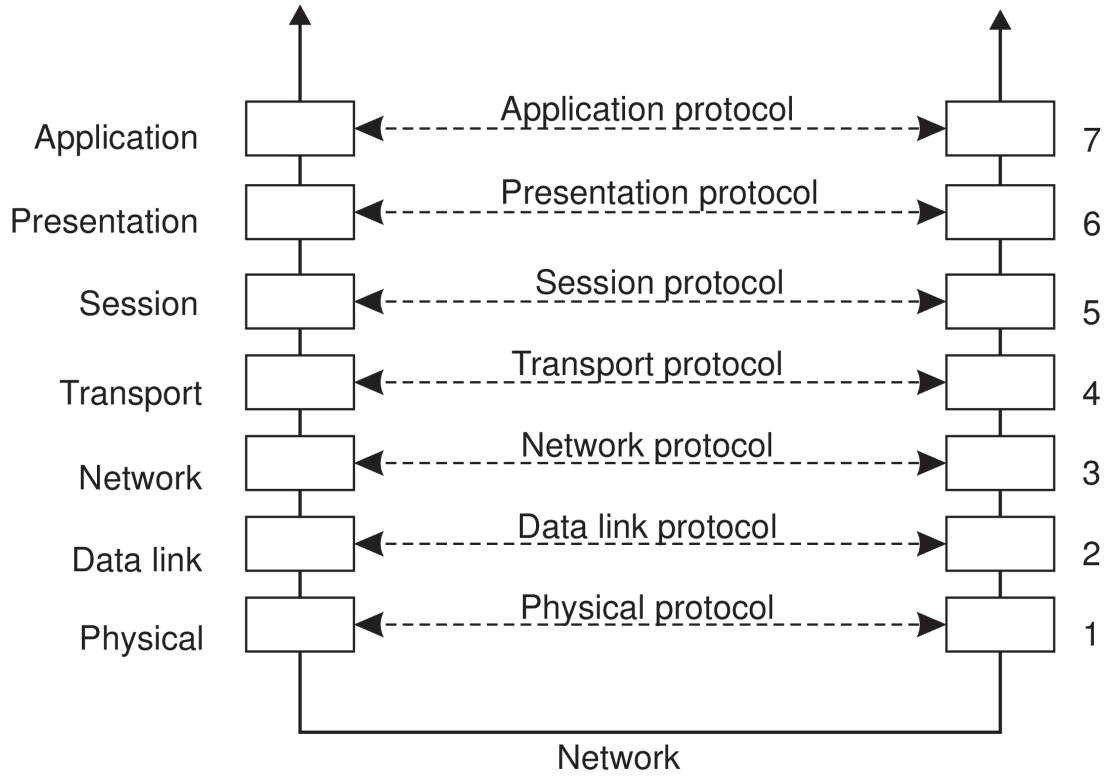
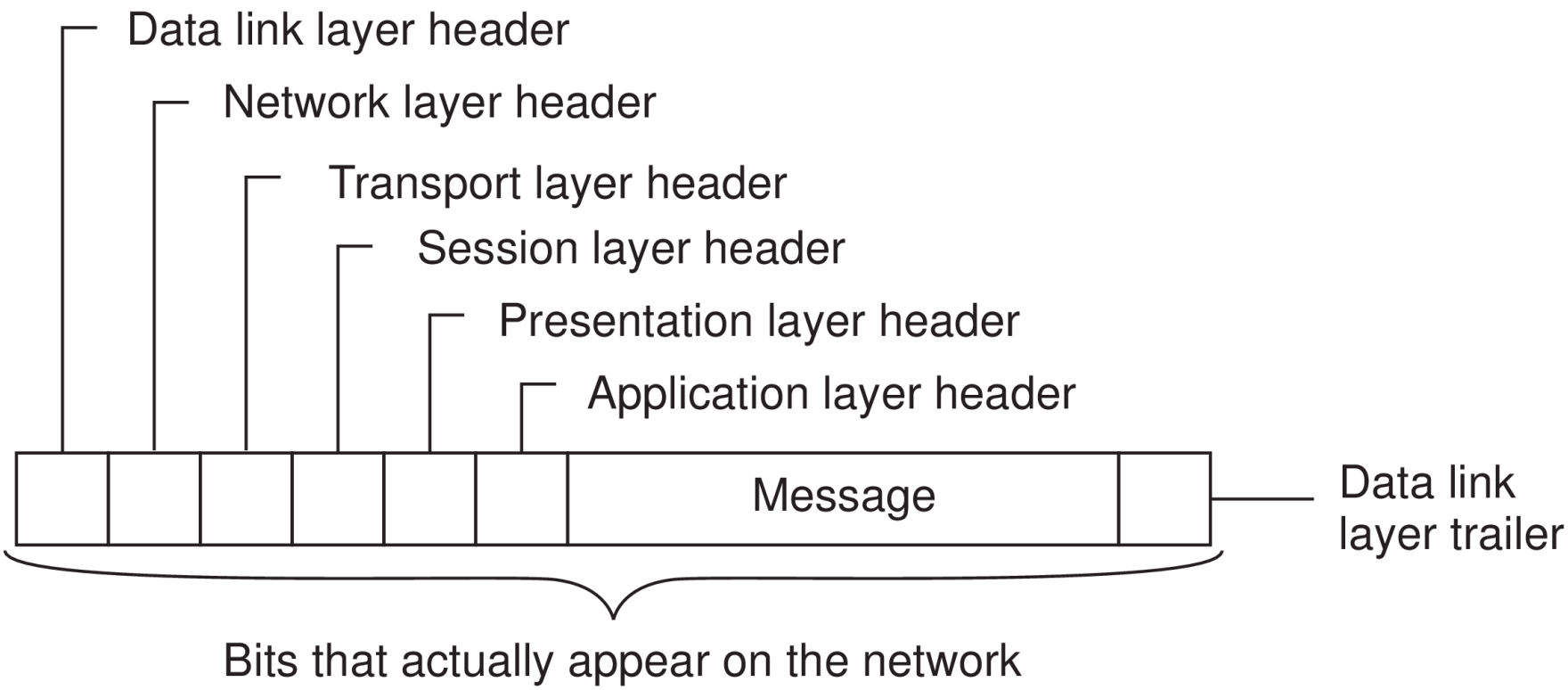
Comunicações

Pilha OSI


Application
Presentation
Session
Transport
Network
Data link
Physical
Application protocol
Presentation protocol
Session protocol
Transport protocol
Network protocol
Data link protocol
Physical protocol
7
6
5
4
3
2
1
Network

Pacote de dados



Camadas de baixo nível

- Camada Física:
  - codificação de bits
- Camada de Ligação:
  - codificação de uma serie de bits num frame com controlo de erros e fluxo
- Camada de Rede:
  - como pacotes são **roteados** numa rede de computadores

Camada de Transporte

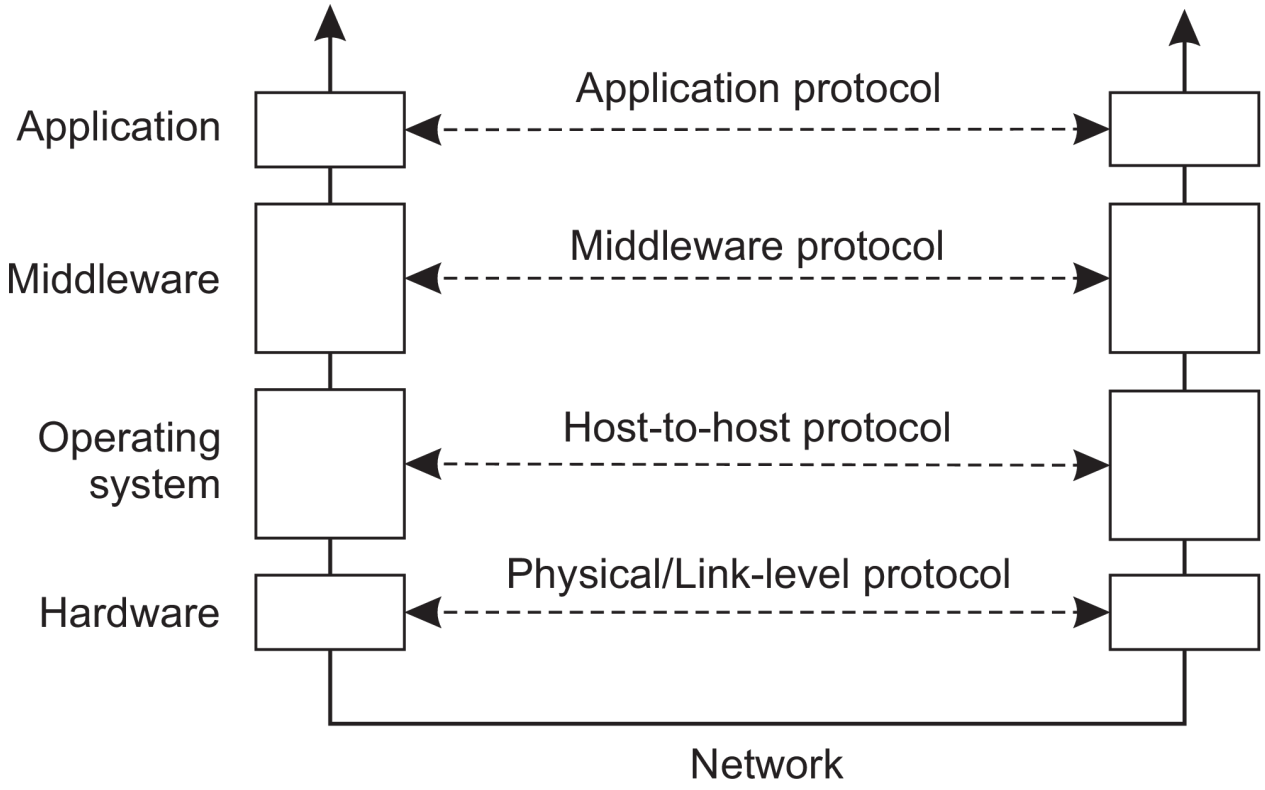
- Camada de suporte à comunicação da maioria dos sistemas distribuídos

- Protocolos principais:
  - **TCP**: orientado à ligação, fiável, orientado aos fluxos de informação (*stream*)
  - **UDP**: comunicação não fiável (*best-effort*)

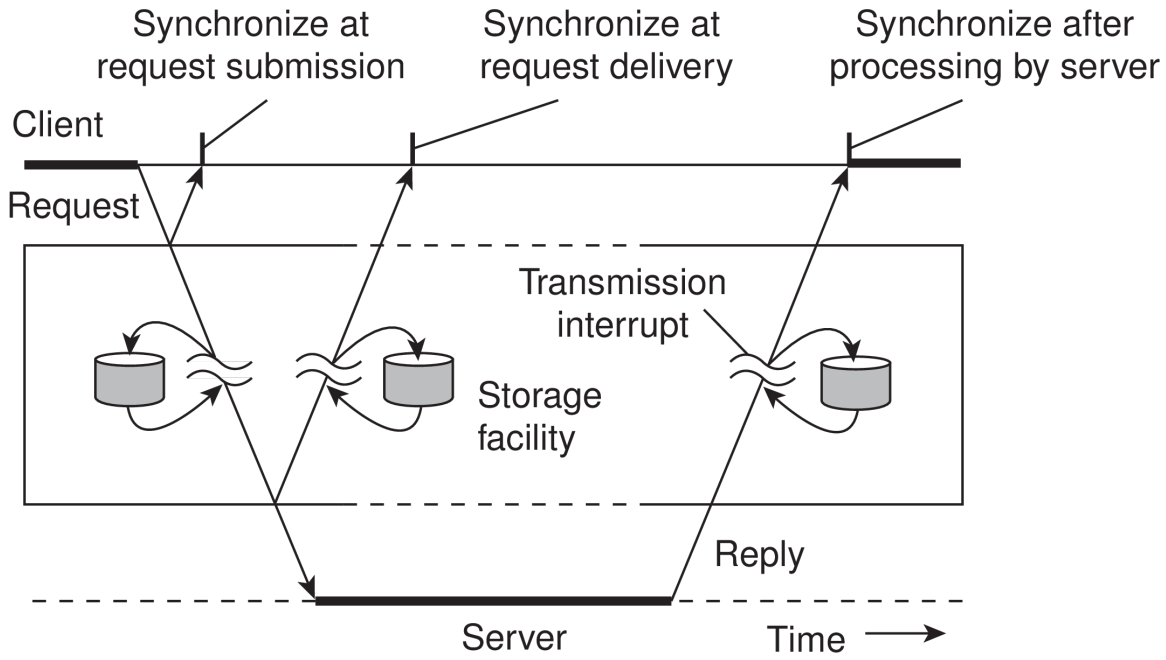
Camada Middleware

- Middleware permite partilha de serviços e protocolos comuns entre diversas aplicações
  - Conjunto rico de protocolos de comunicação
  - *(Un)marshaling* dos dados, necessário à integração de sistemas
  - Protocolos de endereçamento, para partilha fácil de recursos
  - Protocolos de cifragem, para comunicações seguras
  - Mecanismos potenciadores de escalabilidades tais como replicação e *caching*

Esquema de camadas adaptado



Tipos de comunicação



- Transiente vs Persistente
  - Servidor descarta mensagens quando estas não podem ser entregues
  - Servidor guarda mensagens até poderem ser entregues
- Assíncrono vs Síncrono

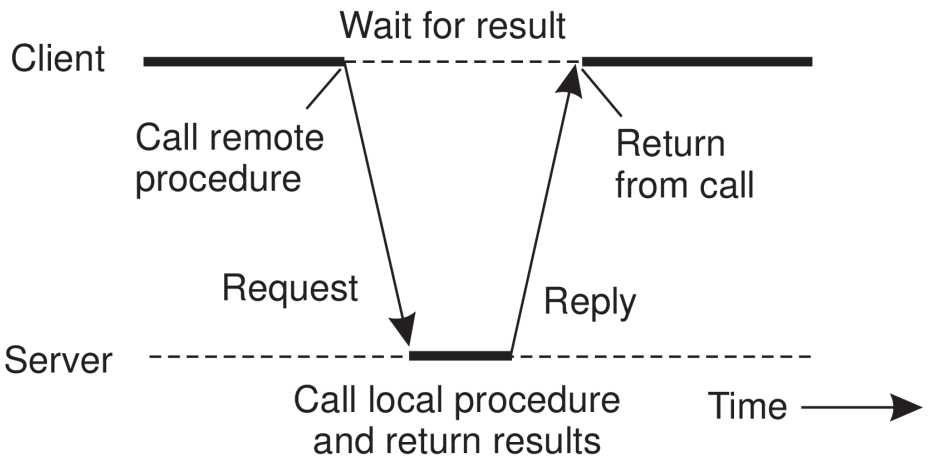
Cliente/Servidor

- Modelo Cliente/Servidor é baseado no modelo de comunicações transiente síncrono.
  - Cliente e Servidor têm que estar activos durante o tempo de comunicação
  - Cliente faz pedido e bloqueia até receber resposta
  - Servidor essencialmente espera por pedidos, e processa os mesmos subsequentemente.
- Problemas:
  - Cliente não pode fazer nada enquanto espera
  - Falhas têm que ser processadas imediatamente (cliente em espera)
  - Modelo pode não se adaptar ao problema (ex. e-mail)

Mensagens

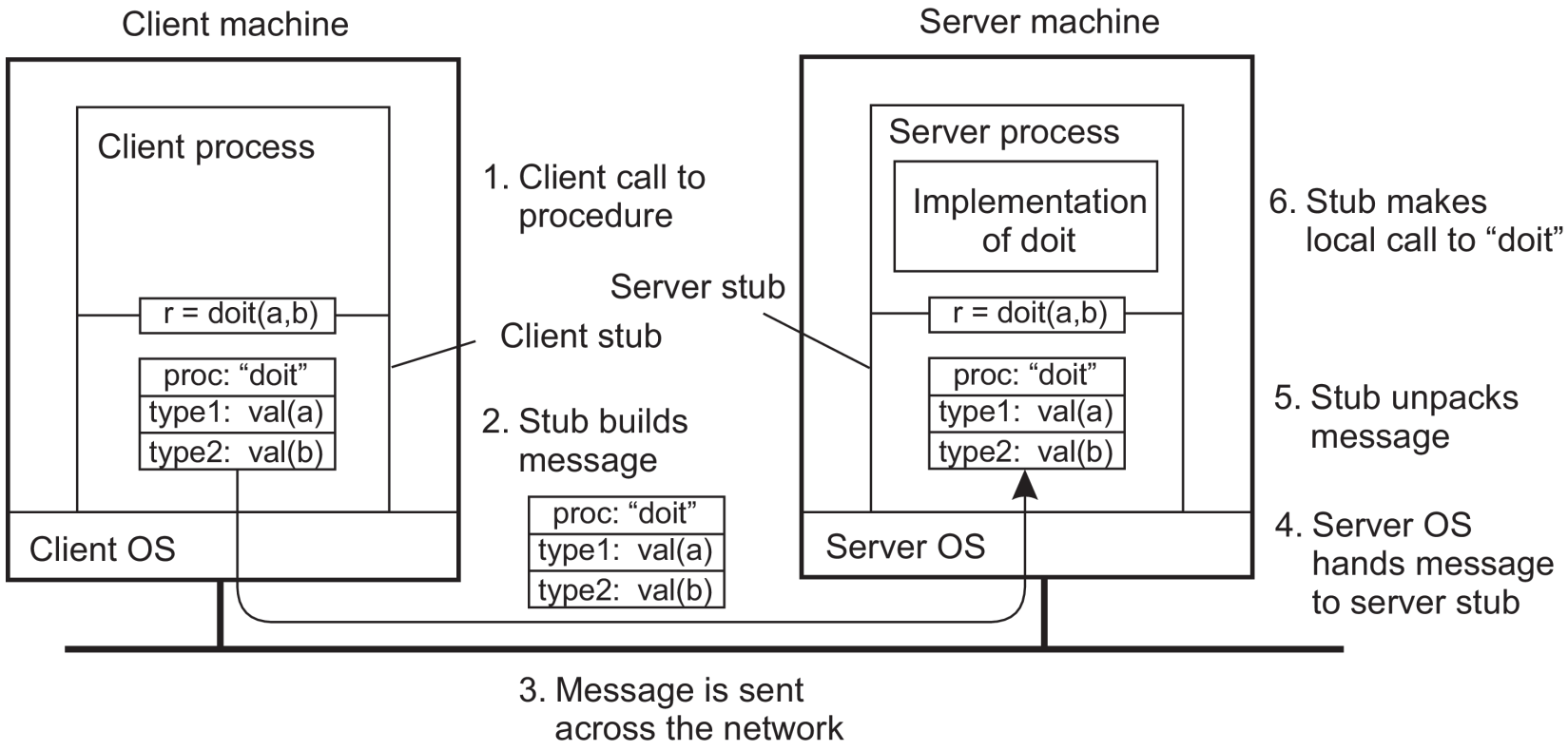
- Middleware orientado a mensagens
- Comunicação Persistente e Assíncrona.
- Processos trocam mensagens, que são colocadas em filas de espera
- Emissor não precisa de esperar por uma resposta imediata
- Tolerância a falhas disponibilizada pelo Middleware.

Remote Procedure Call (RPC)



- Programadores estão familiarizados com modelo procedimental
- Procedimentos quando bem construídos operam de forma isolada
- Não existe nenhuma razão para não executar procedimentos noutra máquina.

Operação de um RPC

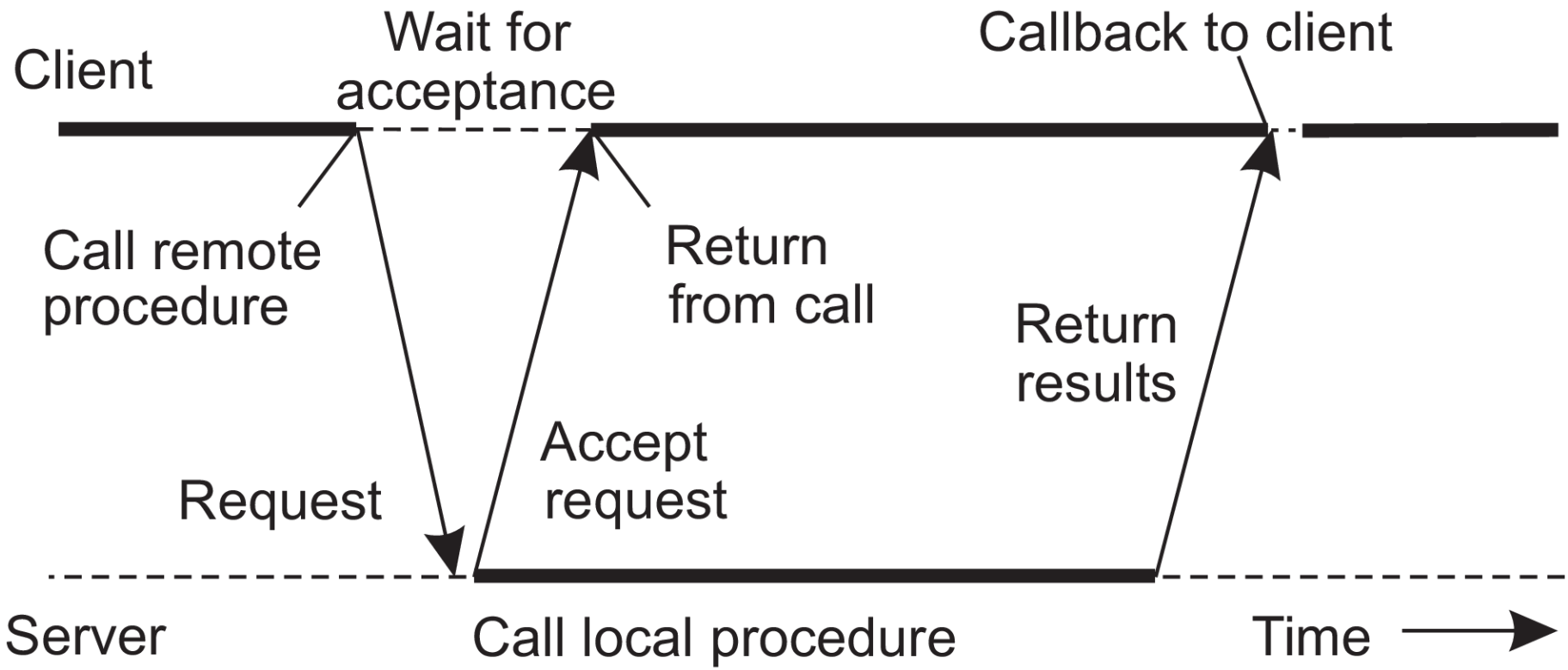


RPC: passagem de parâmetros

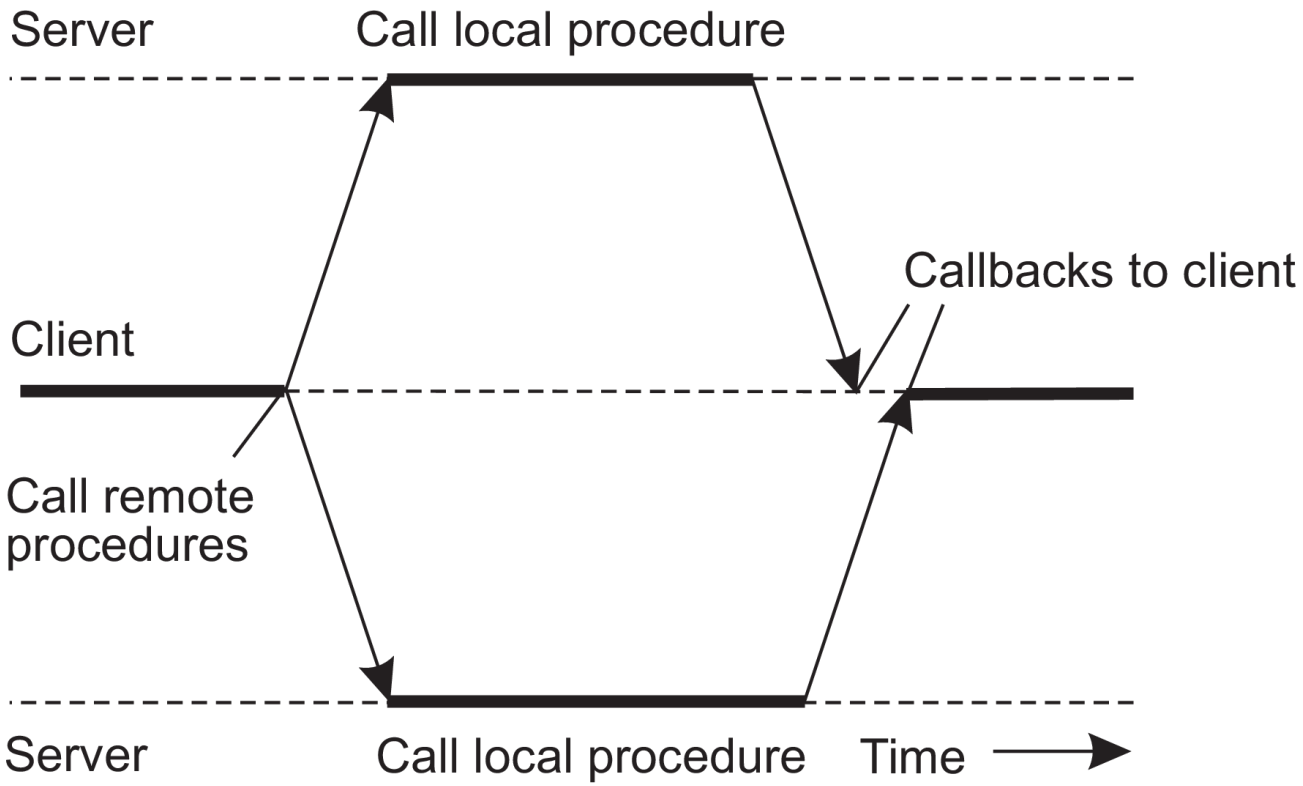
- Cliente e Servidor podem ter representações diferentes dos dados (*little endian vs big endian*)
- Empacotamento de um parâmetro significa transformar um valor numa sequência de bytes
- Cliente e Servidor têm que concordar com a mesma codificação

- Como são representados os valores básicos (*int, float, char*) ?
- Como são representados dados complexos (*arrays, objects*) ?

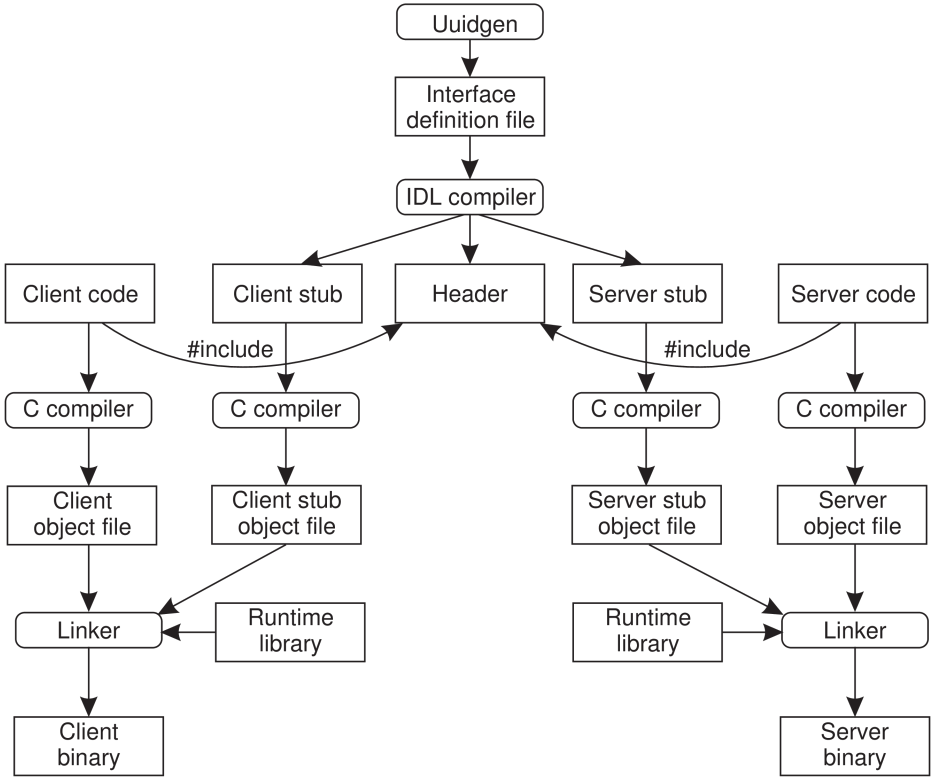
RPC's assíncronos



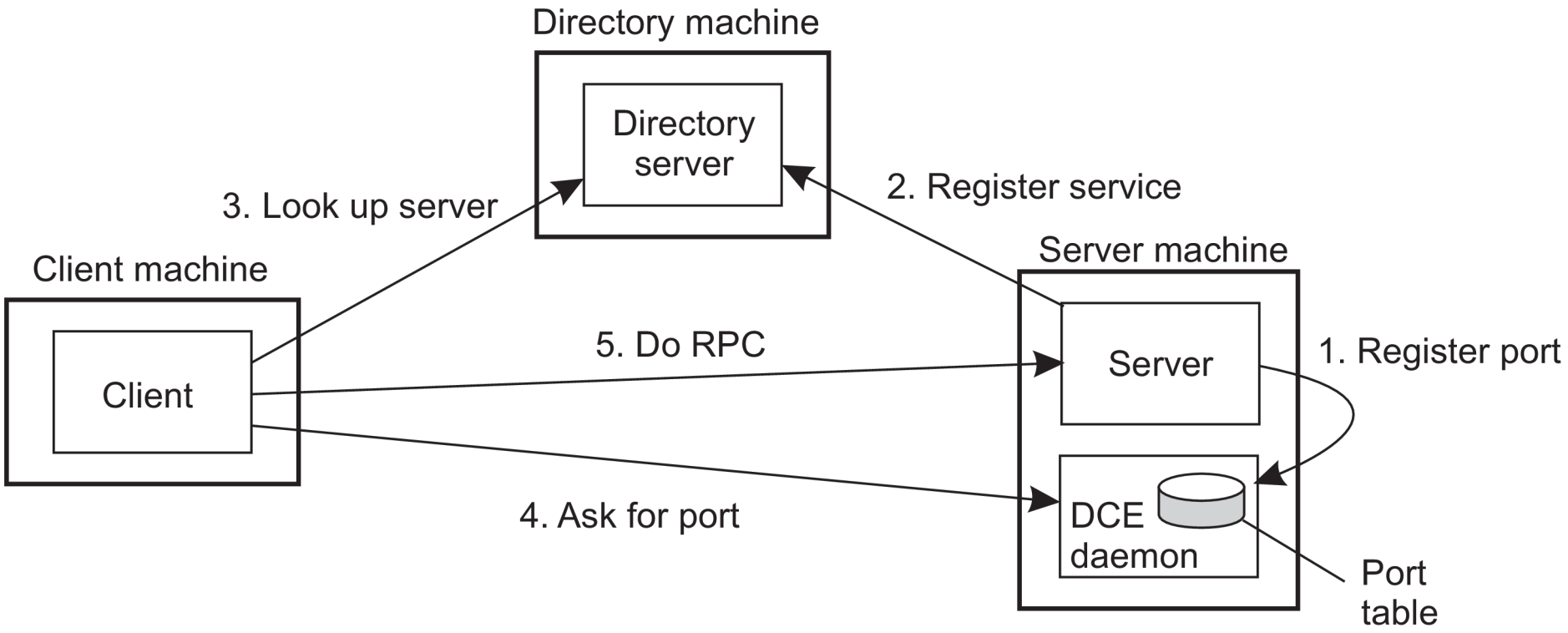
Múltiplas chamadas RPC



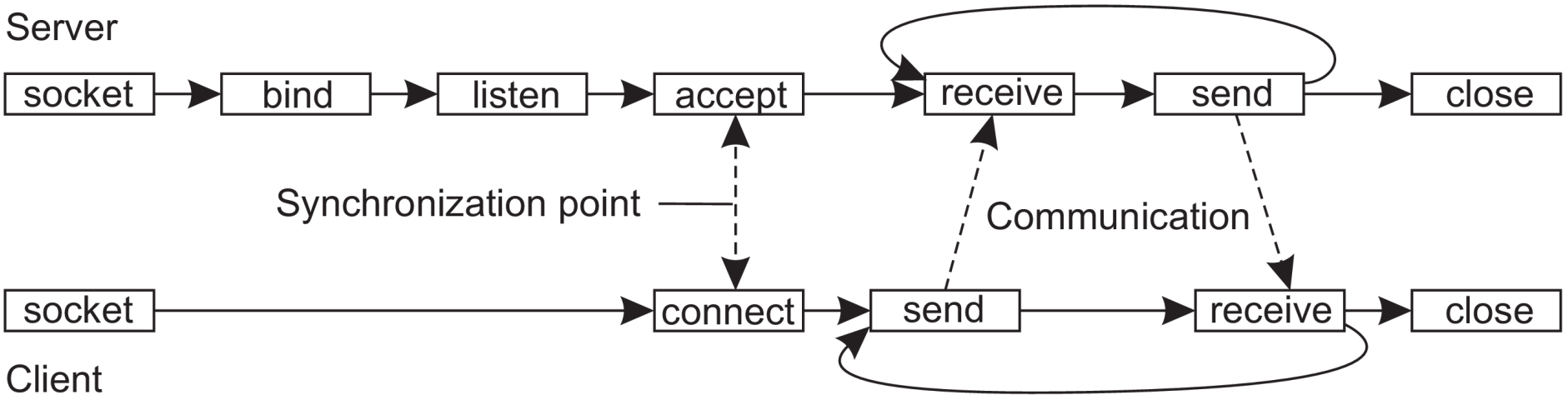
RPC na prática (em Linguagem C)



Ligação Cliente-Servidor (DCE)



Sockets TCP


Server
socket
bind
listen
accept
receive
send
close
Synchronization point
Communication
socket
connect
send
receive
close
Client

Sockets (com ZeroMQ)

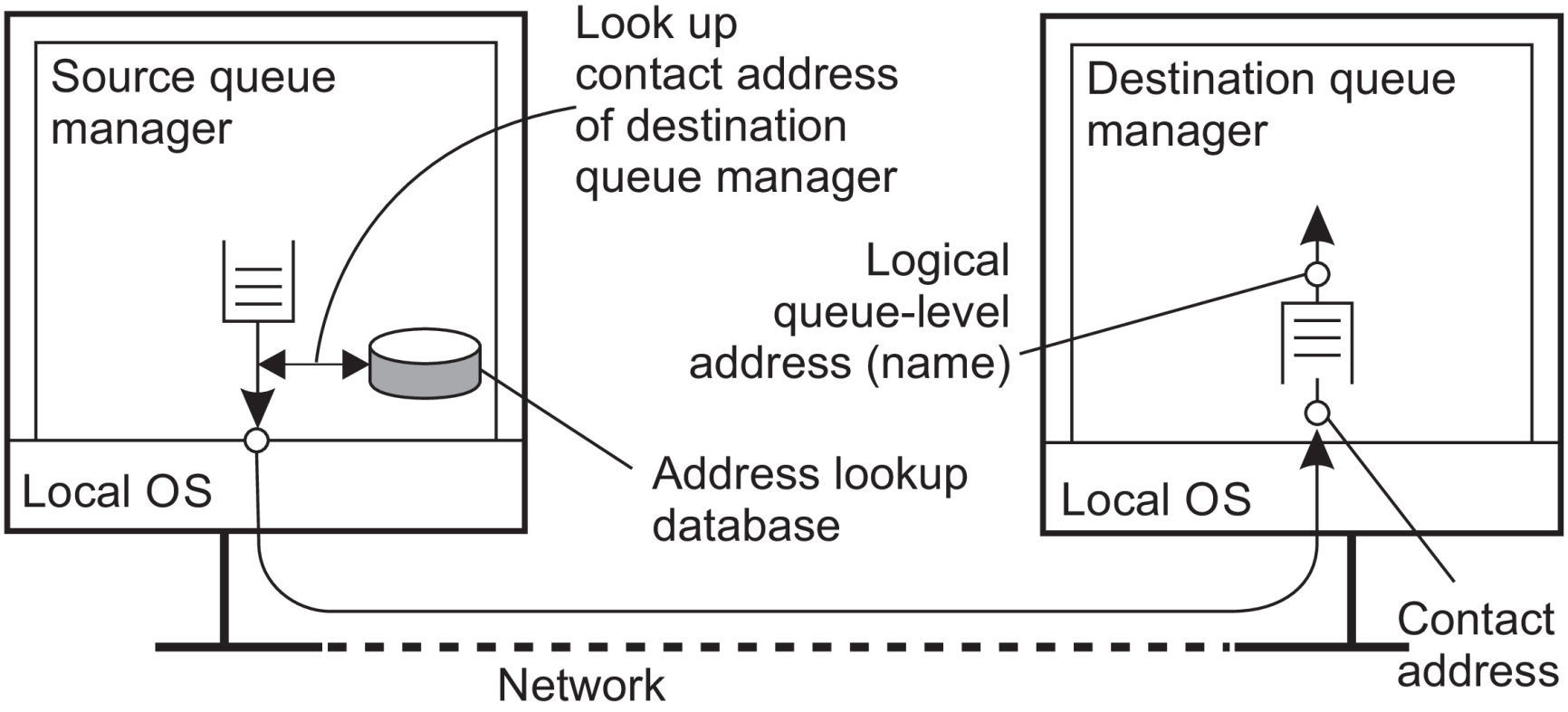
- Camada de abstração de alto nível para sockets
- Criação de pares P, Q (um para enviar outro para receber)
- Toda comunicação é assíncrona.

- Padrões:
  - Request-reply
  - Publish-subscribe
  - Pipeline

Queues (Filas)

- Comunicação assíncrona persistente através de um middleware de filas (queues). As filas são buffers nos servidores de comunicação
- Operações:
  - Put – adiciona uma mensagem a uma fila especifica
  - Get – bloqueia até que haja uma mensagem na fila, remova a primeira mensagem
  - Poll – verifica a existência de mensagens numa dada fila, remove a primeira mensagem. Nunca bloqueia
  - Notify – instala um *handler* que será chamado quando uma mensagem for colocada numa determinada fila.

Modelo Geral

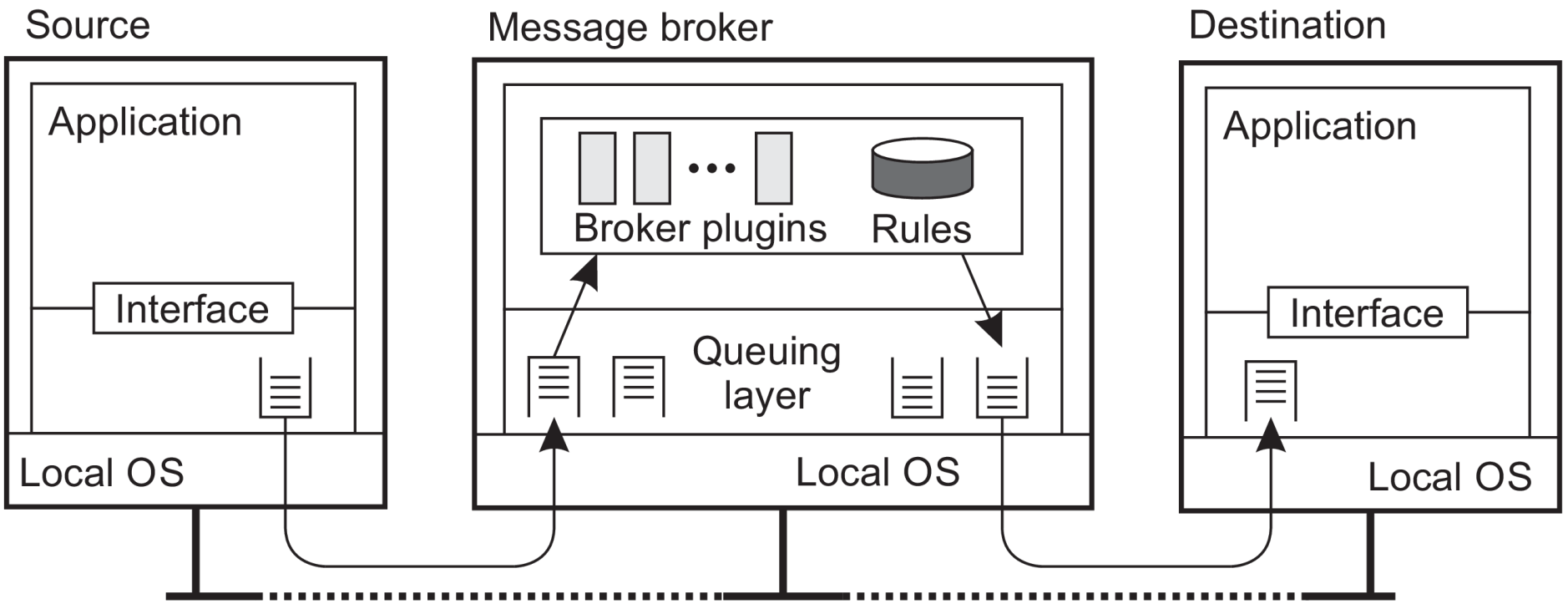


Message Broker

- Num sistema de message queueing (MQ) é assumido um protocolo de mensagens comum.

- Funções do Broker:
  - Transformar mensagens no formato correcto
  - Application Gateway
  - Funcionalidades de roteamento (ex. publish-subscribe)

Arquitectura de um Message Broker



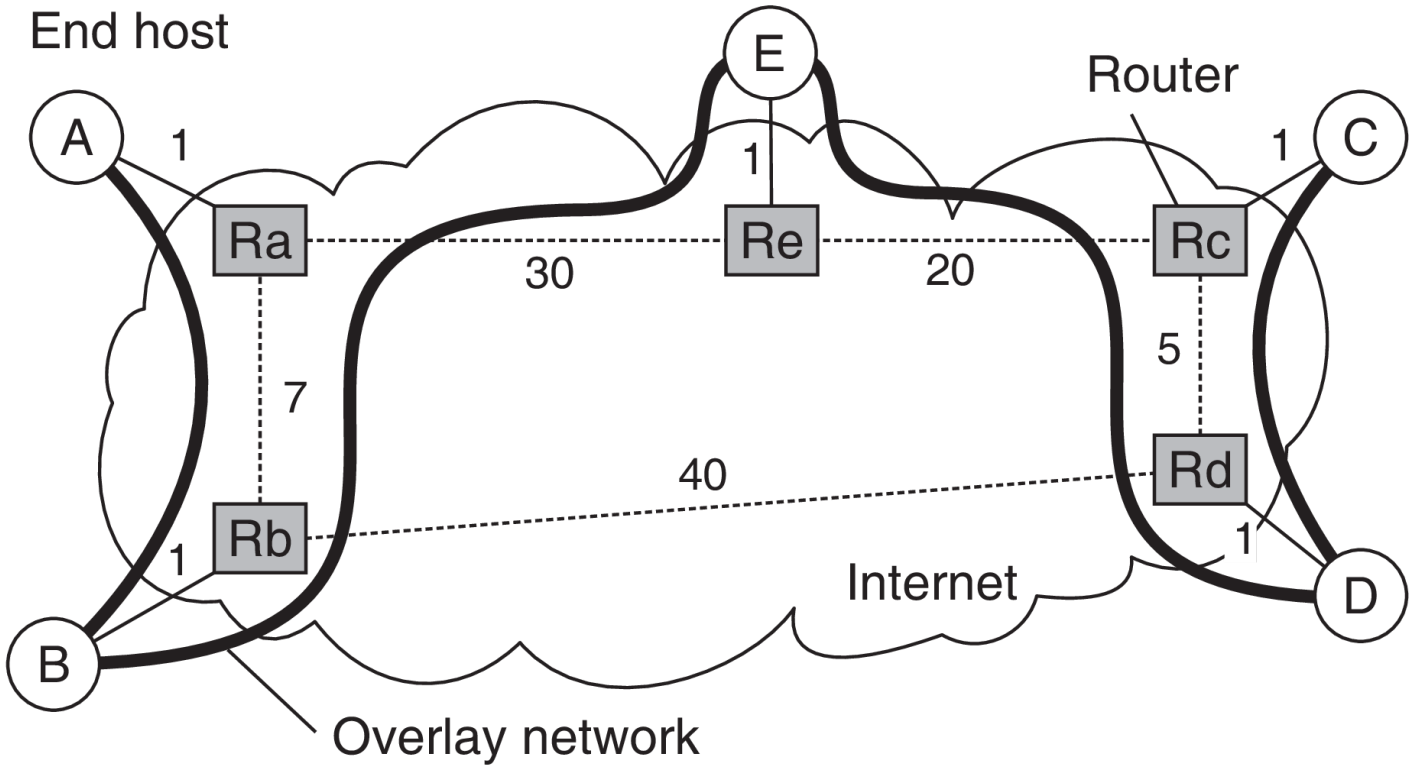
Application-level Multicasting

- Organizar nós de um sistema distribuído numa rede *overlay* e usar essa rede para disseminar dados:
  - Tipicamente em árvore, com caminhos únicos
  - Alternativamente em *mesh*, requere algum tipo de *routing*

ALM: custos

Link stress: quantas vezes a mesma mensagem passa no mesmo link de rede?

Stretch: rácio em atrasos entre o caminho ALM e o caminho de rede.

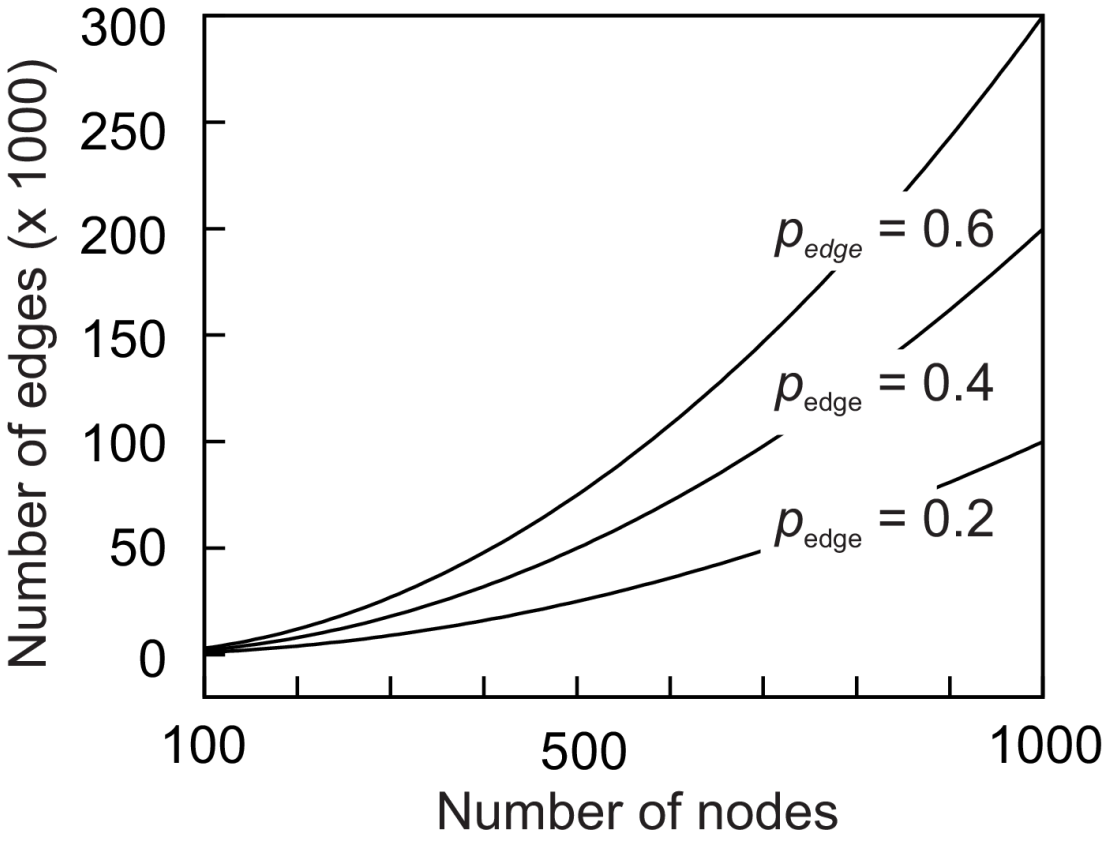


Flooding

Um nó envia uma mensagem para cada um dos seus vizinhos. Cada vizinho re-envia a mensagem para os seus vizinhos à excepção de P, e só se não tinha recebido a mensagem anteriormente.

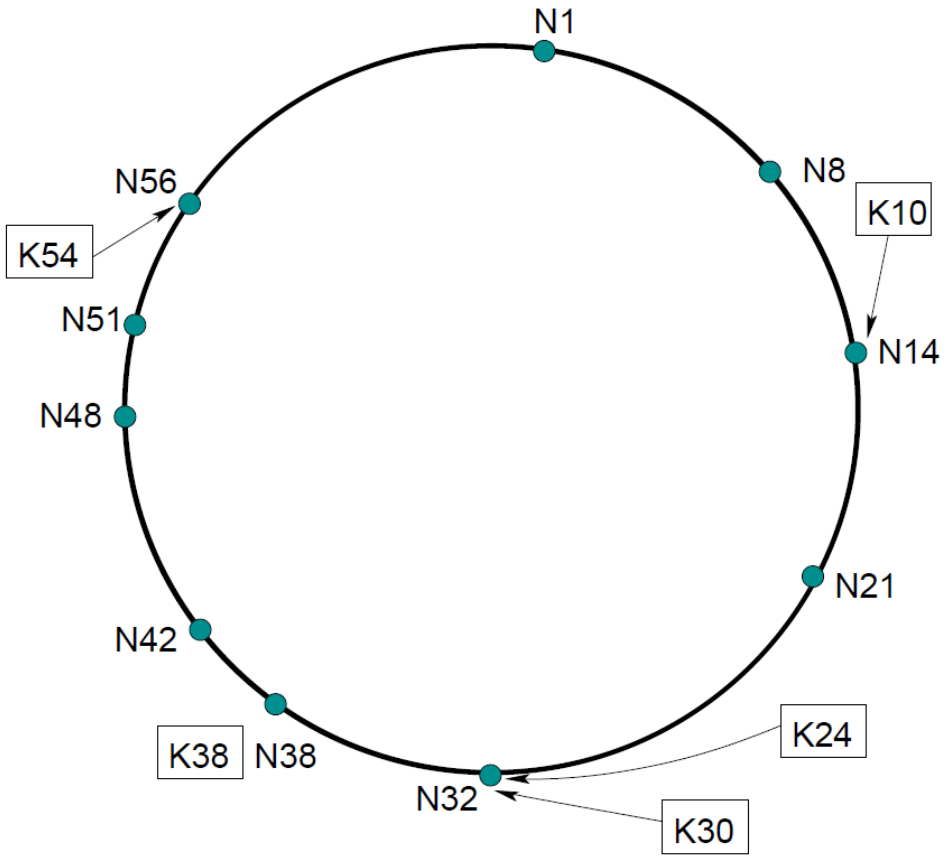
Outros algoritmos:

- Epidimológicos
- Anti-entropia
- Rumores



Chord

- Originalmente proposto por Ion Stoica et al:
  - https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf
- É um protocol de pesquisa P2P escalável para aplicações da Internet
- É também um algoritmo para hash tables distribuídas (DHT)
- Disponibiliza um meio determinístico de localizar recursos, serviços e pares.
- Recursos são expressos como pares `Key, Value`
- Balanciamento de carga através de hashing consistente
- routing tables pequenas: `log n`
- routing delay baixo: `log n` hops
- protocolo para gestão de entradas/saídas rápido (`polylog` time)

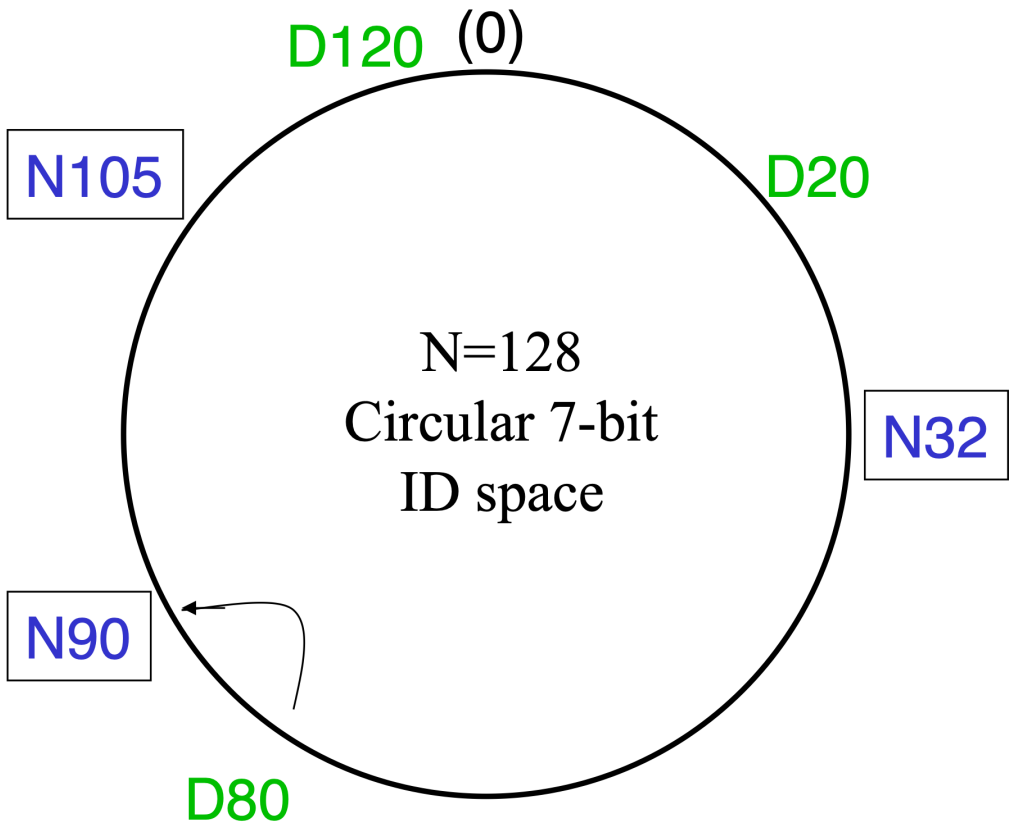


Consistent Hashing

Atribuição de identificadores de nós e objectos apartir de uma chave de **m-bit**

Ordena os nós em volta de um anél de identificadores recorrendo às chaves para ordenar (0 → 2^m-1). Ao anél damos o nome de **Chord Ring**.

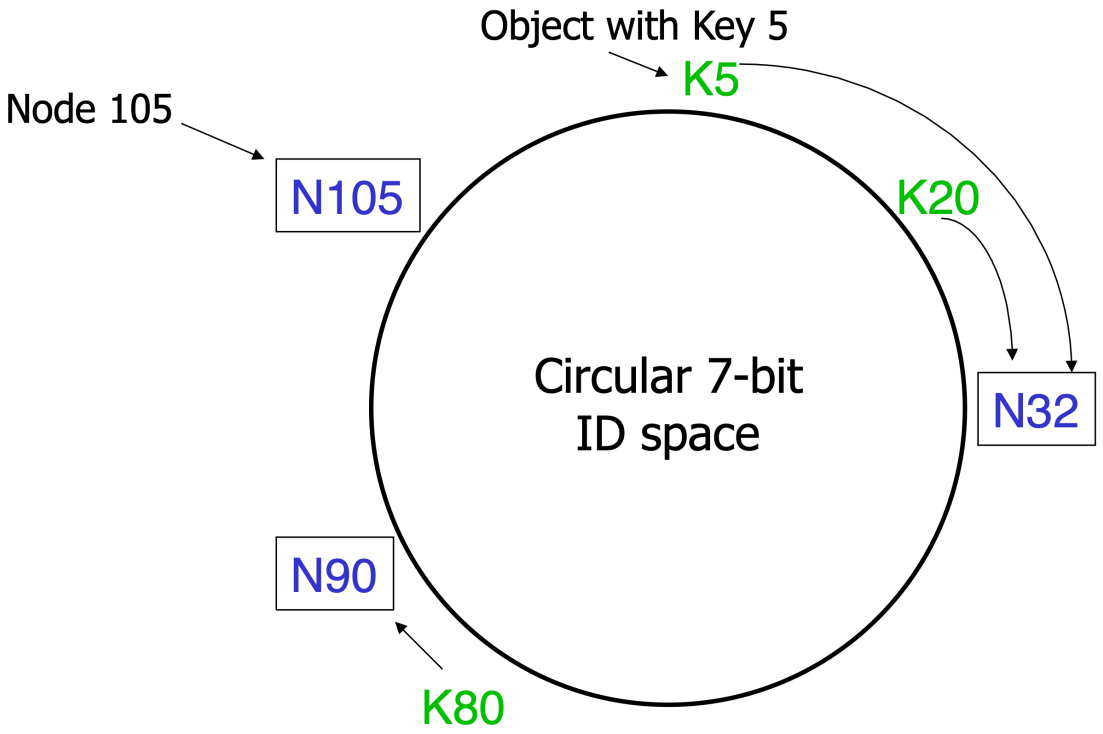
Objecto com a chave k é atribuido ao ***primeiro nó*** cuja chave seja **≥ k** (chamamos de **nó successor** da chave k)



Nó 90 é o **sucessor** do Documento 80

Consistent hashing (2)

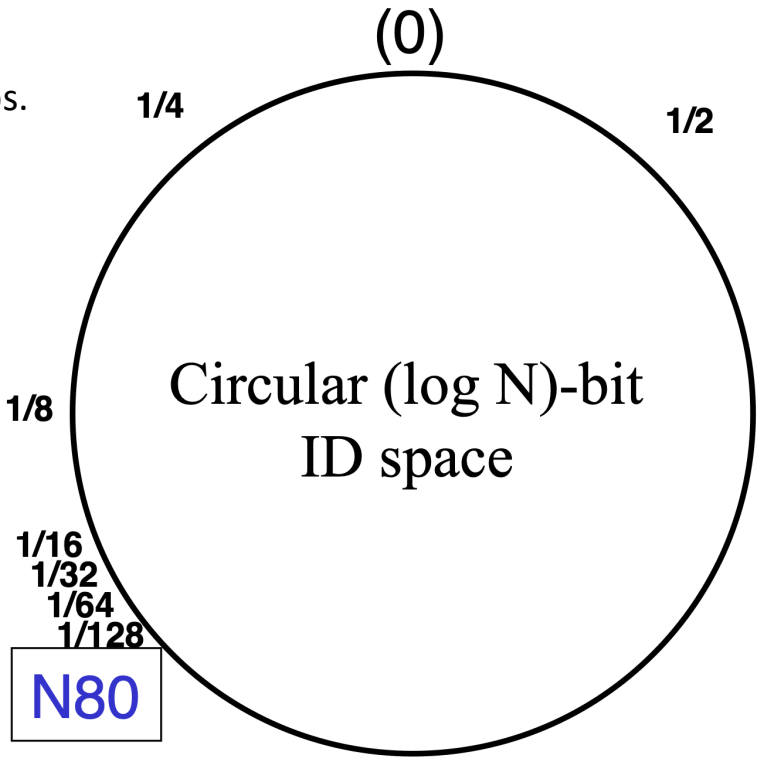
Um objecto com a chave **k** é armazenado no seu **sucessor** (nó com chave ≥ k)



The log N *Fingers*

Cada nó conhece apenas **log N** outros nós.

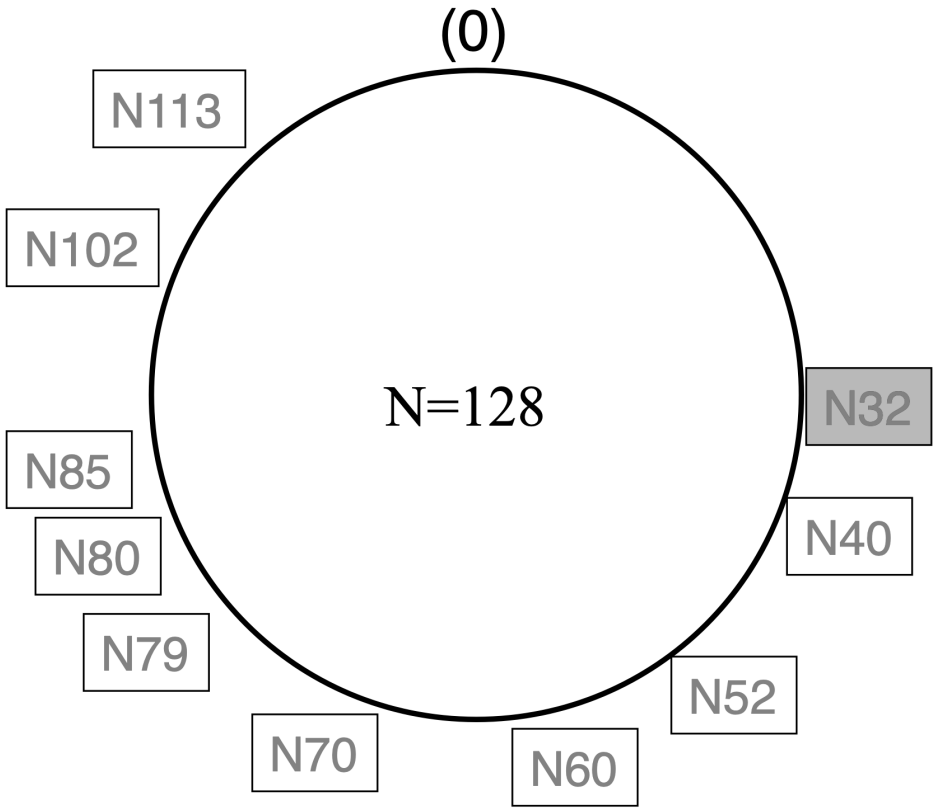
i	$n+2^i$
0	81
1	82
2	84
3	88
4	96
5	112
6	144 → 16



Chord Finger Table

Entrada i de um nó: primeiro nó ≥ $n + 2^i - 1$

i	key	node
0	**33** → 33	N40
1	**34** → 35	N40
2	**36** → 39	N40
3	**40** → 47	N40
4	**48** → 63	N52
5	**64** → 95	N70
6	**96** → 31	N102



Lookup

- Node 32
  - lookup(82): 32 → 70 → 80 → 85

key	node
33 → 33	N40
34 → 35	N40
36 → 39	N40
40 → 47	N40
48 → 63	N52
64 → 95	N70
96 → 31	N102

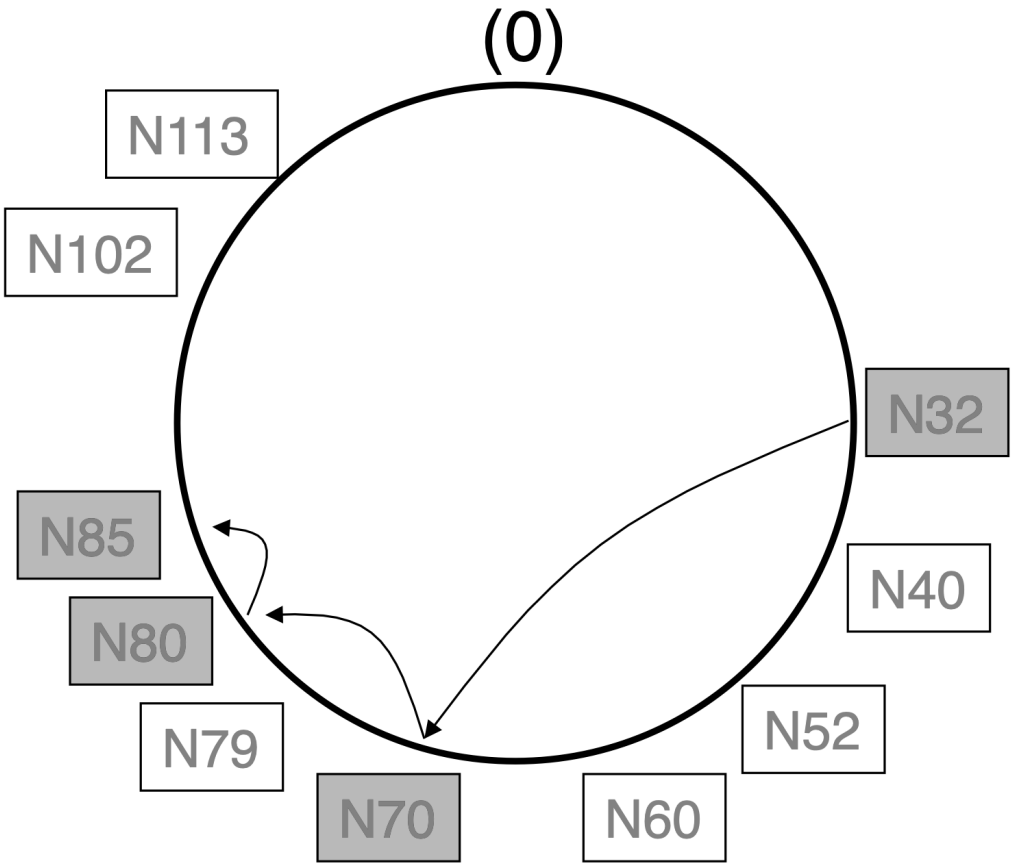
N32 Finger Table

key	node
71 → 71	N79
72 → 73	N79
74 → 77	N79
78 → 85	N80
86 → 101	N102
102 → 5	N102
6 → 69	N32

N70 Finger Table

key	node
81 → 81	N85
82 → 83	N85
84 → 87	N85
88 → 95	N102
96 → 111	N102
112 → 15	N113
16 → 79	N32

N80 Finger Table



Join

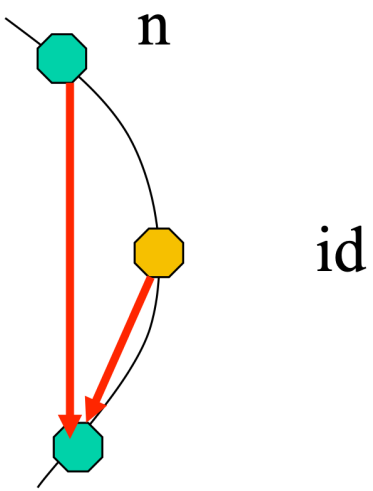
1. Inicializar o **predecessor** e **finger tables** do **novo nó**.

   Conhecimento do predecessor é util na estabilização

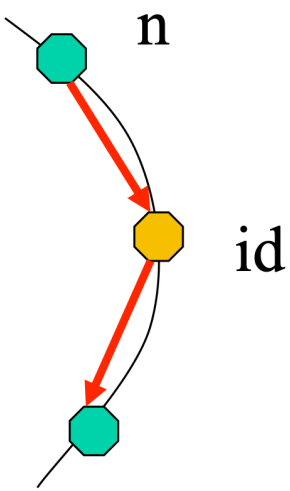
2. Actualizar o **predecessor** e **finger tables** dos **nós existentes**.

   Notificar os nós que devem incluir o novo nó nas suas tabelas.

3. Transferir objectos que o necessitem para o novo nó.



Successor(n)



Finally