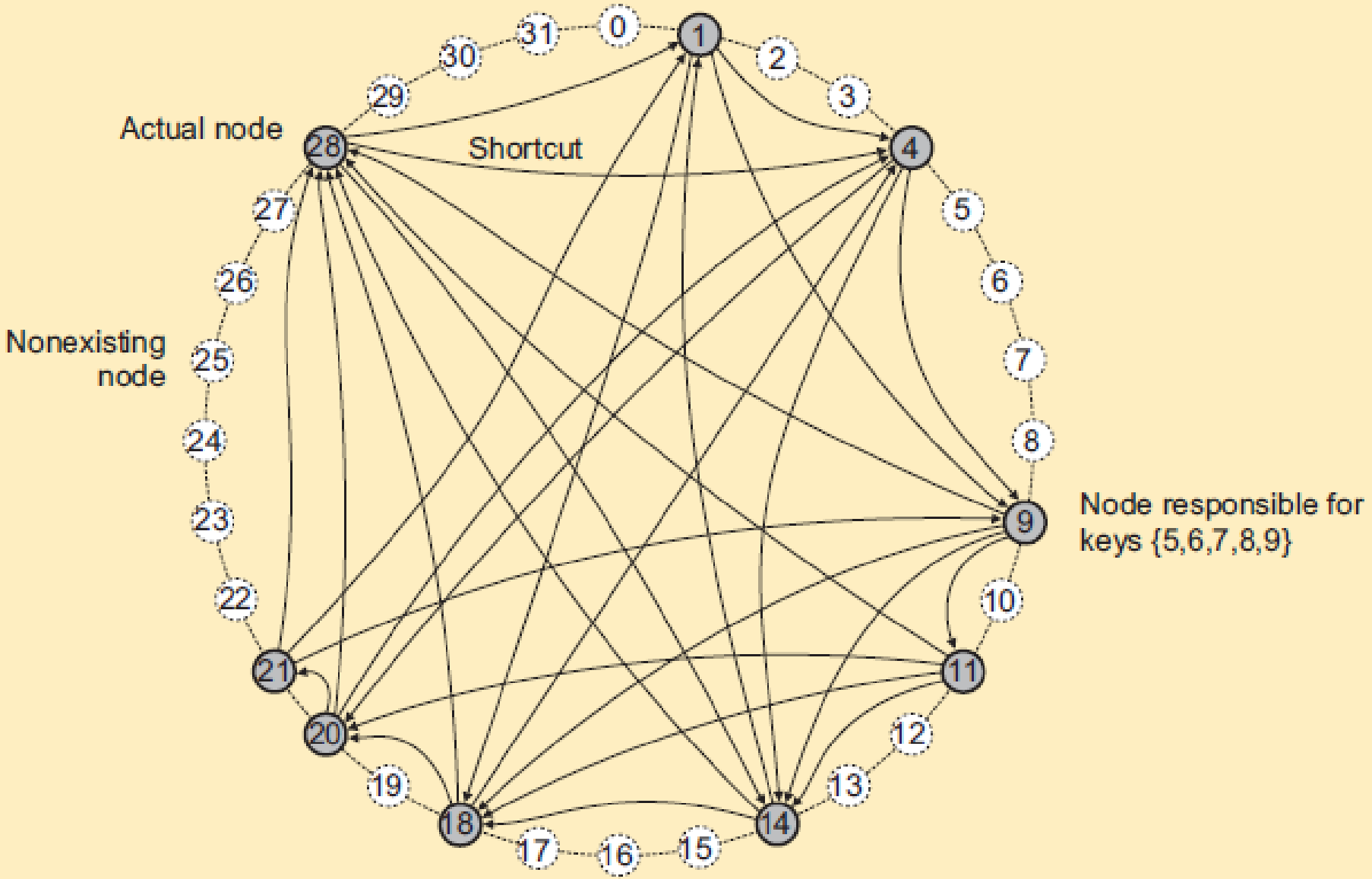
Procurando (looking up) *d* com chave $k \in \{0,1,2,\ldots,2^4-1\}$ significa o roteamento da requisição para o nó com identificador *k*

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* >= *k*, chamado de sucessor da chave *k*.
- O anel é extendido com vários links atalho para outros nós.

EXEMPLO CHORD



lookup(3)@9 : 28 → 1 → 4

P2P DESESTRUTURADO

ESSÊNCIA

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

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.

FLOODING X RANDOM WALK

MODELO

Assuma *N* nós e que cada item de dados é replicado através de *r* nós escolhidos randomicamente.

BUSCANDO

A probabilidade $\mathbb{P}[k]$ de um item ser achado depois de *k* tentativas é:

$$\mathbb{P}[k] = \frac{r}{N}(1 - \frac{r}{N})^{k-1}.$$

S ("search size") é o número de nós que precisams ser sondados é:

$$S = \sum_{k=1}^{N} k \cdot \mathbb{P}[k] = \sum_{k=1}^{N} k \cdot \frac{r}{N}(1 - \frac{r}{N})^{k-1} \approx N/r \text{ for } 1 \ll r \leq N.$$

FLOODING X RANDOM WALK

FLOODING

- Inunde *d* vizinhos escolhidos randomicamente
- Depois de *k* passos, algum $R(k) = d \cdot (d-1)^{k-1}$ terá sido atingido (assumindo *k* pequeno)
- Com uma fração r/N nós tendo dados, se $\frac{r}{N} \cdot R(k) \geq 1,$ então teremos achado o item de dados.

COMPARAÇÃO

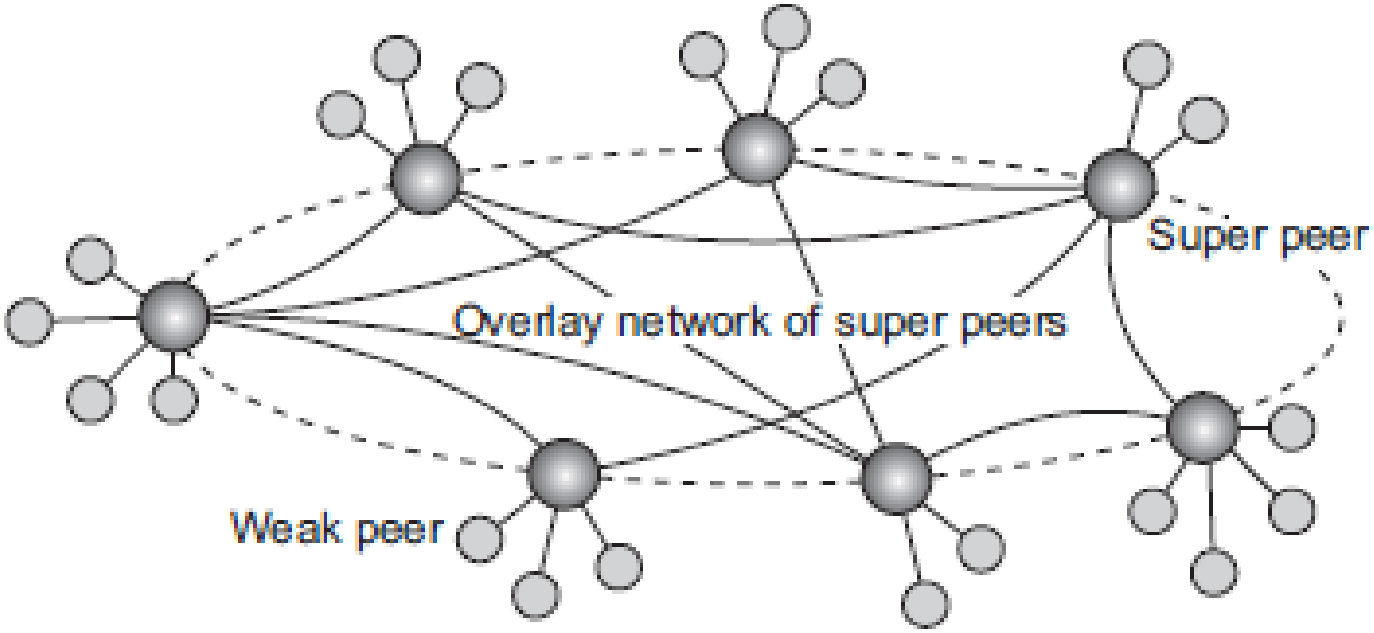
- Se $r/N = 0.001$, então $S \approx 1000$
- Com inundação (*flooding*) e *d = 10*, *k = 4*, contatamos *7290* nós
- *Random walks* são mais eficientes em comunicação, mas pode levar mais tempo para descobrir o resultado

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 existência de 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*

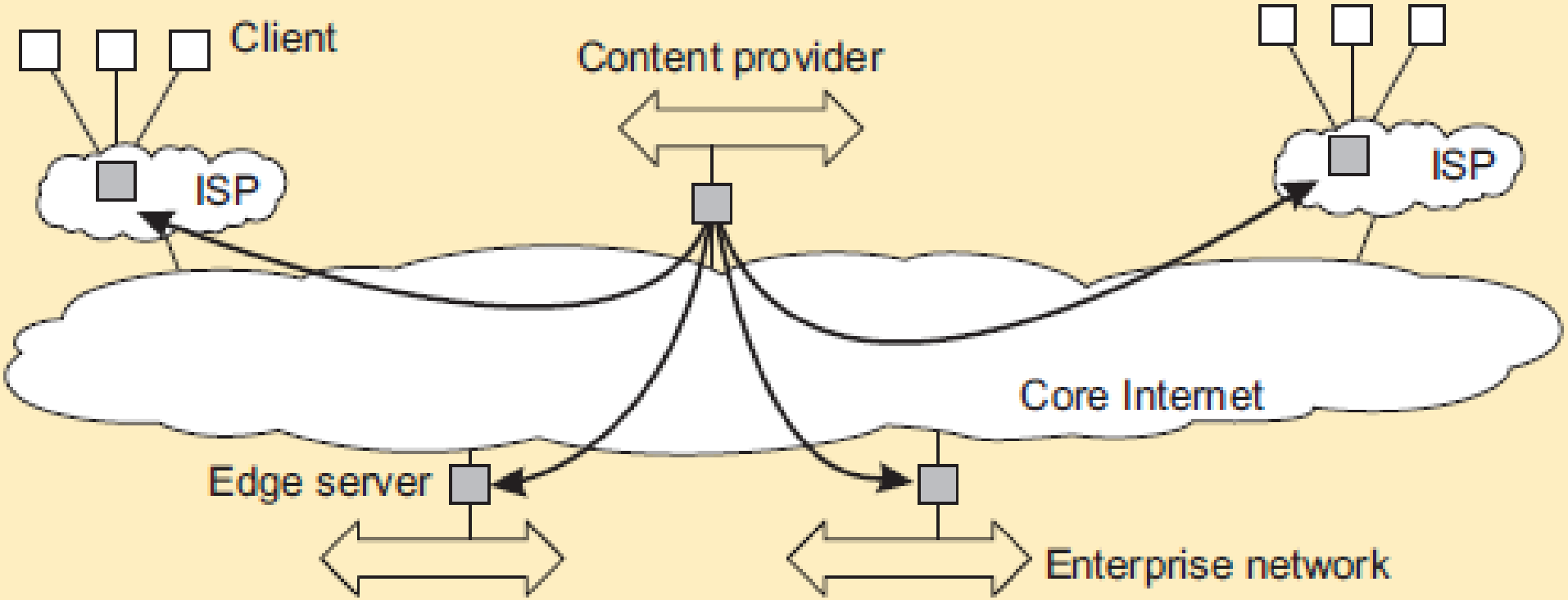
AMBOS *A* E *B* OPERAM ATRÁS DE UM FIREWALL

- *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

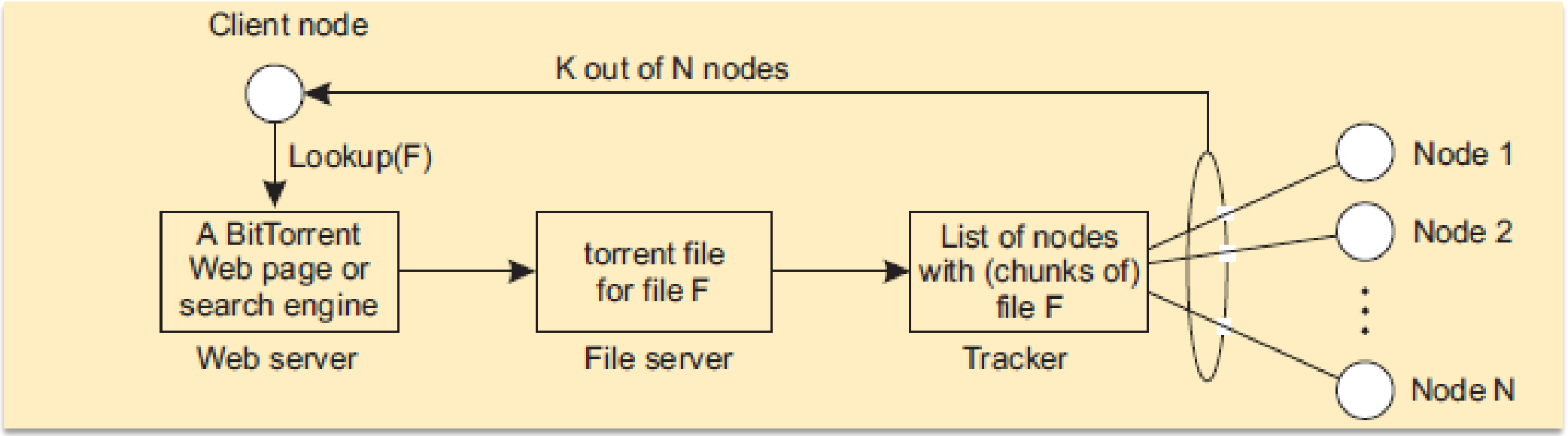
Sistemas disseminados na Internet onde os servidores são colocados na borda (*at the edge*) da rede: a fronteira entre redes empresariais e a Internet atual



COLABORAÇÃO O CASO BITTORRENT

PRINCÍPIO: BUSCA POR UM ARQUIVO F

- Procura de um arquivo em um diretório global => retorna um arquivo *torrent*
- Arquivo *torrent* contém referências para *tracker*: um servidor que mantém uma conta de nós ativos que possuem pedaços (*chunks*) de *F*.
- *P* pode se juntar a *swarm* (enxame), pegar um pedaço de graça, e então negociar uma cópia do *chunk* por outro *chunk* com um peer *Q* também no *swarm*.



BITTORRENT POR DEBAIXO DO CAPUZ

ALGUNS DETALHES ESSÊNCIAIS

- Um *tracker* para arquivo *F* retorna um conjunto de seus processos download: o *swarm* atual
- *A* comunica somente com um subconjunto do *swarm*: o conjunto vizinho N_A.
- Se $B \in N_A$ então $A \in N_B$
- Conjuntos vizinhos são regularmente atualizados pelo *tracker*

BLOCOS DE TROCA

- Um arquivo é dividido pedaços de tamanhos iguais (tipicamente com 256 KB)
- Peers trocam blocos de pedações, tipicamente de 16 KB
- *A* pode carregar um bloco *d* do pedaço *D*, somente se ele tem o pedaço *D*.
- Vizinho *B* pertence ao conjunto potencial P_A de *A*, se *B* tem o bloco que *A* precisa.
- Se $B \in P_A$ e $A \in P_B$: *A* e *B* estão em uma posição na qual eles podem negociar um bloco.

FASES DO BITTORRENT

FASE BOOTSTRAP

A acabou de receber seu primeiro pedaço (através de desafogamento otimista – *optimistic unchoking*): um nó de N_A não egoísta provê os blocos de um pedaço para pegar um nó iniciado que acabou de chegar

FASE NEGOCIAÇÃO

$|P_A| > 0$: existe (em princípio) sempre um peer com o qual *A* pode negociar

ÚLTIMA FASE DOWNLOAD

$|P_A| = 0$: *A* é dependente de novos peers que chegam em N_A, de forma a pegar as últimas partes faltantes. N_A pode mudar somente através do tracker

FASES DO BITTORRENT

DESENVOLVIMENTO DE |P| RELATIVO A |N|