MCTA025-13 - SISTEMAS DISTRIBUÍDOS

ARQUITETURAS DE SISTEMAS DISTRIBUÍDOS

Emilio Francesquini

18 de junho de 2018

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

DISCLAIMER

- Estes slides foram preparados para o curso de **Sistemas Distribuídos na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro "Distributed Systems", 3ª Edição em: `https://www.distributed-systems.net`.

ARQUITETURAS

- Estilos arquiteturais
- Arquiteturas de software
- Arquiteturas versus middleware
- Sistemas distribuídos autogerenciáveis

Ideia básica

Um estilo é definido em termos de:

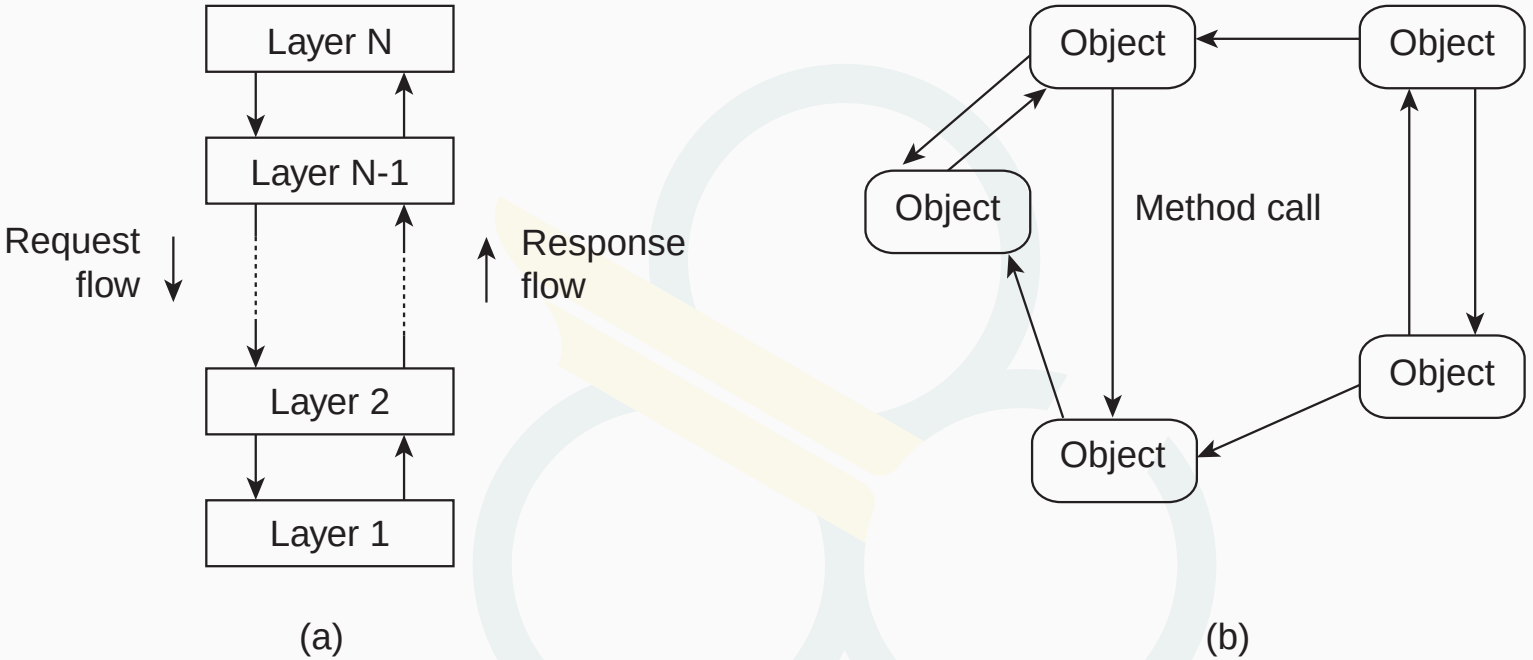
- componentes (substituíveis) com interfaces bem definidas
- o modo como os componentes são conectados entre si
- como os dados são trocados entre componentes
- como esses componentes e conectores são configurados conjuntamente em um sistema

Conector

Um mecanismo que intermedeia comunicação, coordenação ou cooperação entre componentes. Exemplo: recursos para chamadas de procedimento (remotos), mensagens ou *streaming*.

Ideia básica

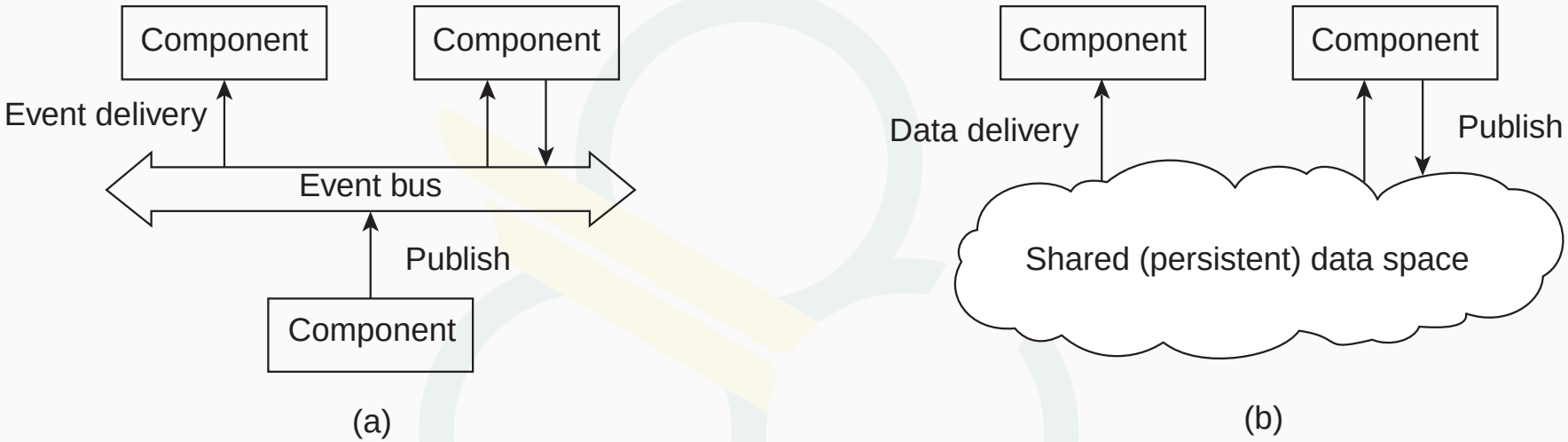
Organize em componentes logicamente diferentes e os distribua entre as máquinas disponíveis.



(a) Estilo em camadas é usado em sistemas cliente-servidor
(b) Estilo orientado a objetos usado em sistemas de objetos distribuídos.

Observação

Desacoplar processos no espaço (anônimos) e tempo (assíncronos) pode levar a estilos diferentes.

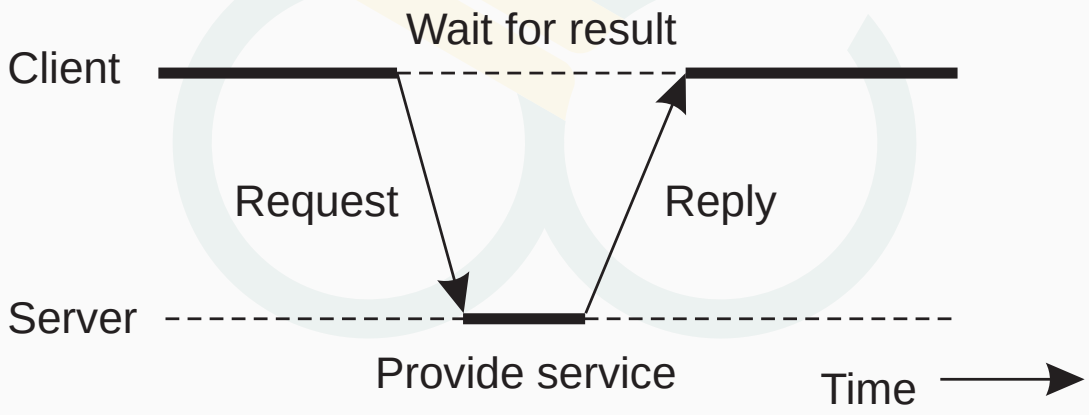


(a) Publish/subscribe [desaclopado no espaço]

(b) Espaço de dados compartilhados [desacoplado no espaço e tempo]

Características do modelo Cliente–Servidor

- Existem processos que oferecem serviços (servidores)
- Existem processos que usam esses serviços (clientes)
- Clientes e servidores podem estar em máquinas diferentes
- Clientes seguem um modelo requisição/resposta ao usar os serviços



Visão tradicional em três camadas

- A camada de apresentação contém o necessário para a aplicação poder interagir com o usuário
- A camada de negócio contém as funções de uma aplicação
- A camada de dados contém os dados que o cliente quer manipular através dos componentes da aplicação

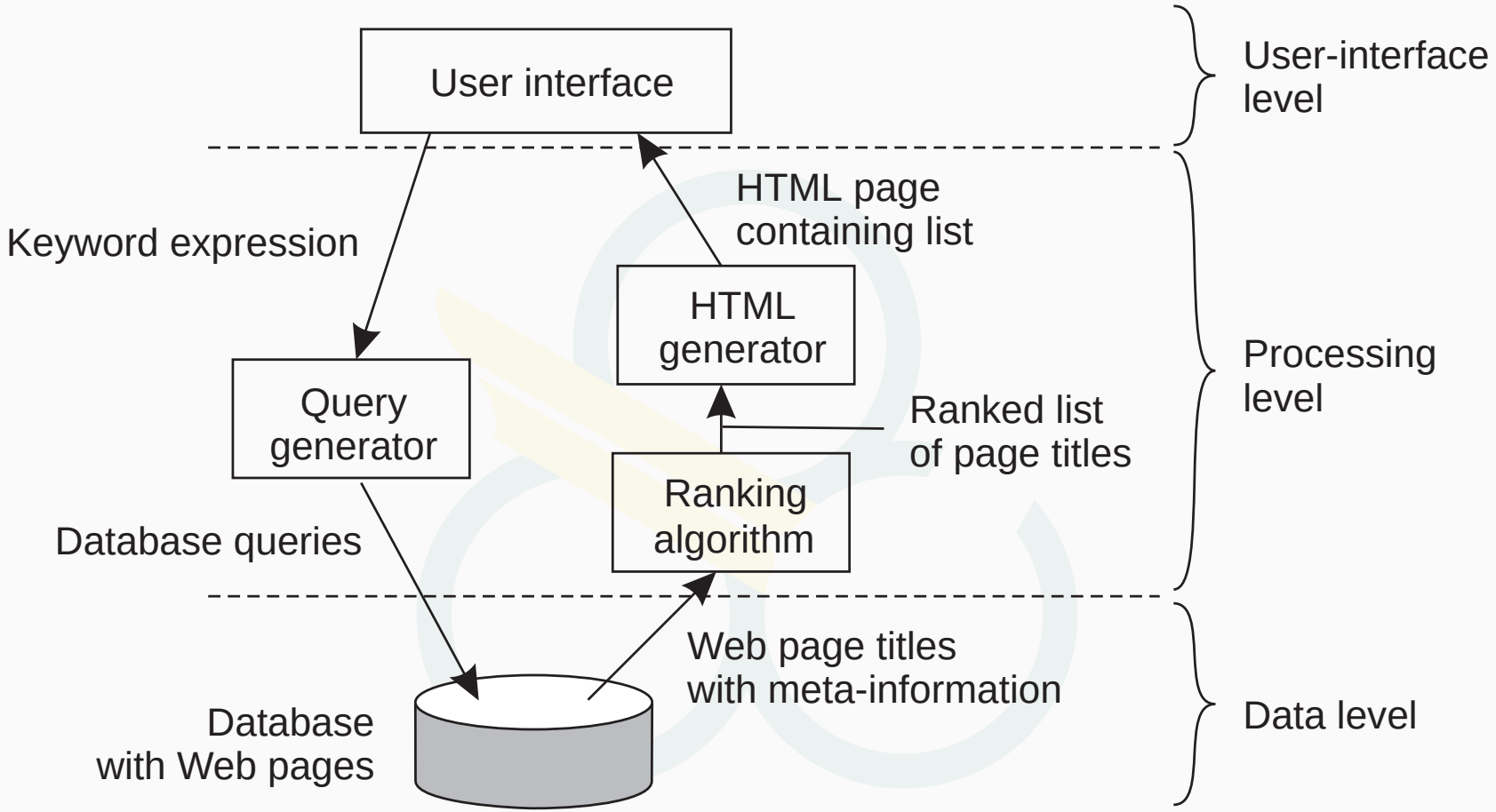
Visão tradicional em três camadas

- A camada de apresentação contém o necessário para a aplicação poder interagir com o usuário
- A camada de negócio contém as funções de uma aplicação
- A camada de dados contém os dados que o cliente quer manipular através dos componentes da aplicação

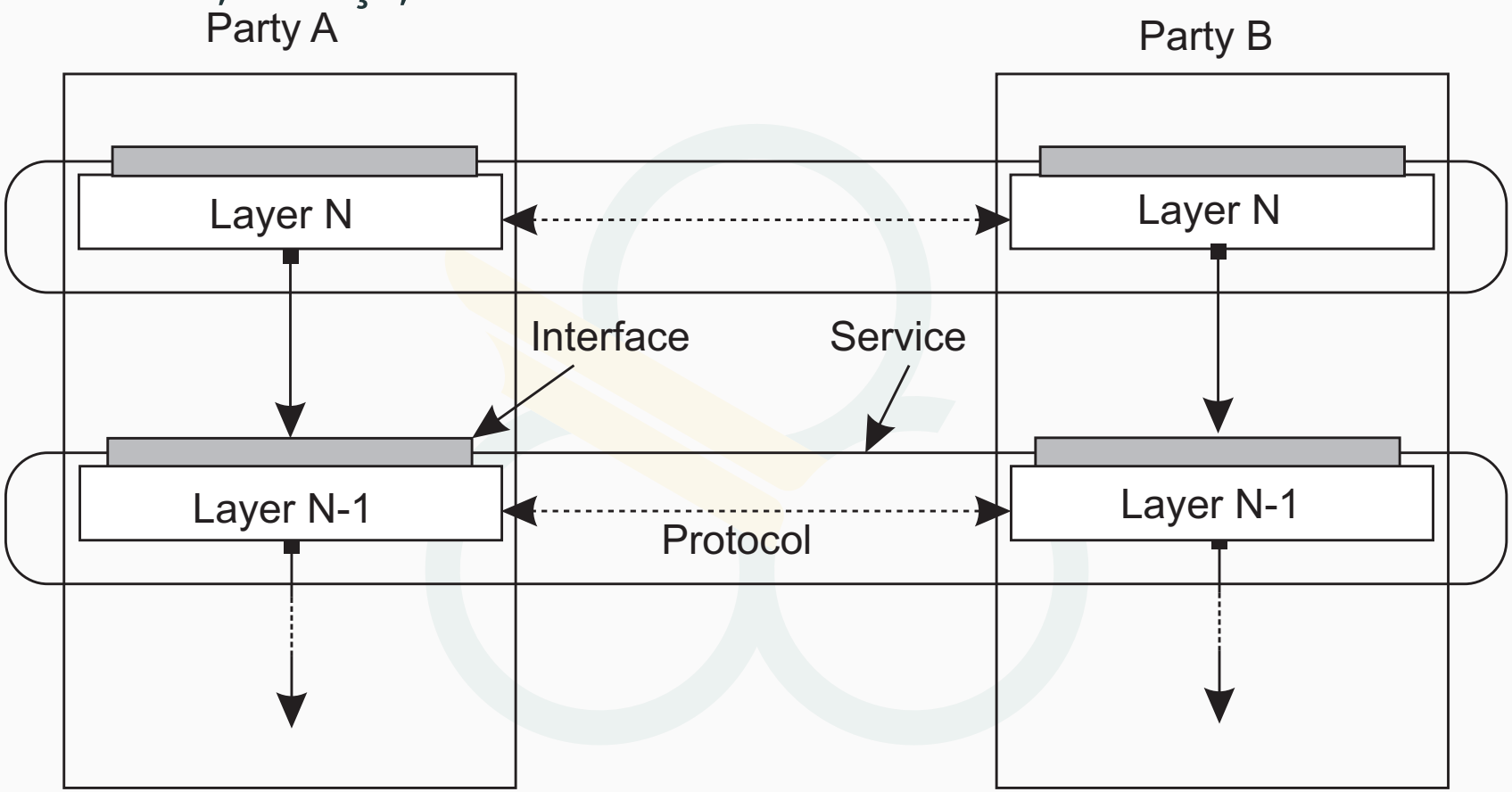
Observação

Estas camadas são encontradas em muitos sistemas de informação distribuídos, que usam tecnologias de bancos de dados tradicionais e suas aplicações auxiliares.

ARQUITETURAS MULTICAMADA



Protocolo, serviço, interface



Servidor

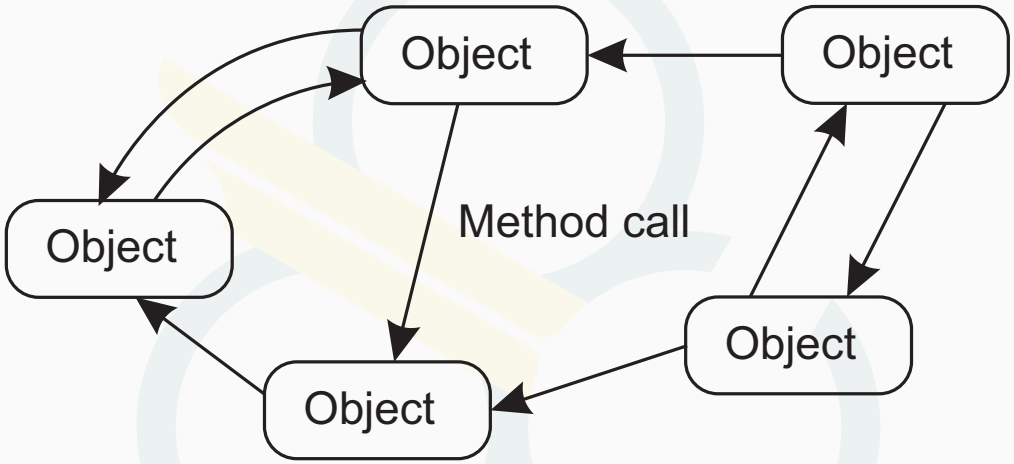
```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
(conn, addr) = s.accept()  # returns new socket and addr. client
while True:                # forever
  data = conn.recv(1024)   # receive data from client
  if not data: break       # stop if client stopped
  conn.send(str(data)+"*") # return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

Cliente

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

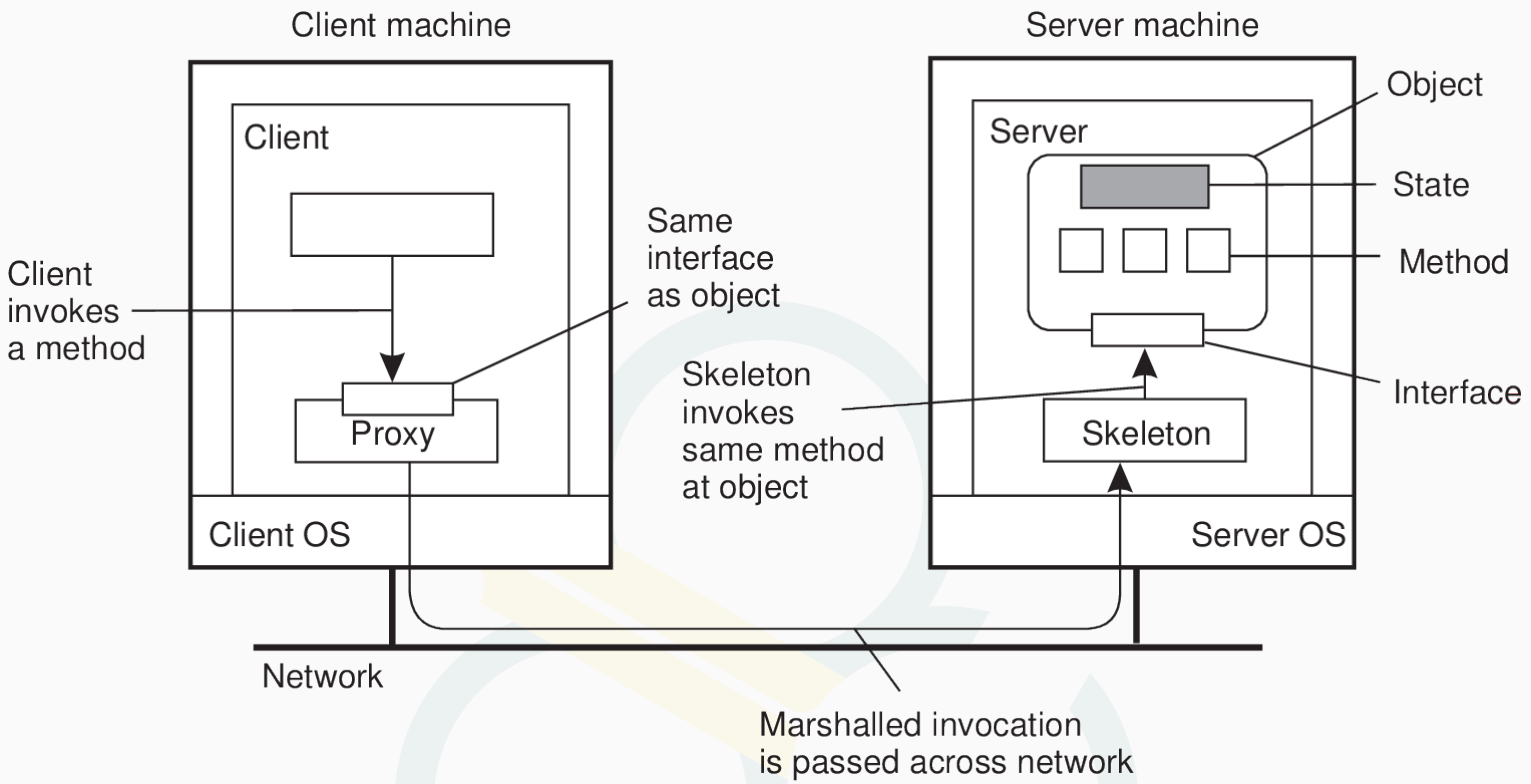
Essência

Componentes são objetos, conectados entre si usando chamadas de procedimentos. Objetos podem ser colocados em máquinas diferentes; chamadas, por tanto, devem executar usando a rede.



Encapsulamento

Dizemos que um objeto *encapsula dados* e oferece *métodos para os dados* sem revelar sua implementação.



Encapsulamento

Os objetos (e consequentemente dados e comportamentos) ficam distribuídos pelo sistema. Apesar do usuário fazer chamadas que são equivalentes a chamadas locais, elas podem estar sendo feitas em objetos remotos.

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

1. Recursos são identificados usando um único esquema de nomeação
2. Todos os serviços oferecem a mesma interface
3. Mensagens enviadas de ou para um serviço são auto-descritivas
4. Após a execução de uma operação em um serviço, o componente esquece tudo sobre quem chamou a operação

Operações básicas

Operação	Descrição
PUT	Cria um novo recurso
GET	Recupera o estado de um recurso usando um tipo de representação
DELETE	Apaga um recurso
POST	Modifica um recurso ao transferir um novo estado

Essência

Objetos (arquivos) são armazenados em buckets (diretórios). Buckets não podem ser colocados dentro de outros buckets. Operações em `ObjectName` em `BucketName` requerem o seguinte identificador:

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

Operações típicas

Todas as operações são realizadas com requisições HTTP:

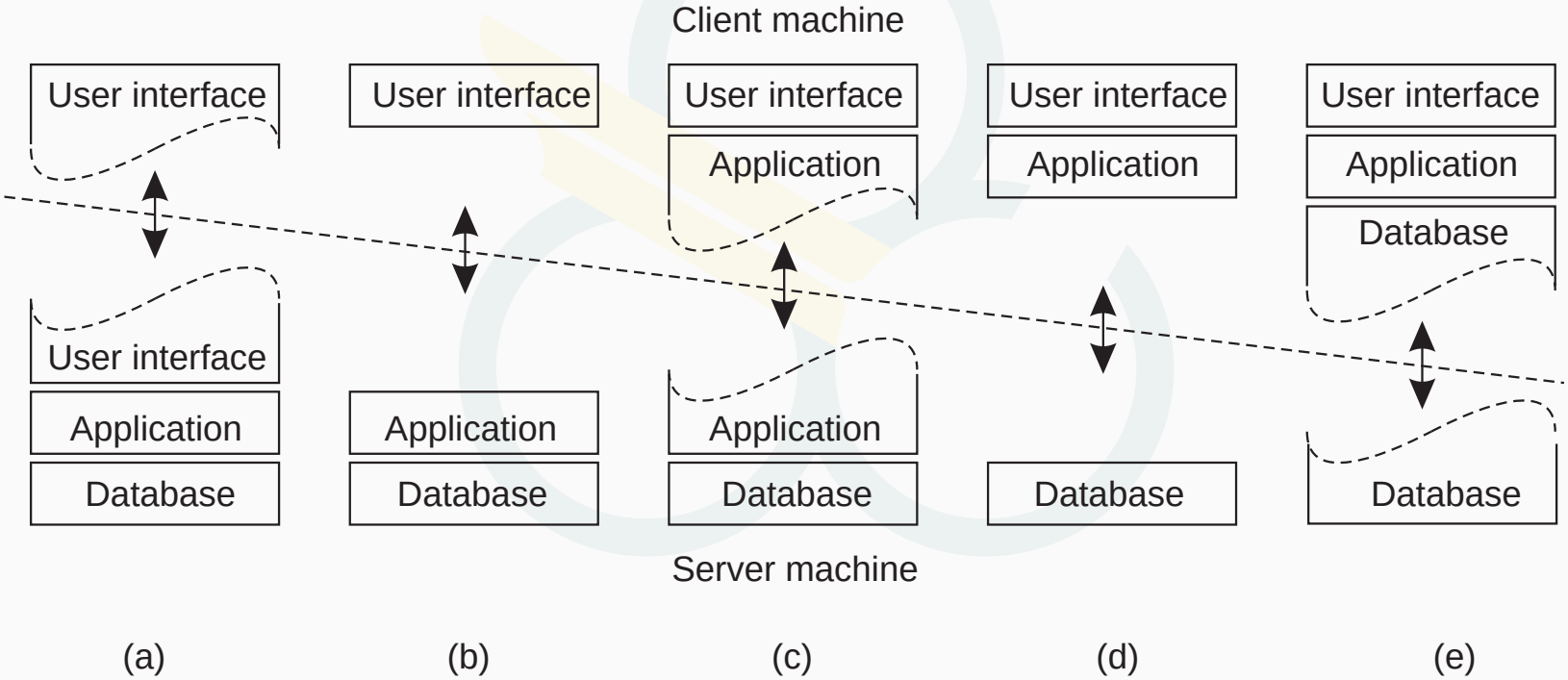
- Criar um bucket/objeto: `PUT` + URI
- Listar objetos: `GET` em um nome de bucket
- Ler um objeto: `GET` em uma URI completa

uma camada: configurações de terminal burro/mainframe

duas camadas: configuração cliente–servidor único.

três camadas: cada camada em uma máquina separada

Configurações tradicionais em duas camadas físicas:



P2P estruturado os nós são organizados seguindo uma estrutura de dados distribuída específica

P2P não-estruturado os nós selecionam aleatoriamente seus vizinhos

P2P híbrido alguns nós são designados, de forma organizada, a executar funções especiais

P2P estruturado os nós são organizados seguindo uma estrutura de dados distribuída específica

P2P não-estruturado os nós selecionam aleatoriamente seus vizinhos

P2P híbrido alguns nós são designados, de forma organizada, a executar funções especiais

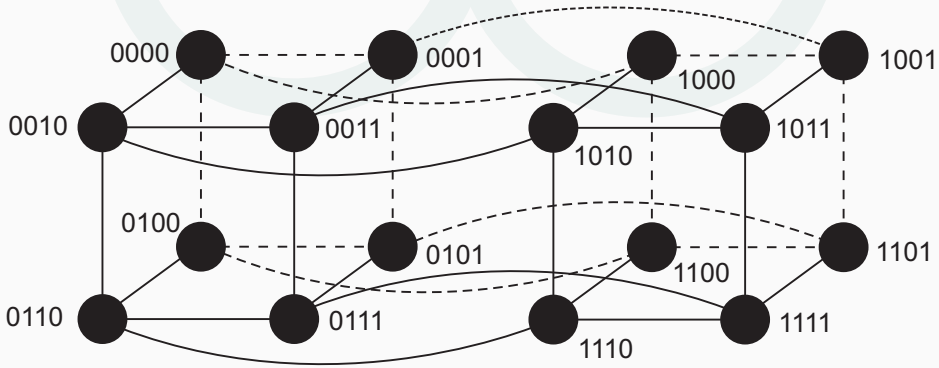
Nota:
Praticamente todos os casos são exemplos de redes overlay: dados são roteados usando conexões definidas pelos nós (*Cf.* multicast em nível de aplicação)

- A ideia é utilizar um índice não baseado na semântica dos dados: cada conjunto de dados é associado unicamente a uma chave que, por sua vez, é usada como índice. A maneira mais comum de fazer isto é através de uma função de hash.
  - `chave(dado) = hash(dado)`

  O sistema P2P passa então a ser responsável apenas por associar chaves a valores ou , de maneira equivalente, lidar apenas com **pares (chave, valor)**.

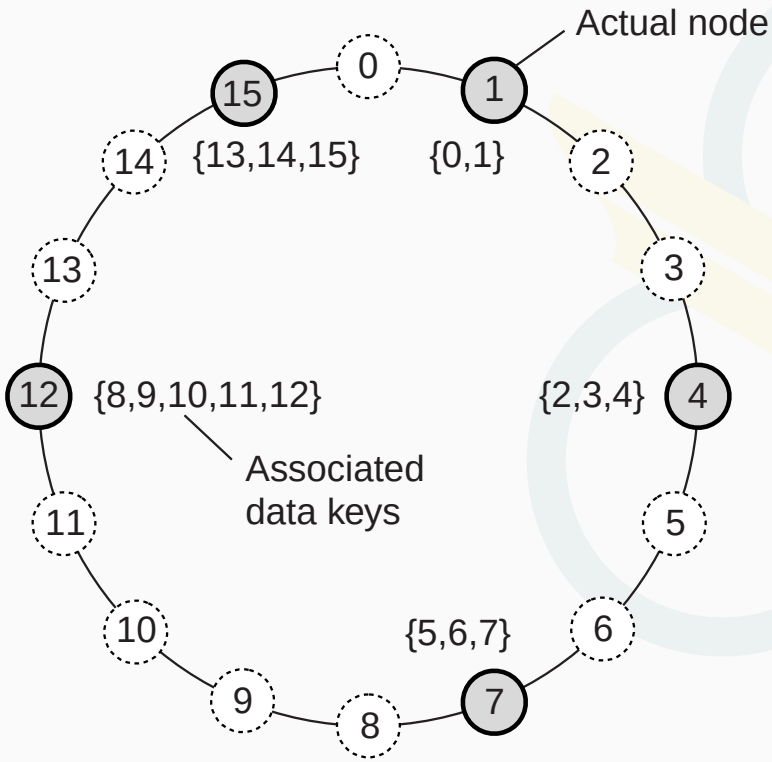
Exemplo simples: Hipercubo:

A procura por um dado d com chave $k \in \{0, 1, 2, ...2^4 - 1\}$ causa o roteamento da busca para o nó com identificador k.



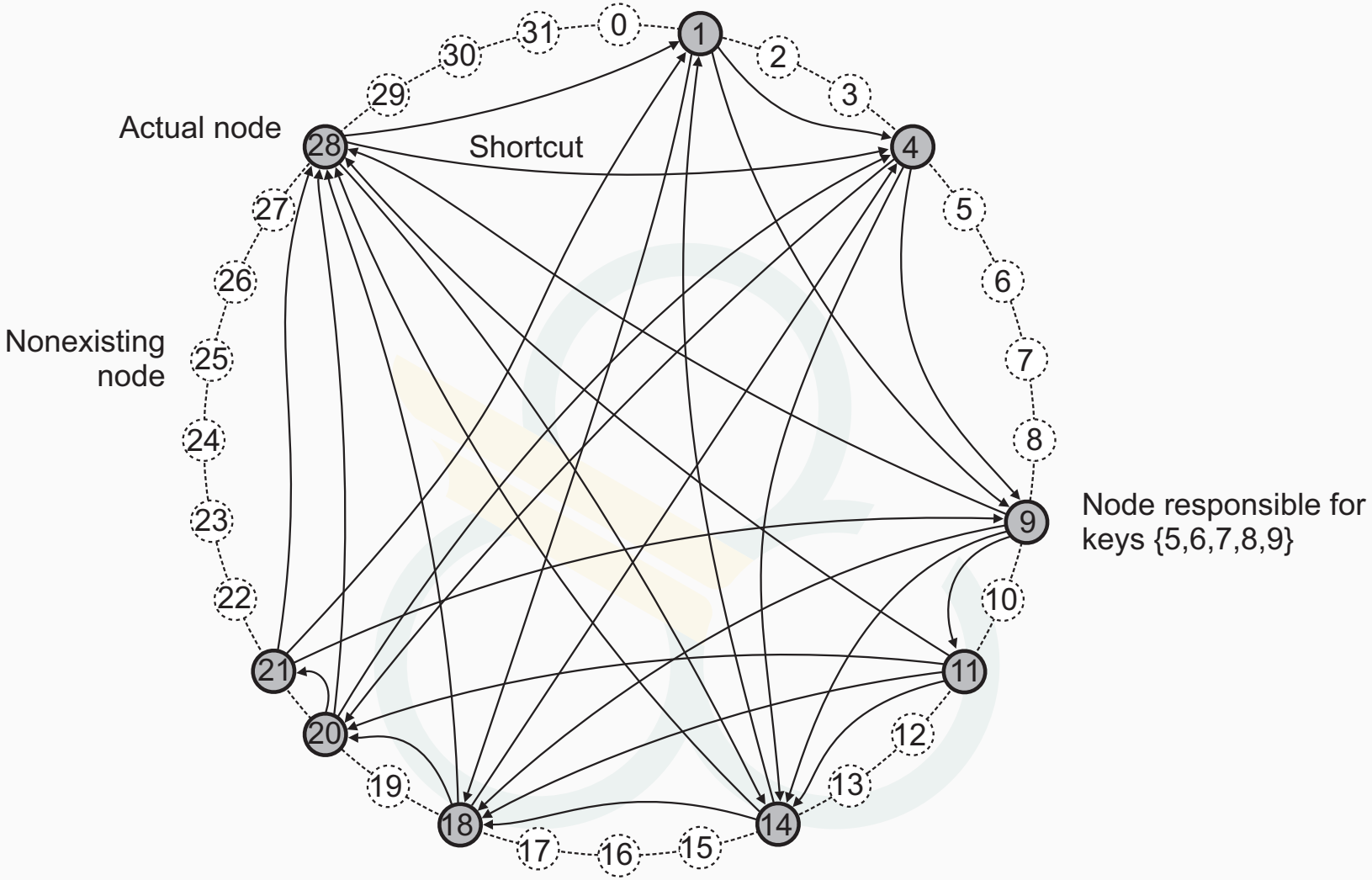
Ideia básica

Organizar os nós em uma rede overlay estruturada, tal como um anel lógico, e fazer com que alguns nós se tornem responsáveis por alguns serviços baseado unicamente em seus IDs.



Nota

O sistema provê uma operação `LOOKUP(key)` que irá fazer o roteamento de uma requisição até o nó correspondente.

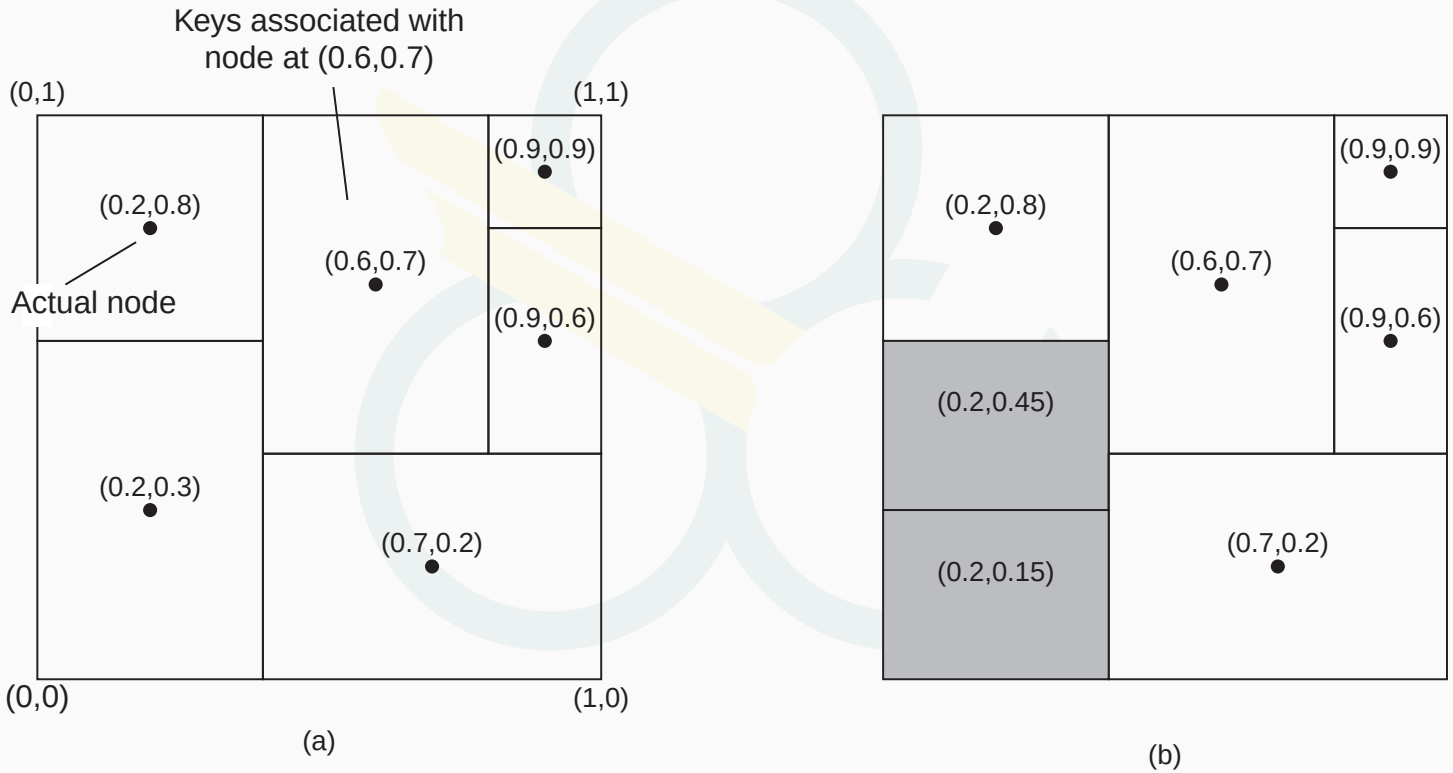


$lookup(3)@9 : 28 \rightarrow 1 \rightarrow 4$

Outro exemplo

Organize os nós em um espaço d-dimensional e faça todos os nós ficarem responsáveis por um dado em uma região específica.

Quando um nó for adicionado, divida a região.



Observação
Muitos sistemas P2P não-estruturados tentam manter um grafo aleatório.

Princípio básico
Cada nó deve contactar um outro nó selecionado aleatoriamente:

- Cada participante mantém uma visão parcial da rede, consistindo de c outros nós
- Cada nó P seleciona periodicamente um nó Q de sua visão parcial
- P e Q trocam informação && trocam membros de suas respectivas visões parciais

Nota
Dependendo de como as trocas são feitas, não só a aleatoriedade mas também a robustez da rede pode ser garantida.

O QUE É GOSSIPING?

Thread ativa

Thread passiva

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
```

Thread passiva

selectPeer: Seleciona aleatoriamente um vizinho de sua visão parcial.

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
selectToSend(&bufs);
```

Thread passiva

selectPeer:	Seleciona aleatoriamente um vizinho de sua visão parcial.
selectToSend:	Seleciona s entradas de seu cache local.

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
selectToSend(&bufs);
sendTo(B, bufs);
```

Thread passiva

```
receiveFromAny(&A, &bufr);
```

selectPeer:	Seleciona aleatoriamente um vizinho de sua visão parcial.
selectToSend:	Seleciona s entradas de seu cache local.

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
selectToSend(&bufs);
sendTo(B, bufs);
```

Thread passiva

```
receiveFromAny(&A, &bufr);
selectToSend(&bufs);
```

selectPeer:	Seleciona aleatoriamente um vizinho de sua visão parcial.
selectToSend:	Seleciona s entradas de seu cache local.

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
selectToSend(&bufs);
sendTo(B, bufs);

receiveFrom(B, &bufr);
```

Thread passiva

```
receiveFromAny(&A, &bufr);
selectToSend(&bufs);
sendTo(A, bufs);
```

selectPeer:	Seleciona aleatoriamente um vizinho de sua visão parcial.
selectToSend:	Seleciona s entradas de seu cache local.

O QUE É GOSSIPING?

Thread ativa

```
selectPeer(&B);
selectToSend(&bufs);
sendTo(B, bufs);

receiveFrom(B, &bufr);
selectToKeep(cache, bufr);
```

Thread passiva

```
receiveFromAny(&A, &bufr);
selectToSend(&bufs);
sendTo(A, bufs);
selectToKeep(cache, bufr);
```

selectPeer:	Seleciona aleatoriamente um vizinho de sua visão parcial.
selectToSend:	Seleciona s entradas de seu cache local.
selectToKeep:	(1) Adiciona as entradas recebidas ao cache local. (2) Remove os itens repetidos. (3) Encolhe o tamanho do cache para c (usando alguma estratégia).

FUNDAMENTO: AMOSTRAGEM DE PEERS USANDO GOSSIP

Unifica a visão parcial e o cache local $\Rightarrow$ troca os vizinhos

Thread ativa	Thread passiva

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa	Thread passiva
`selectPeer(&B);`	

selectPeer: Seleciona aleatoriamente um vizinho.

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa	Thread passiva

```
selectPeer(&B);
selectToSend(&peers_s);
```

selectPeer:	Seleciona aleatoriamente um vizinho.
selectToSend:	Seleciona s referências a vizinhos.

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa	Thread passiva
`selectPeer(&B);`	
`selectToSend(&peers_s);`	
`sendTo(B, peers_s);`	`receiveFromAny(&A, &peers_r);`

selectPeer:	Seleciona aleatoriamente um vizinho.
selectToSend:	Seleciona s referências a vizinhos.

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa	Thread passiva
`selectPeer(&B);`	
`selectToSend(&peers_s);`	
`sendTo(B, peers_s);`	`receiveFromAny(&A, &peers_r);`
	`selectToSend(&peers_s);`

selectPeer:	Seleciona aleatoriamente um vizinho.
selectToSend:	Seleciona s referências a vizinhos.

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa	Thread passiva
`selectPeer(&B);`	
`selectToSend(&peers_s);`	
`sendTo(B, peers_s);`	`receiveFromAny(&A, &peers_r);`
	`selectToSend(&peers_s);`
`receiveFrom(B, &peers_r);`	`sendTo(A, peers_s);`

selectPeer:	Seleciona aleatoriamente um vizinho.
selectToSend:	Seleciona s referências a vizinhos.

Unifica a visão parcial e o cache local ⇒ troca os vizinhos

Thread ativa

```
selectPeer(&B);
selectToSend(&peers_s);
sendTo(B, peers_s);

receiveFrom(B, &peers_r);
selectToKeep(pview, peers_r);
```

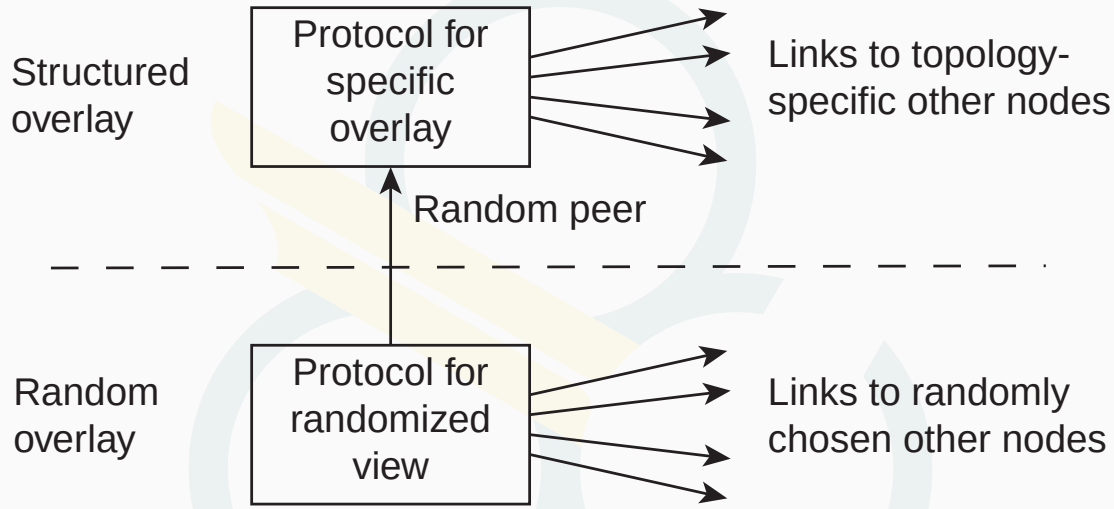
Thread passiva

```
receiveFromAny(&A, &peers_r);
selectToSend(&peers_s);
sendTo(A, peers_s);
selectToKeep(pview, peers_r);
```

selectPeer:	Seleciona aleatoriamente um vizinho.
selectToSend:	Seleciona s referências a vizinhos.
selectToKeep:	(1) Adiciona as referências recebidas à visão parcial. (2) Remove as refs. repetidas. (3) Encolhe o tamanho da visão para c, removendo aleatoriamente as refs enviadas (mas nunca as recebidas).

Ideia básica

Distinguir duas camadas: (1) mantém uma visão parcial aleatória na camada inferior; (2) seleciona quem manter nas visões parciais das camadas superiores.



Nota

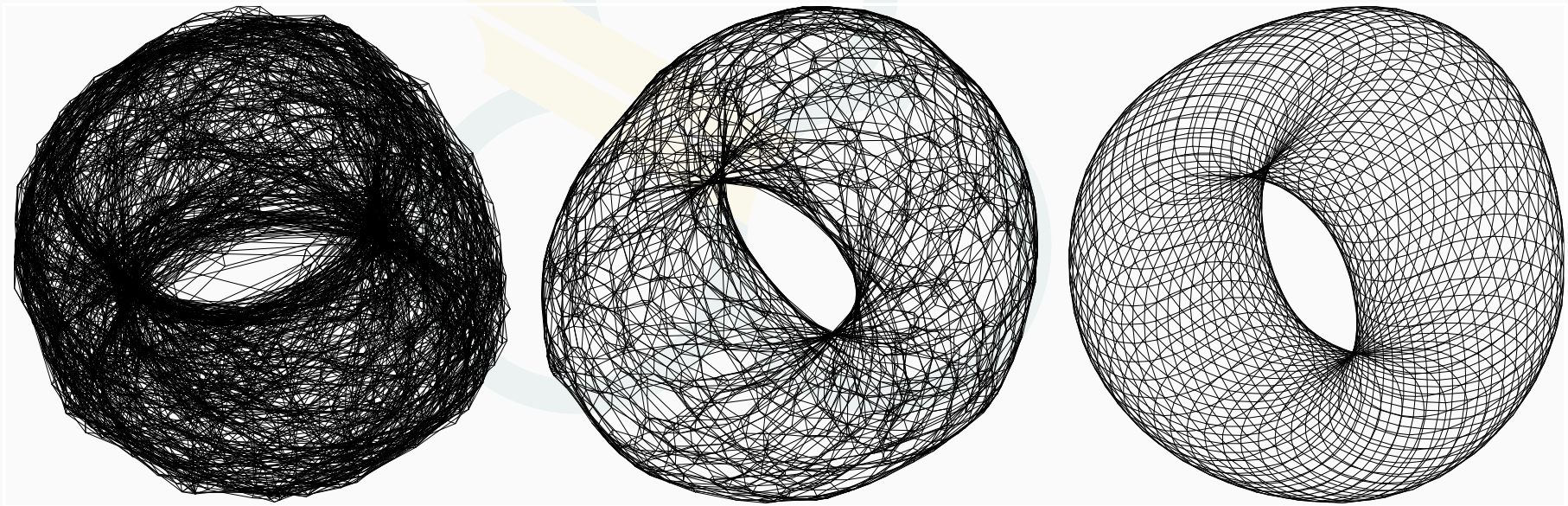
As camadas inferiores alimentam as camadas superiores com nós escolhidos aleatoriamente; a camada superior é seletiva para manter as referências.

Construindo um toro

Considere uma grade $N \times N$. Mantenha referências apenas aos vizinhos mais próximos:

$$\| (a_1, a_2) - (b_1, b_2) \| = d_1 + d_2$$

$$d_i = \min\{N - |a_i - b_i|, |a_i - b_i|\}$$

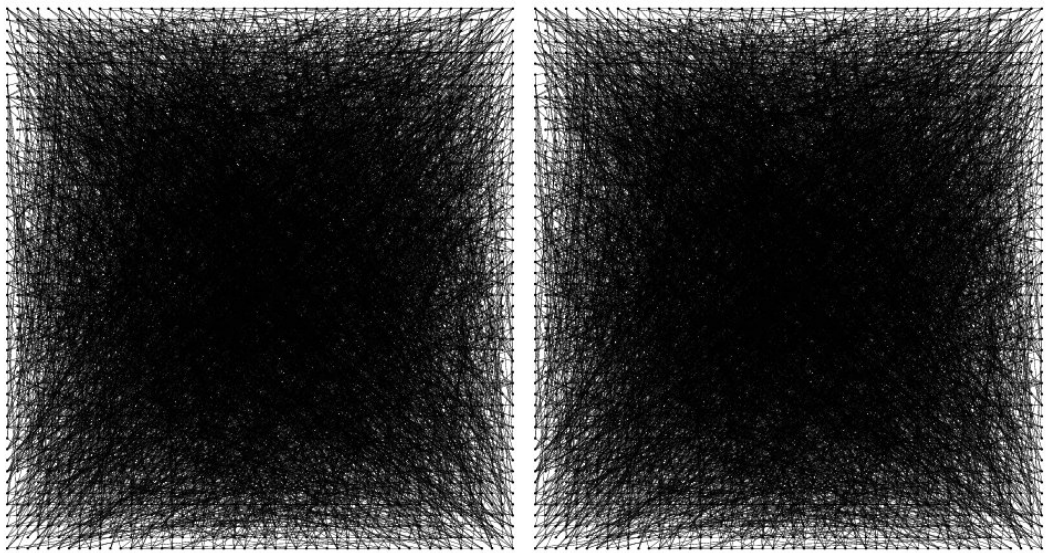


Time

EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

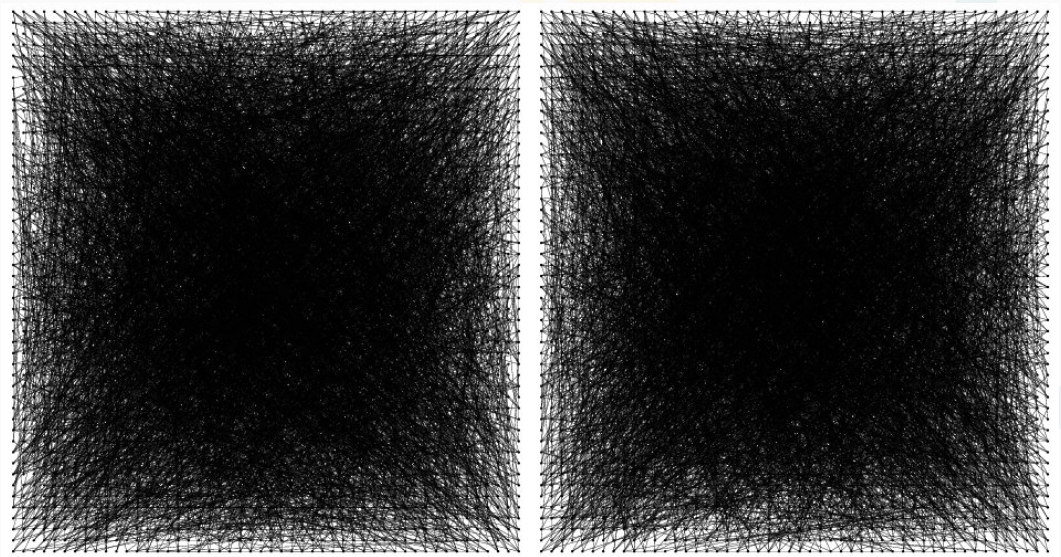
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

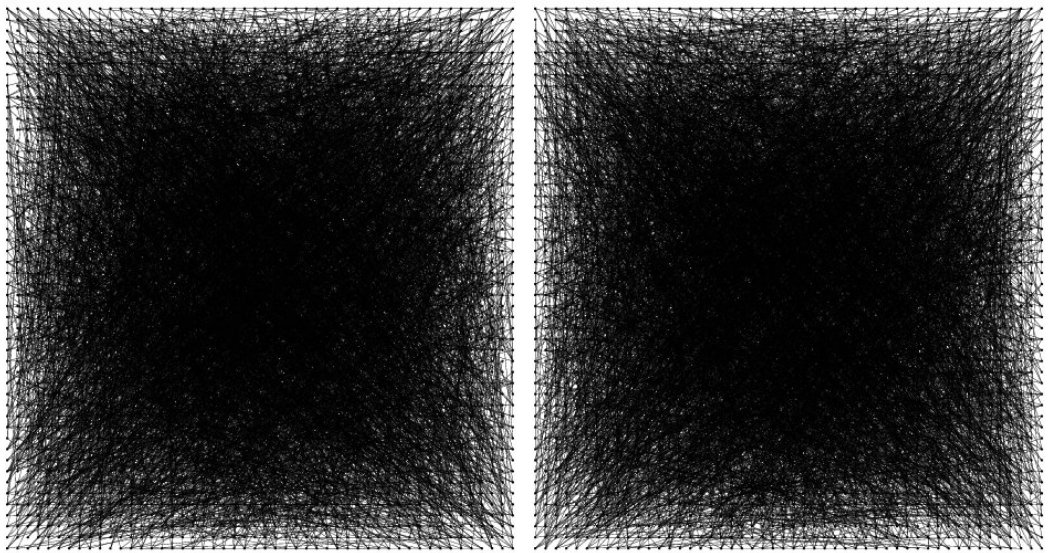
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

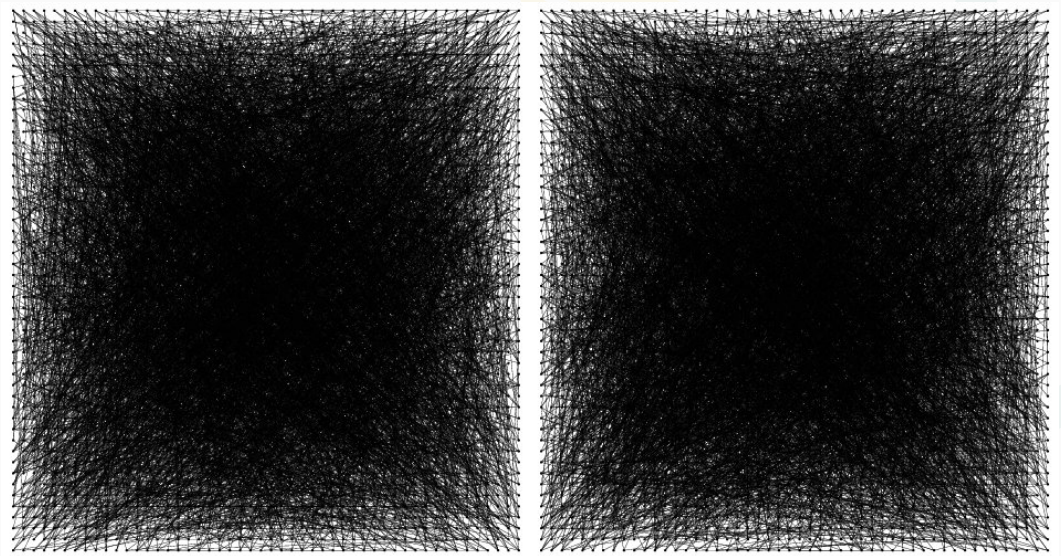
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

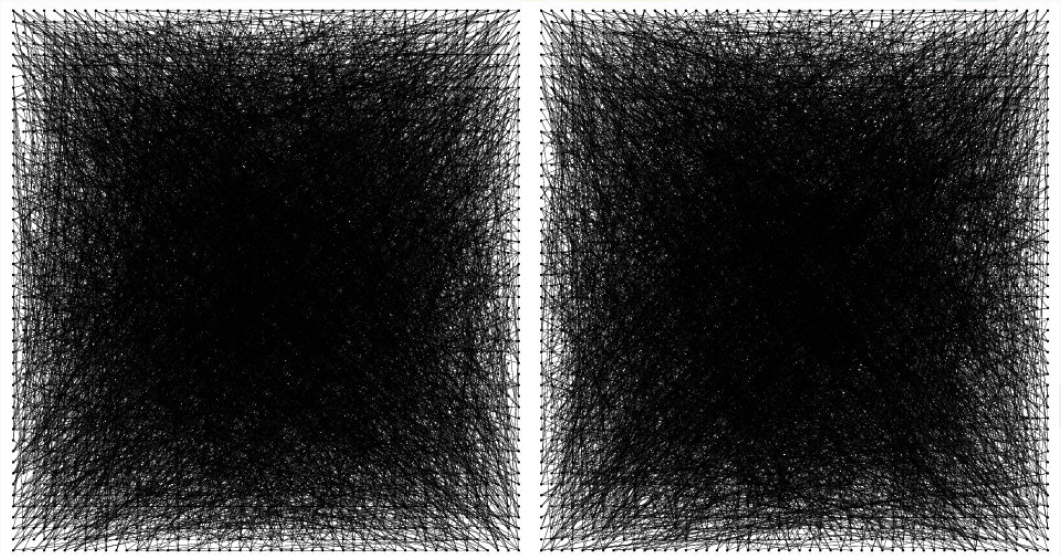
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

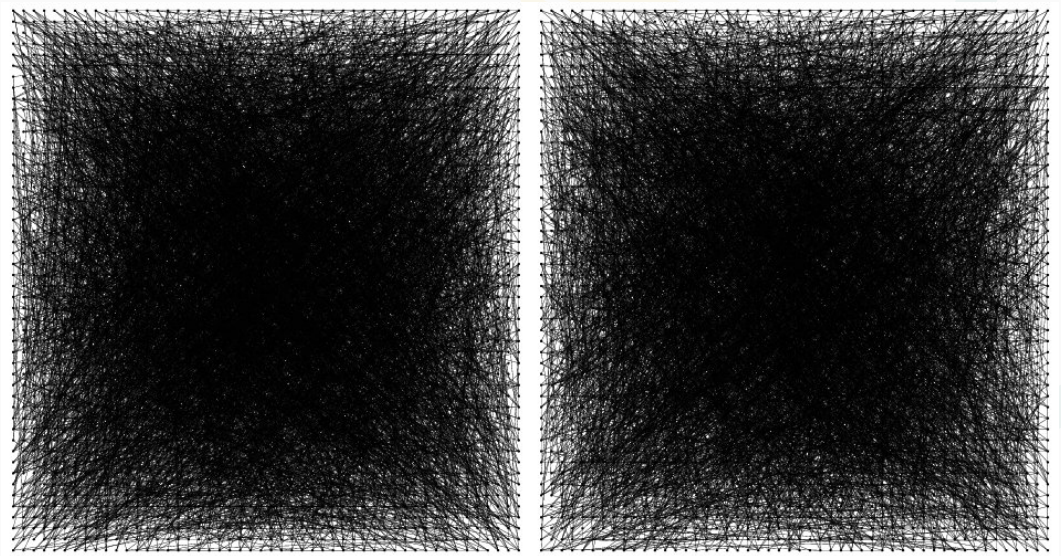
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

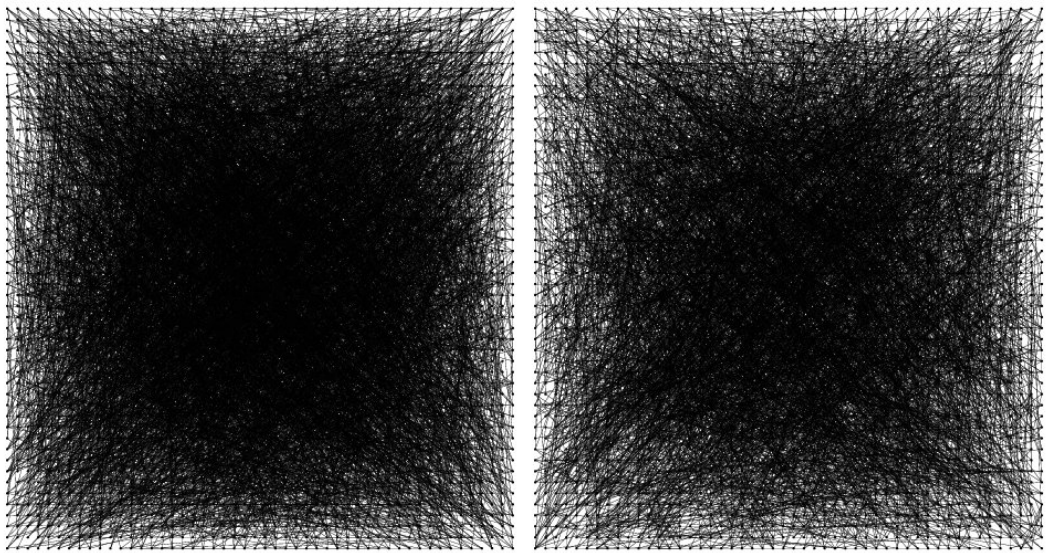
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

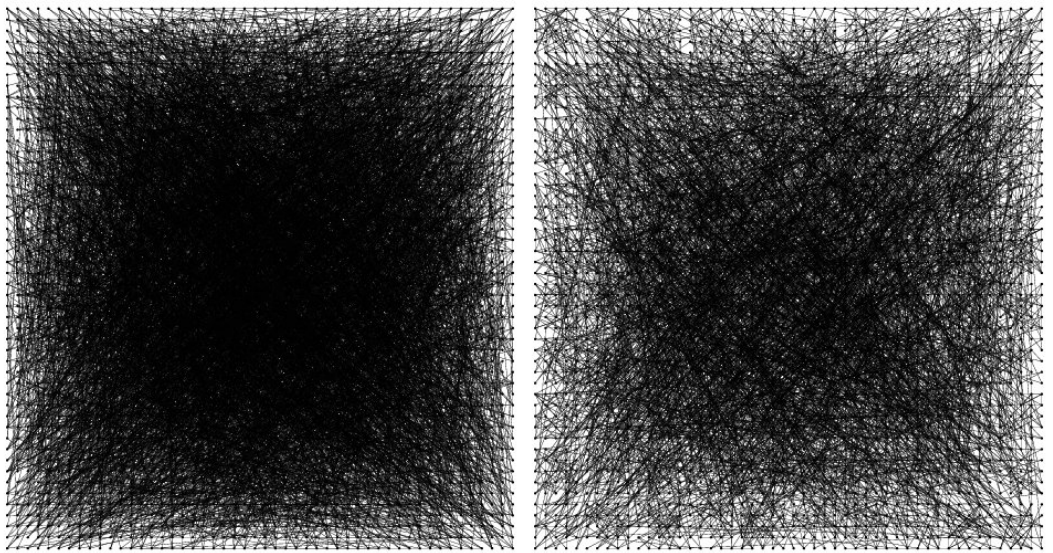
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

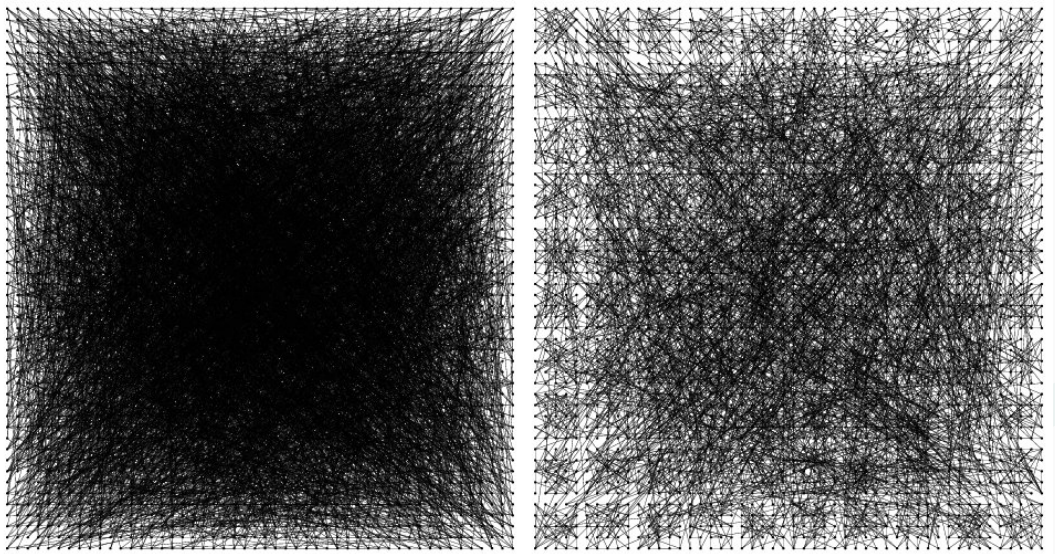
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

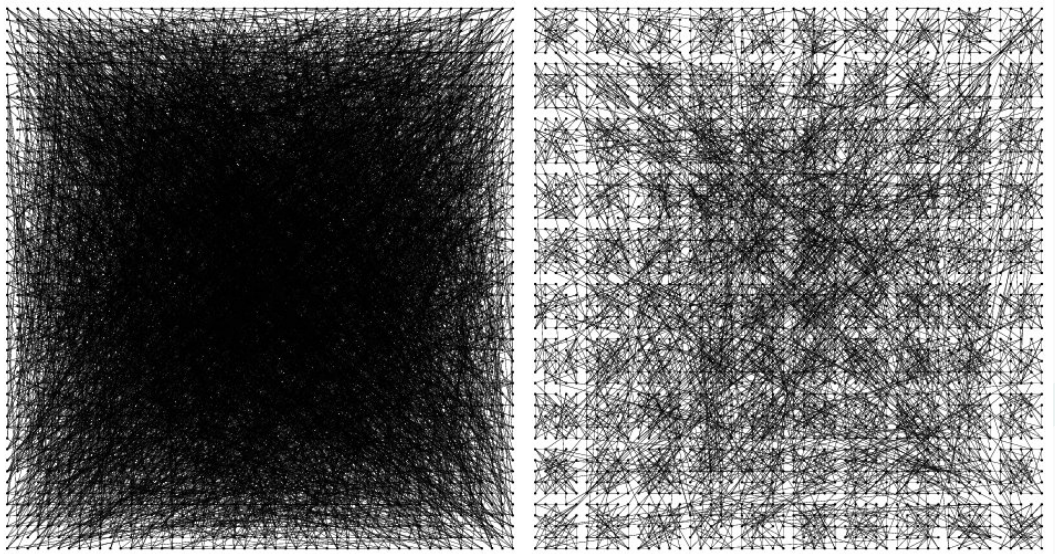
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

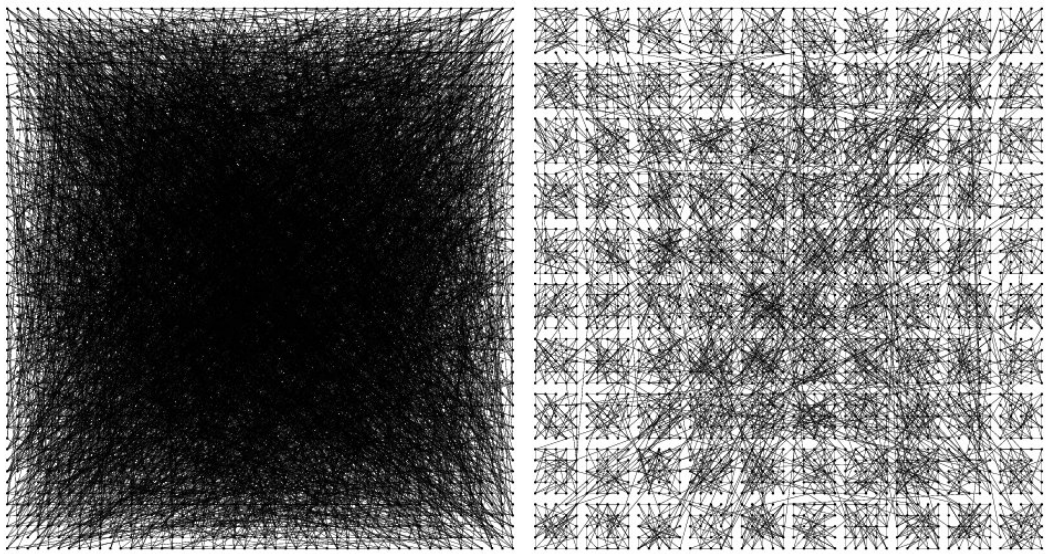
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

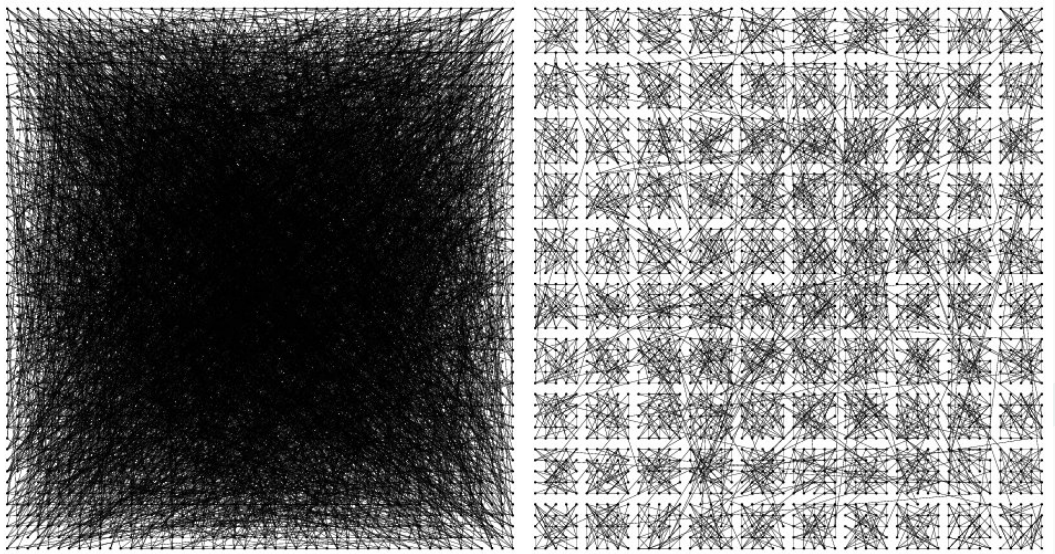
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

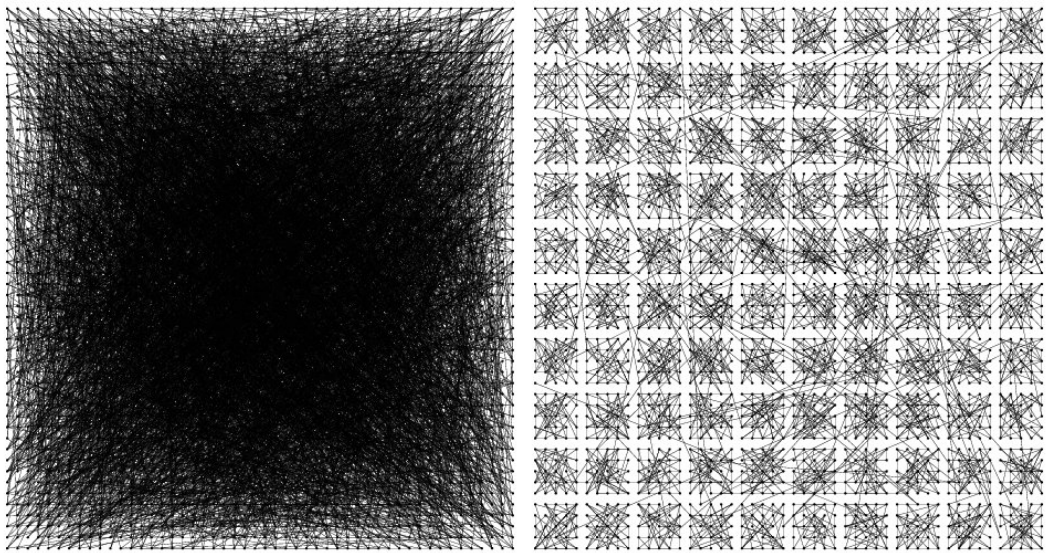
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

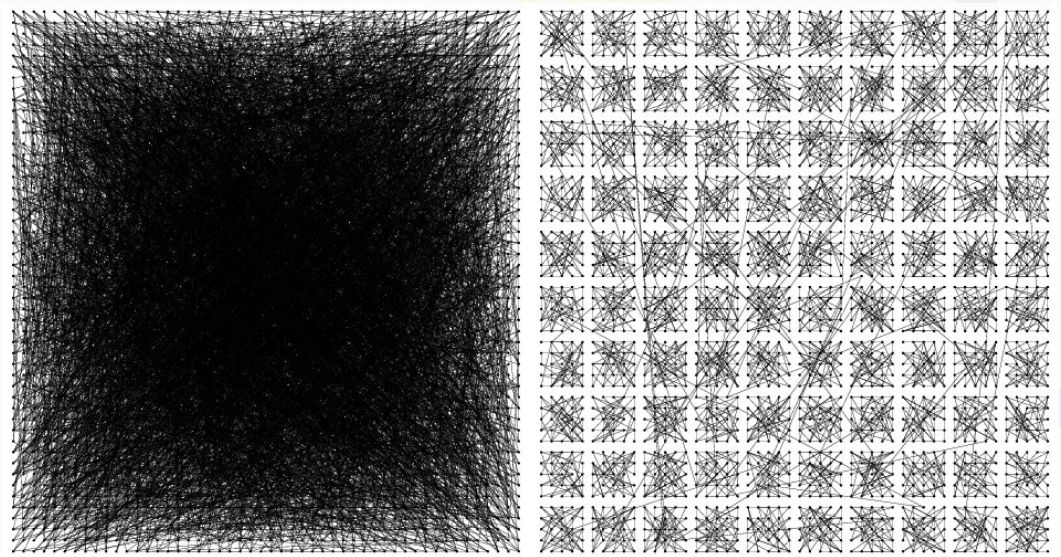
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

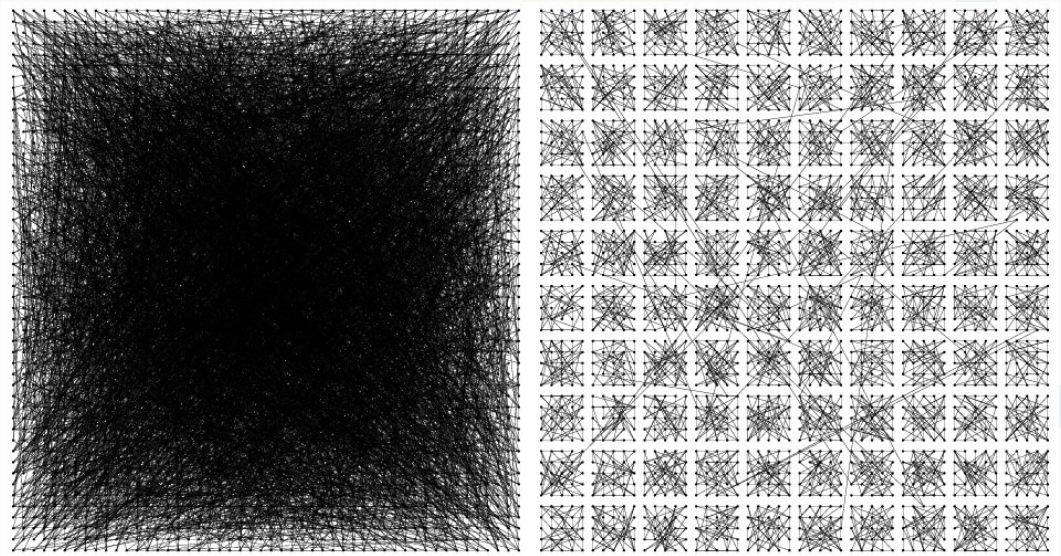
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

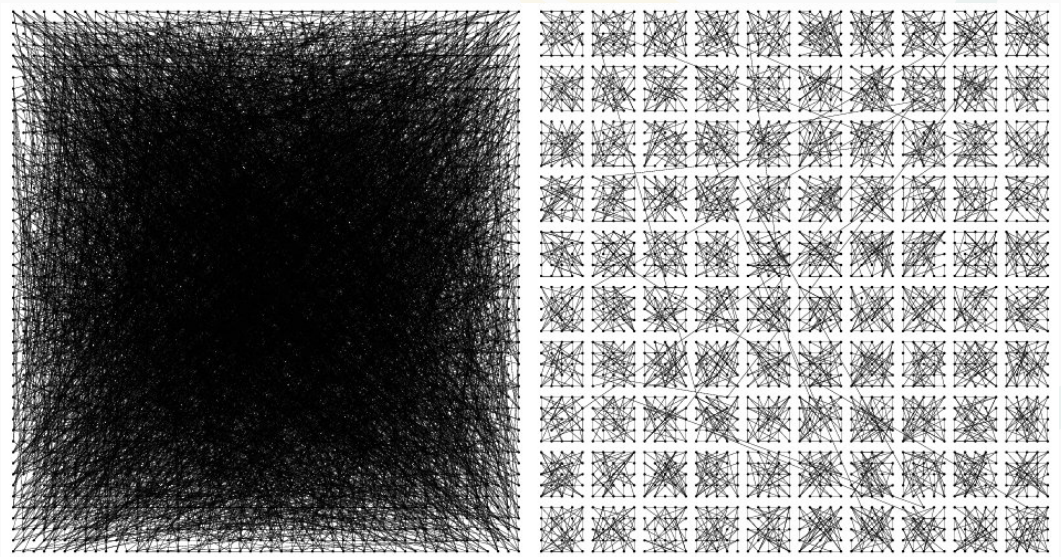
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

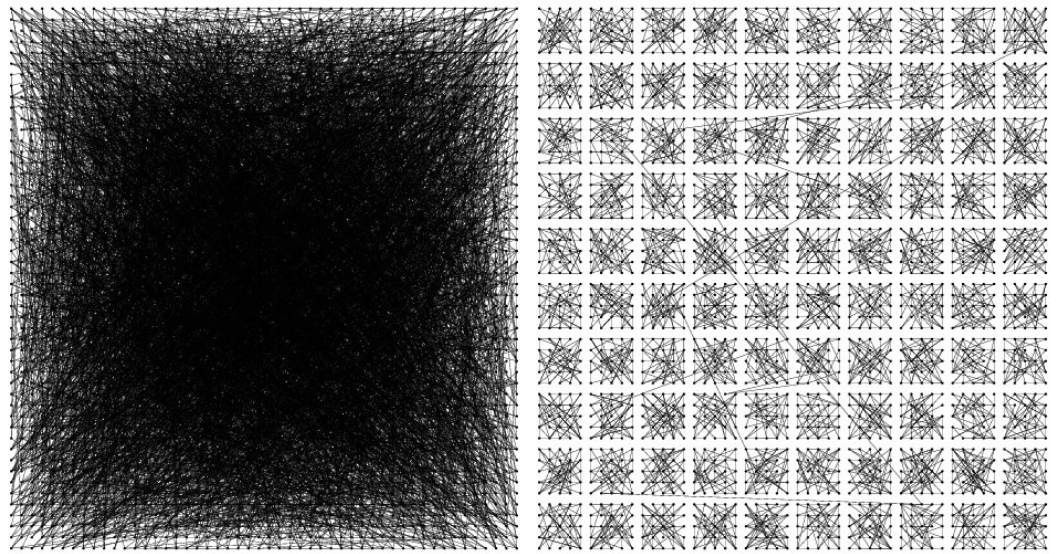
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

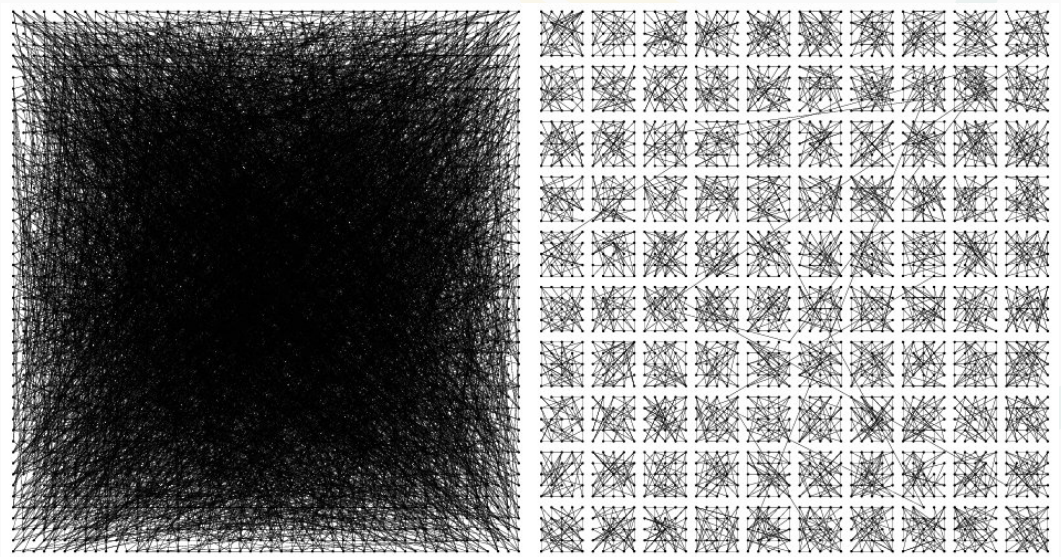
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

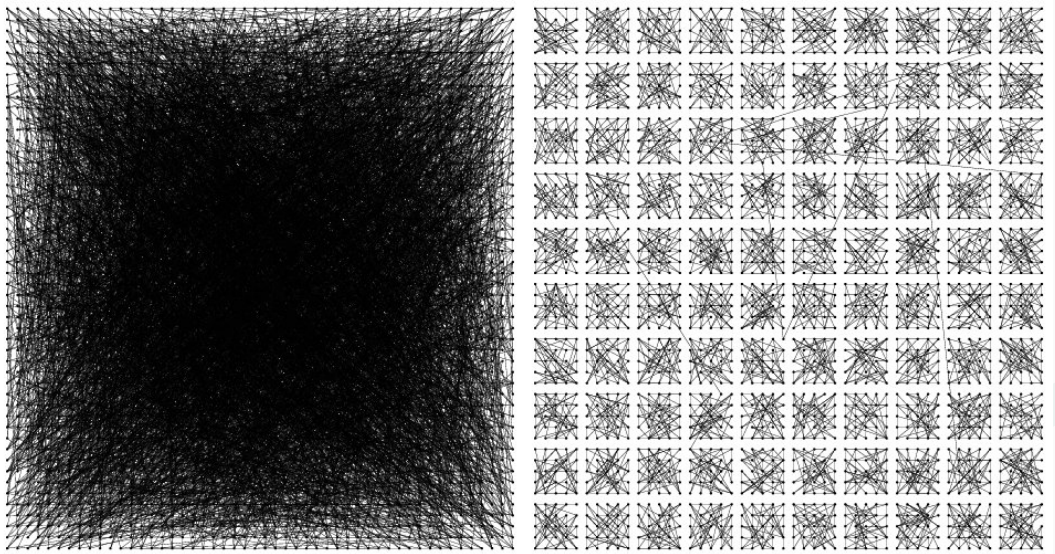
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

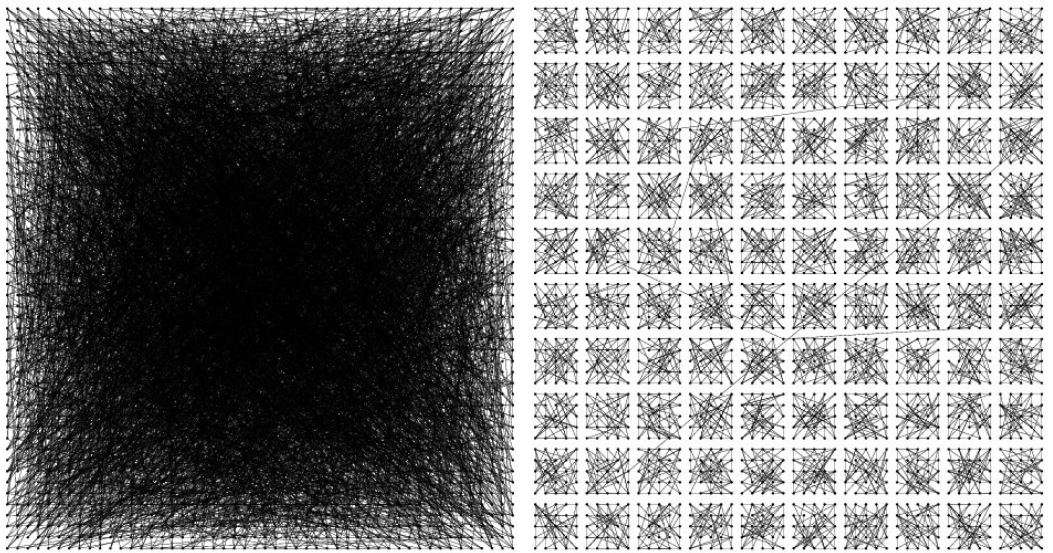
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

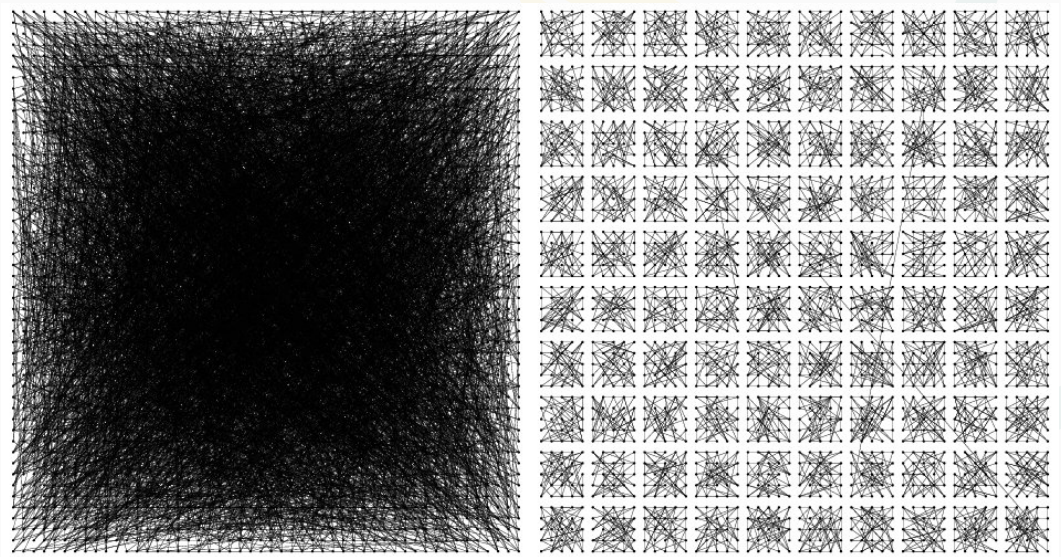
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

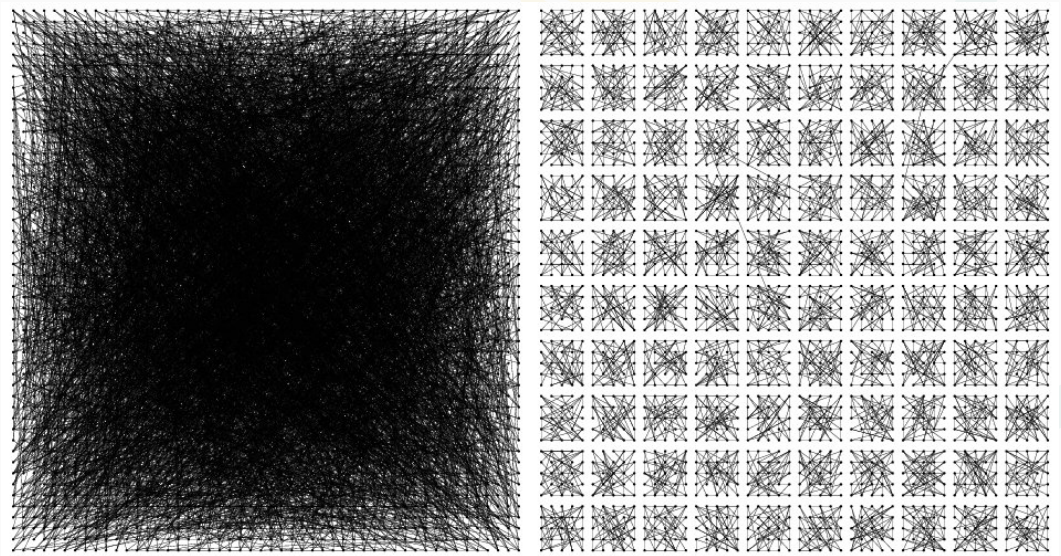
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

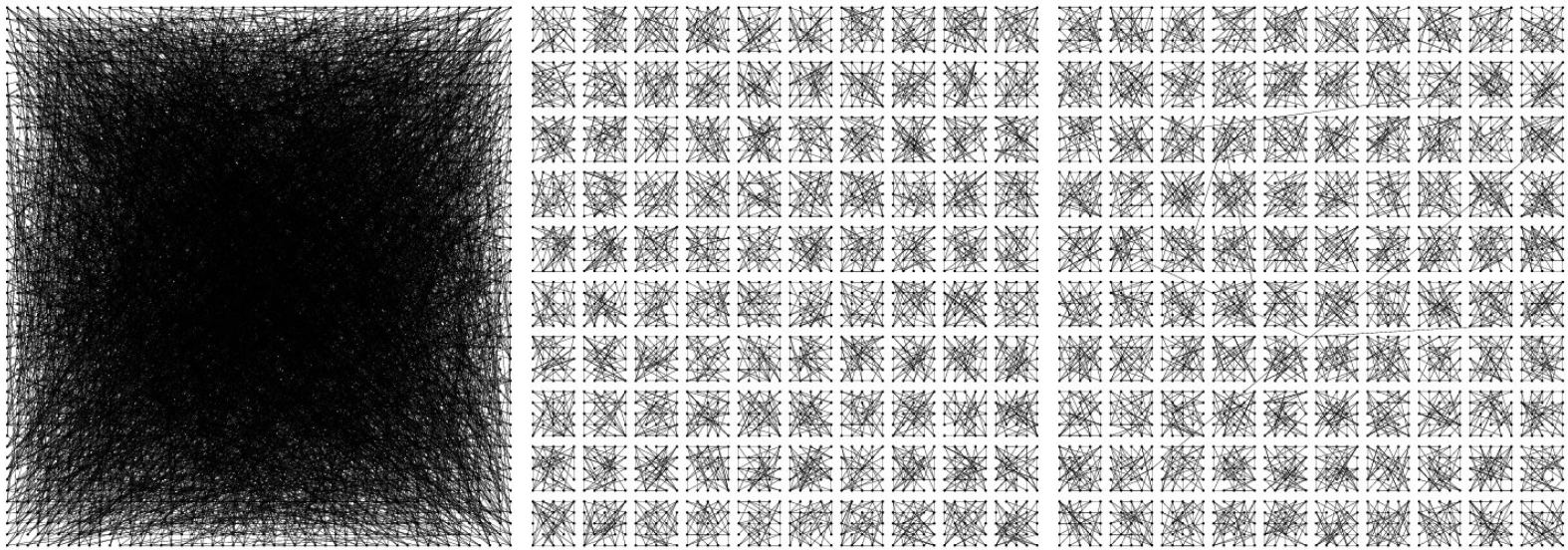
$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



EXEMPLO: CRIANDO CLUSTERS DE NÓS

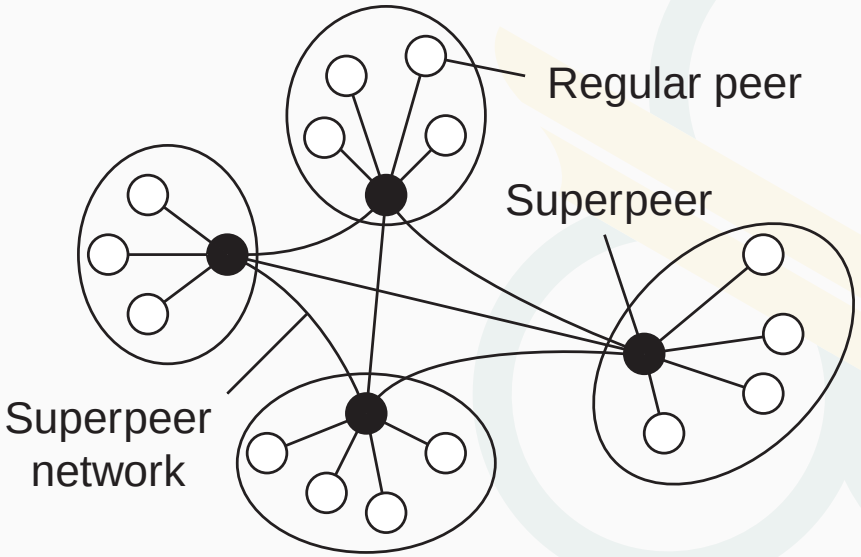
Ideia básica: a todo nó i é definido um *identificador de grupo* $GID(i) \in \mathbb{N}$. O objetivo é particionar o overlay em componentes disjuntos (clusters) tais que:

$$dist(i,j) = \begin{cases} 1 & \text{se } i \text{ e } j \text{ pertencem ao mesmo grupo } [GID(i) = GID(j)] \\ 0 & \text{caso contrário} \end{cases}$$



Observação

Às vezes, selecionar alguns nós para realizar algum trabalho específico pode ser útil.



Exemplos:

- Peers para manter um índice (para buscas)
- Peers para monitorar o estado da rede
- Peers capazes de configurar conexões

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

A está atrás de um firewall, B está na Internet pública

- A configura uma conexão TCP (para os pacotes de controle) com um superpeer S
- S configura uma conexão TCP (para redirecionar os pacotes de controle) com B
- A chamada real usa pacotes UDP diretamente entre A e B

Princípio de operação do Skype: A quer contactar B

Tanto A quanto B estão na Internet pública

- Uma conexão TCP é estabelecida entre A e B para envio de pacotes de controle
- A chamada real usa pacotes UDP entre as portas negociadas

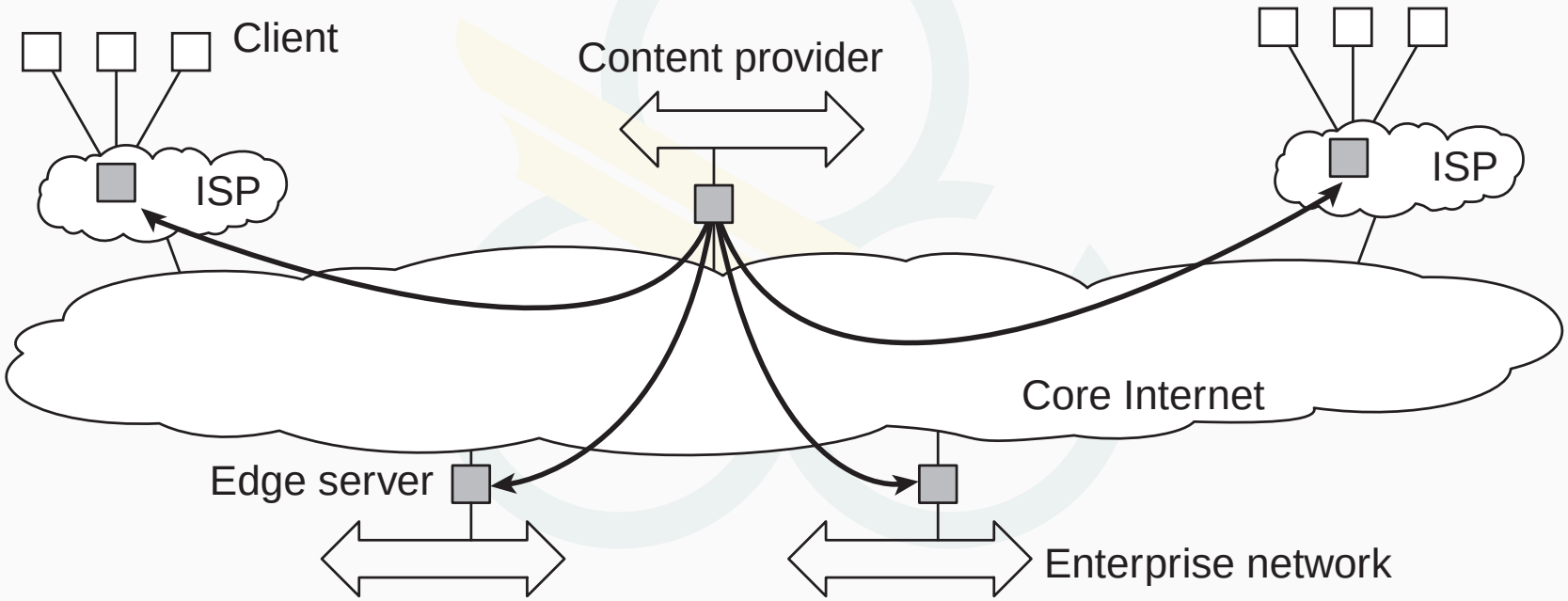
A está atrás de um firewall, B está na Internet pública

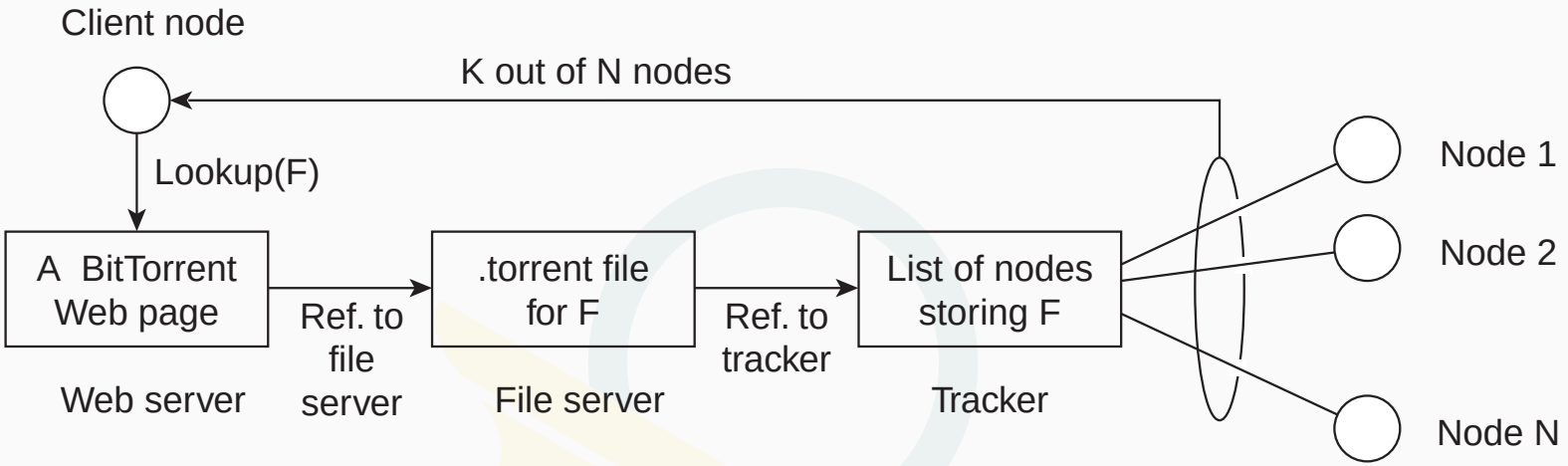
- A configura uma conexão TCP (para os pacotes de controle) com um superpeer S
- S configura uma conexão TCP (para redirecionar os pacotes de controle) com B
- A chamada real usa pacotes UDP diretamente entre A e B

Tanto A quanto B estão atrás de um firewall

- A conecta com um superpeer S via TCP
- S configura uma conexão TCP com B
- Para a chamada real, outro superpeer é usado para funcionar como retransmissor (relay): A (e B) configura a conexão com R
- A chamada é encaminhada usando duas conexões TCP, usando R como intermediário

Exemplo:
Arquiteturas de servidores de borda (*edge-server*), utilizados com frequência como Content Delivery Networks (redes de distribuição de conteúdo).





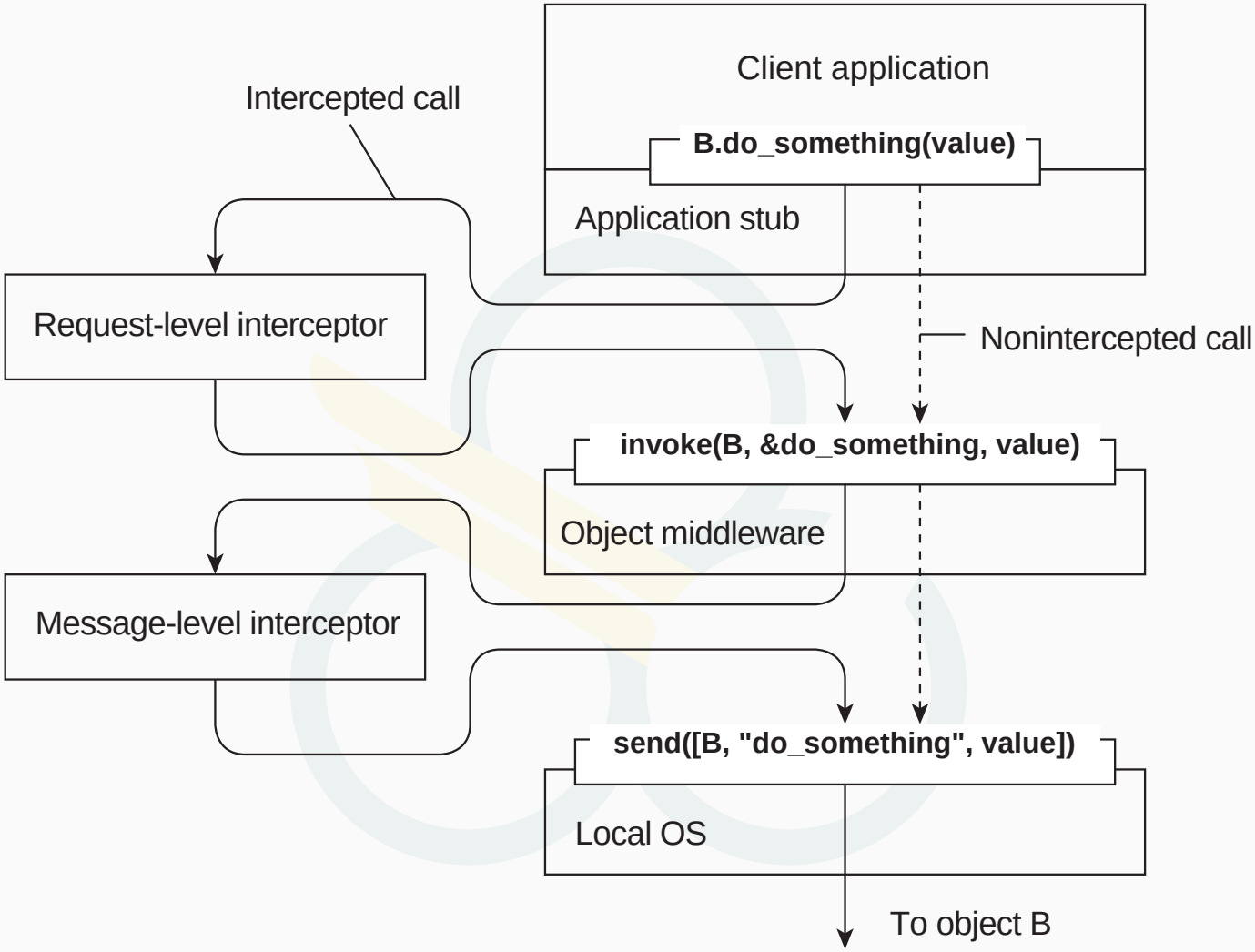
Ideia básica

Assim que um nó identifica de onde o arquivo será baixado, ele se junta a uma swarm (multidão) de pessoas que, em paralelo, receberão pedaços do arquivo da fonte e redistribuirão esses pedaços entre os outros.

Problema
Em muitos casos, arquiteturas/sistemas distribuídos são desenvolvidos de acordo com um estilo arquitetural específico. O estilo escolhido pode não ser o melhor em todos os casos $\Rightarrow$ é necessário adaptar o comportamento do middleware (dinamicamente).

Interceptors
Interceptam o fluxo de controle normal quando um objeto remoto for invocado.

INTERCEPTORS



- Separação de interesses: tente separar as funcionalidades extras e depois costurá-las em uma única implementação ⇒ aplicabilidade restrita (*toy examples*)
- Reflexão computacional: deixe o programa inspecionar-se em tempo de execução e adaptar/mudar suas configurações dinamicamente, se necessário ⇒ ocorre principalmente no nível da linguagem, aplicabilidade não é muito clara.
- Projeto baseado em componentes: organize uma aplicação distribuída em componentes que podem ser substituídos dinamicamente quando necessário ⇒ causa muitas e complexas interdependências entre componentes.

Observação

A distinção entre arquiteturas de sistemas e arquiteturas de software fica confusa quando adaptação automática deve ser considerada:

- Autoconfiguração
- Autogerenciamento
- Autocura
- Auto-otimização
- Auto-*

Governador centrífugo

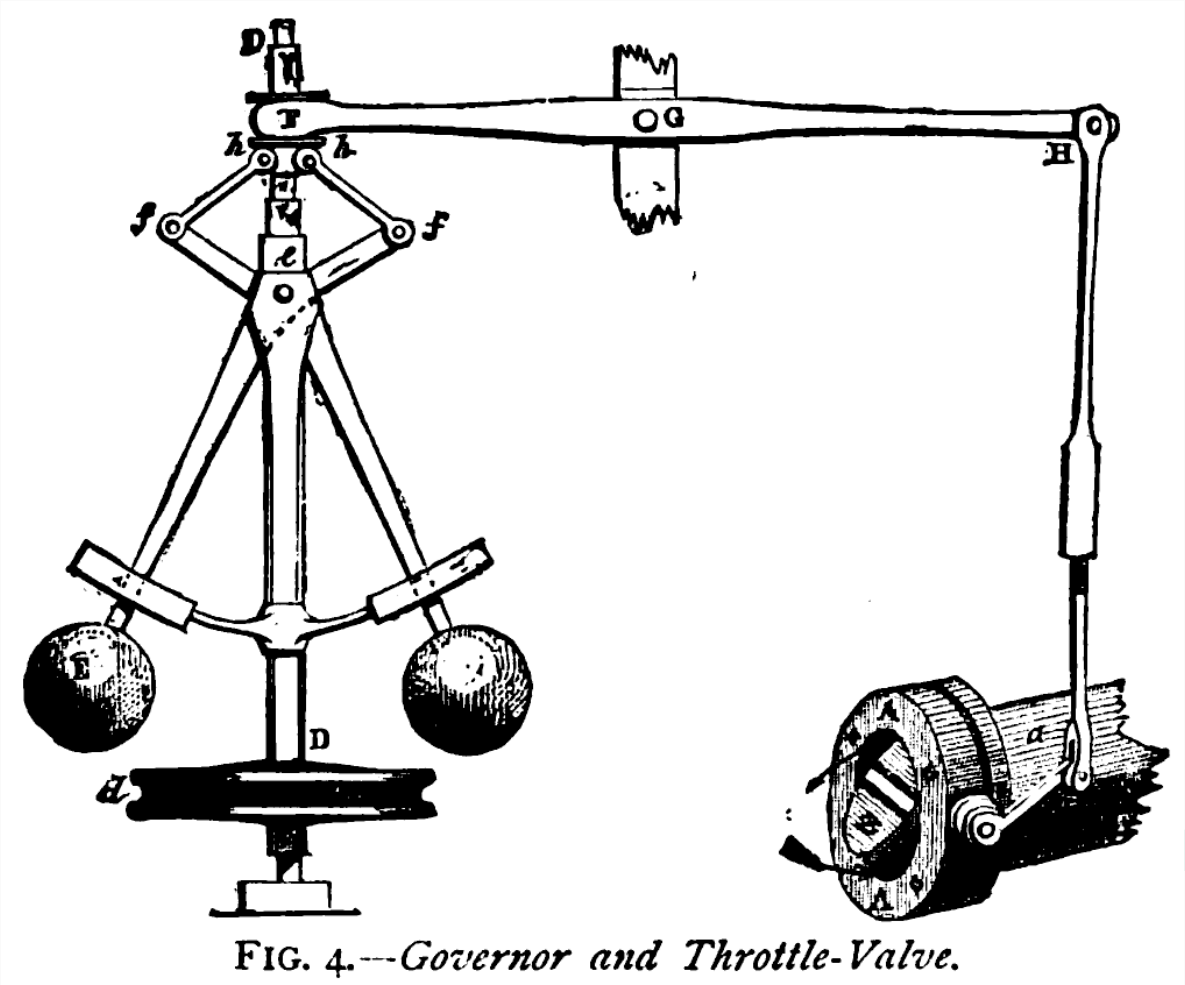


FIG. 4.—*Governor and Throttle-Valve.*

Governador centrífugo

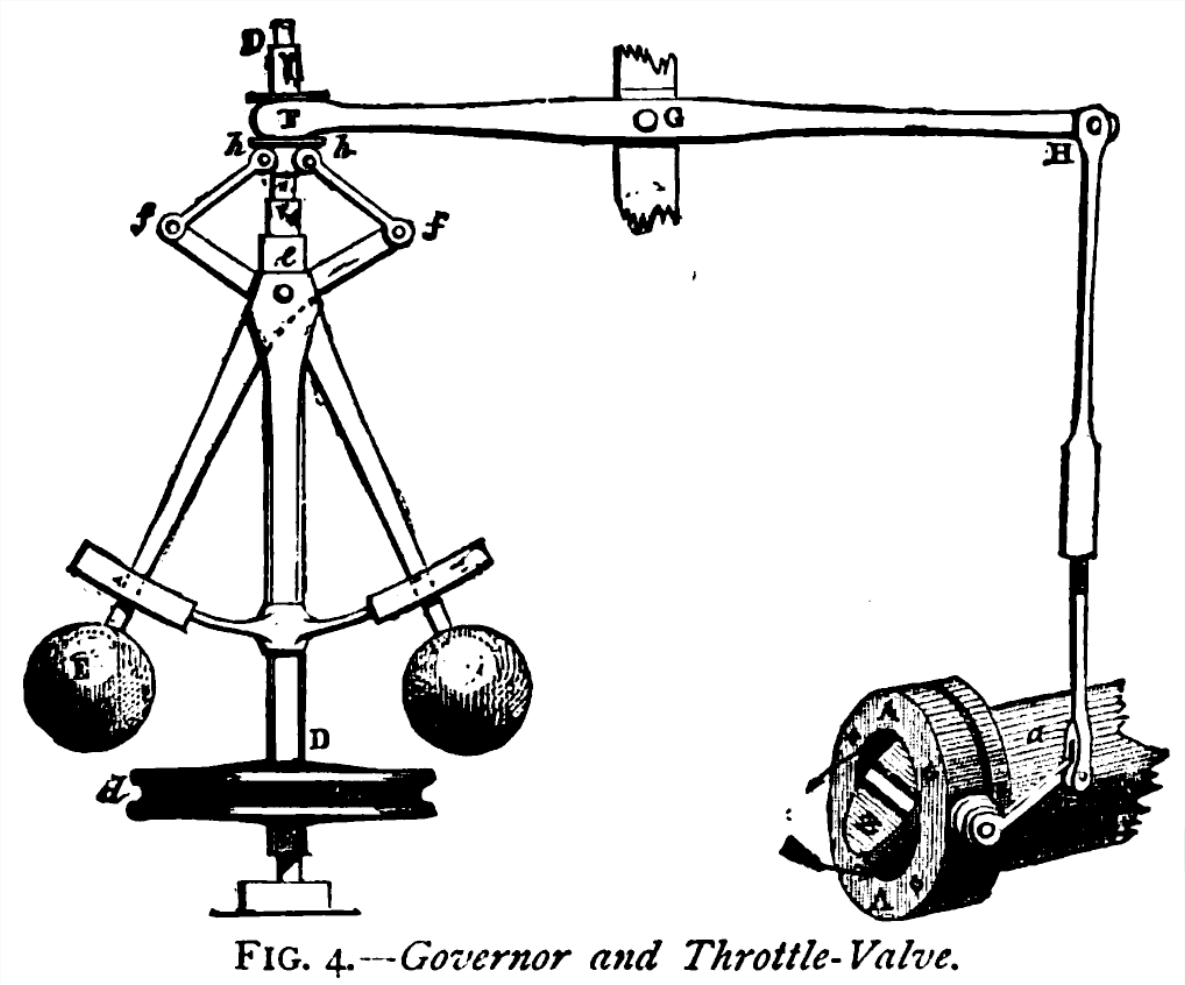


FIG. 4.—*Governor and Throttle-Valve.*

- Criado em **1788** por James Watt.

Governador centrífugo

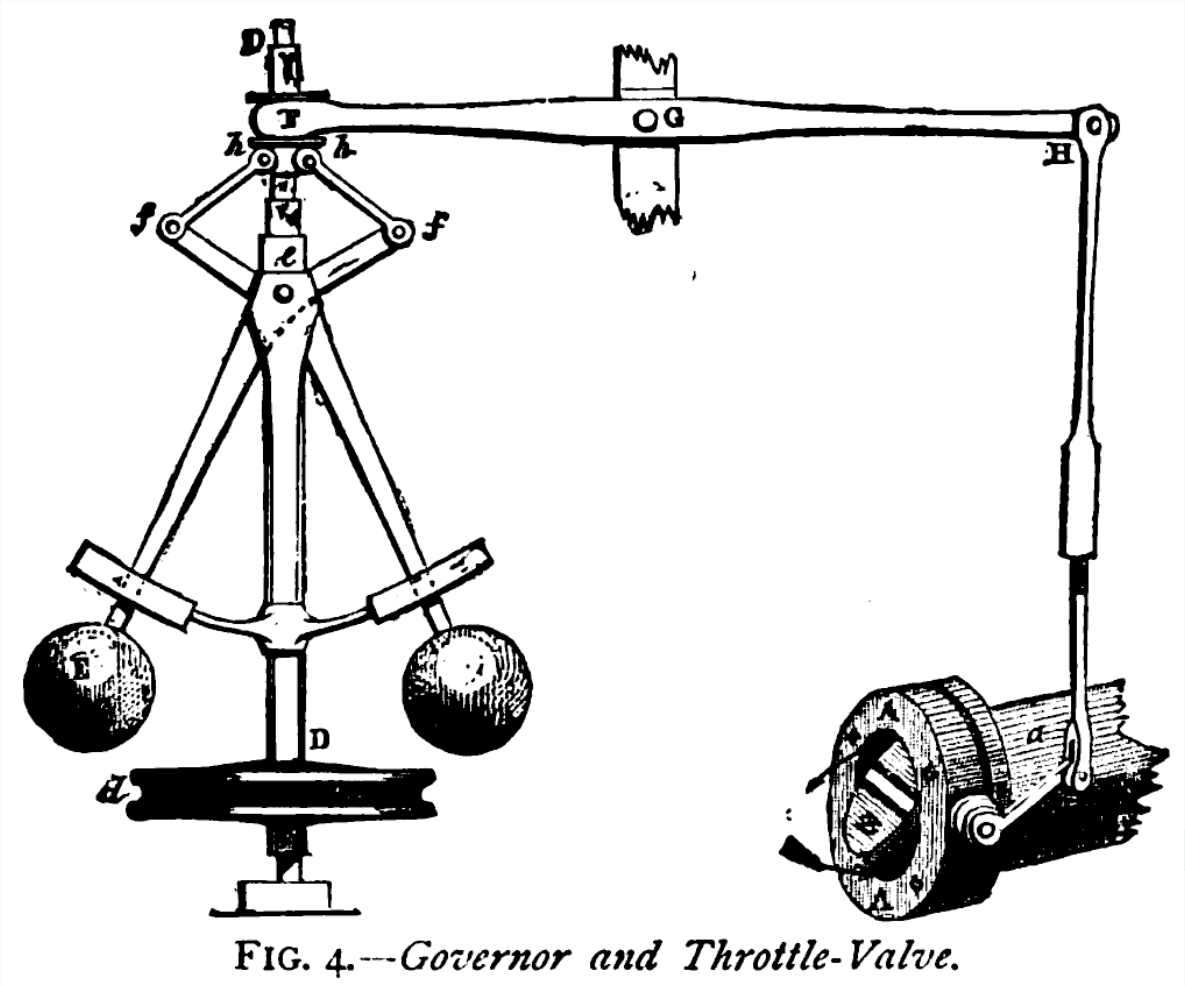


FIG. 4.—*Governor and Throttle-Valve.*

- Criado em **1788** por James Watt.
- Controla a admissão de vapor no cilindro de máquinas a vapor.

MODELO DE REGULAÇÃO POR FEEDBACK

Em muitos casos, sistemas auto-* são organizados como um sistema de regulação por feedback