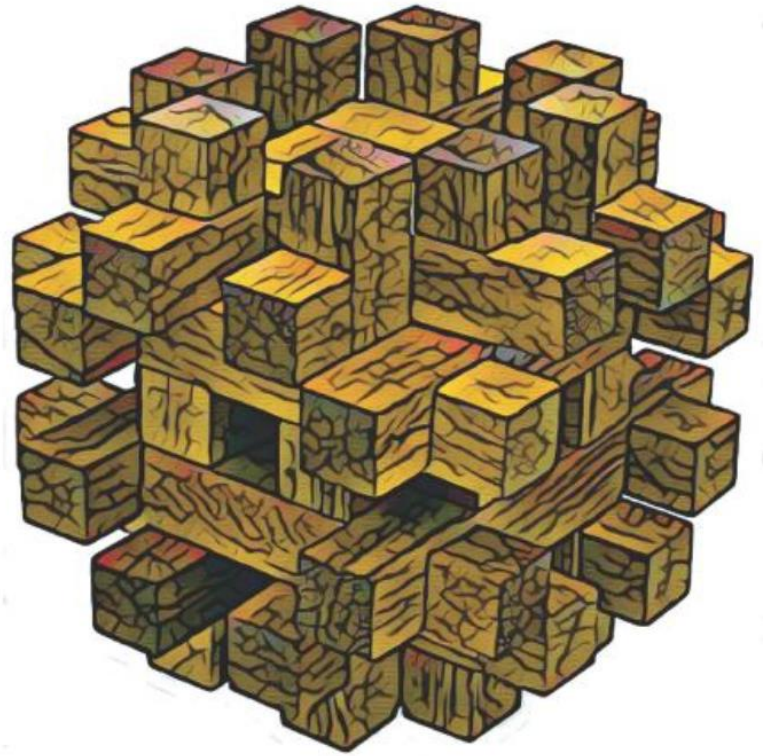
Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



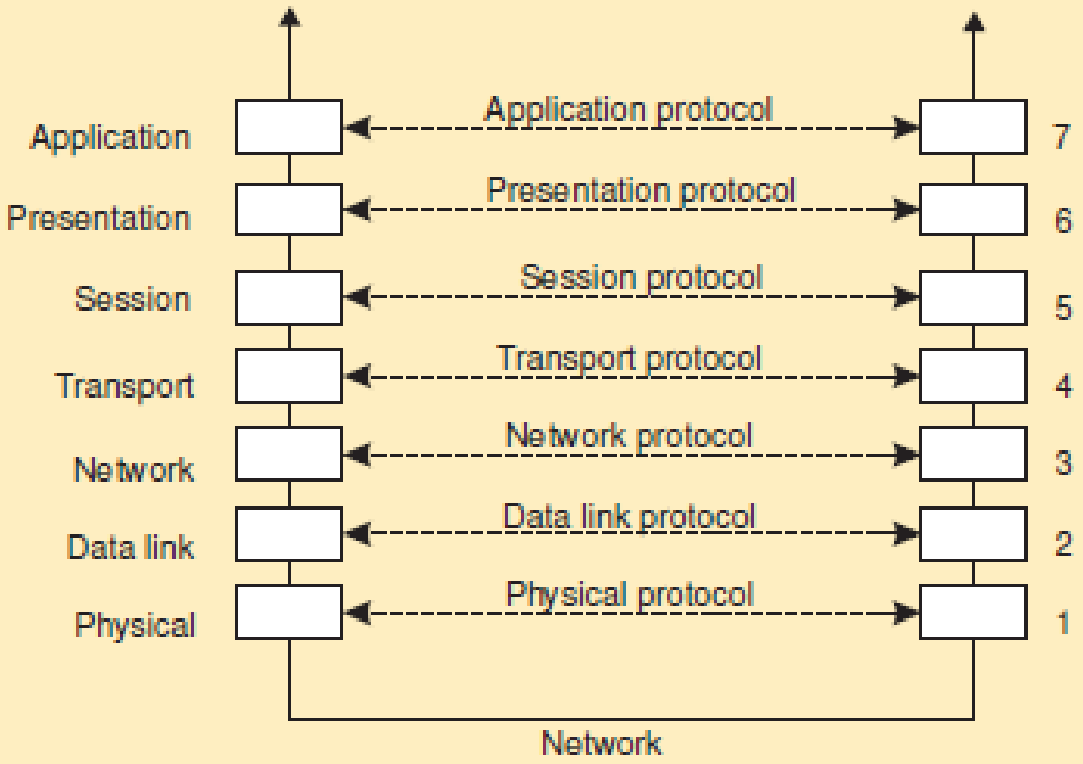
3th Edition – Version 3.03 - 2022

Capítulo 4

Comunicação

Quinta-feira, 21 de Julho de 2022

MODELO BÁSICO DE REDES



PROBLEMAS

- Foco somente em passagem de mensagem
- Funcionalidade frequentemente indesejada e não necessária
- Viola transparência de acesso

CAMADAS DE BAIXO NÍVEL

RECAPITULANDO

- Camada Física: contém a especificação e implementação de bits, e sua transmissão entre despachante e recebedor
- Camada Data Link: prescreve a transmissão de uma série de bits em um Quadro (frame) para permitir controle de fluxo e de erros
- Camada de Rede: descreve como pacotes em uma rede de Computadores deverão ser roteados.

OBSERVAÇÃO

- Para muitos sistemas distribuídos, a interface de mais baixo nível é a camada de rede.

CAMADAS DE TRANSPORTE

IMPORTANTE

- A camada de transporte provê as facilidades de comunicação para maioria dos sistemas distribuídos.

PROTOCOLOS INTERNET PADRÃO

- TCP: orientado a conexão (connection-oriented), confiável, comunicação orientada a streams.
- UDP: não confiável (melhor esforço) comunicação por datagrama

CAMADA MIDDLEWARE

OBSERVAÇÃO

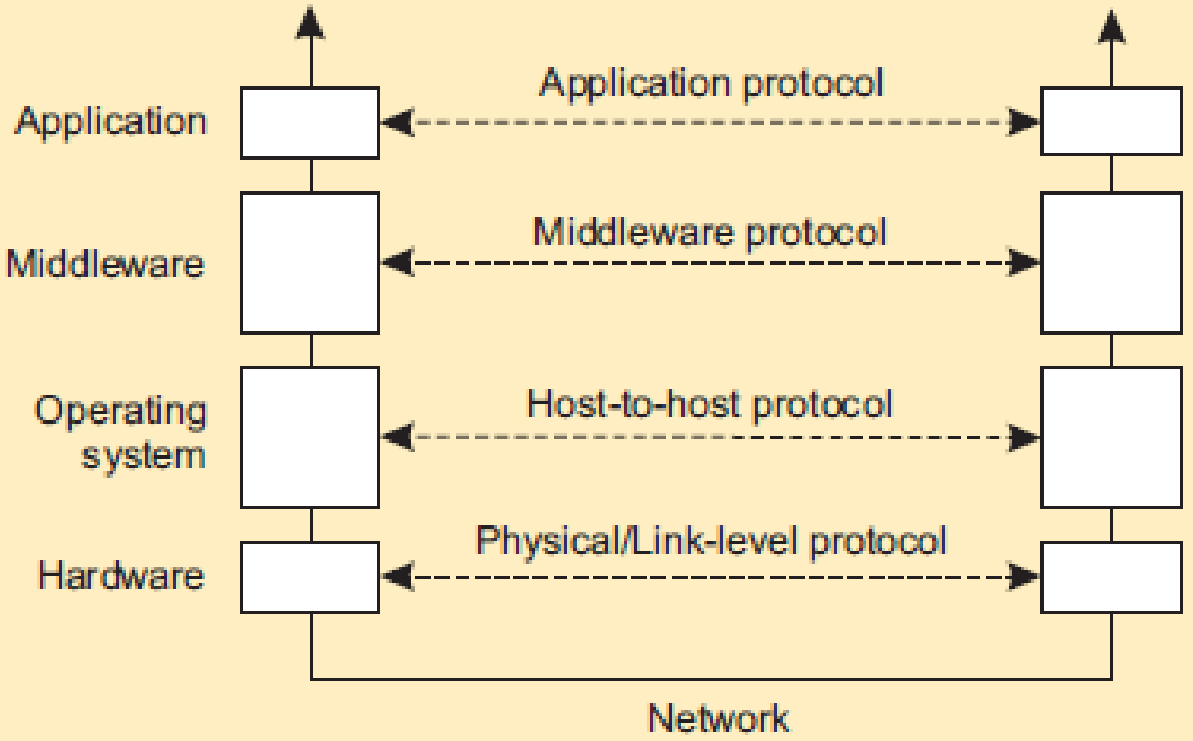
Middleware foi desenvolvido para prover serviços comuns e protocolos que podem ser usados por muitas diferentes aplicações

- Um rico conjunto de protocolos de comunicação
- (Un)marshaling de dados, necessário para sistemas integrados
- Protocolos de Nomes, para permitir o compartilhamento de recursos de forma fácil
- Protocolos de segurança para comunicação segura
- Mecanismos de escalagem, tais como replicação e caching

NOTA

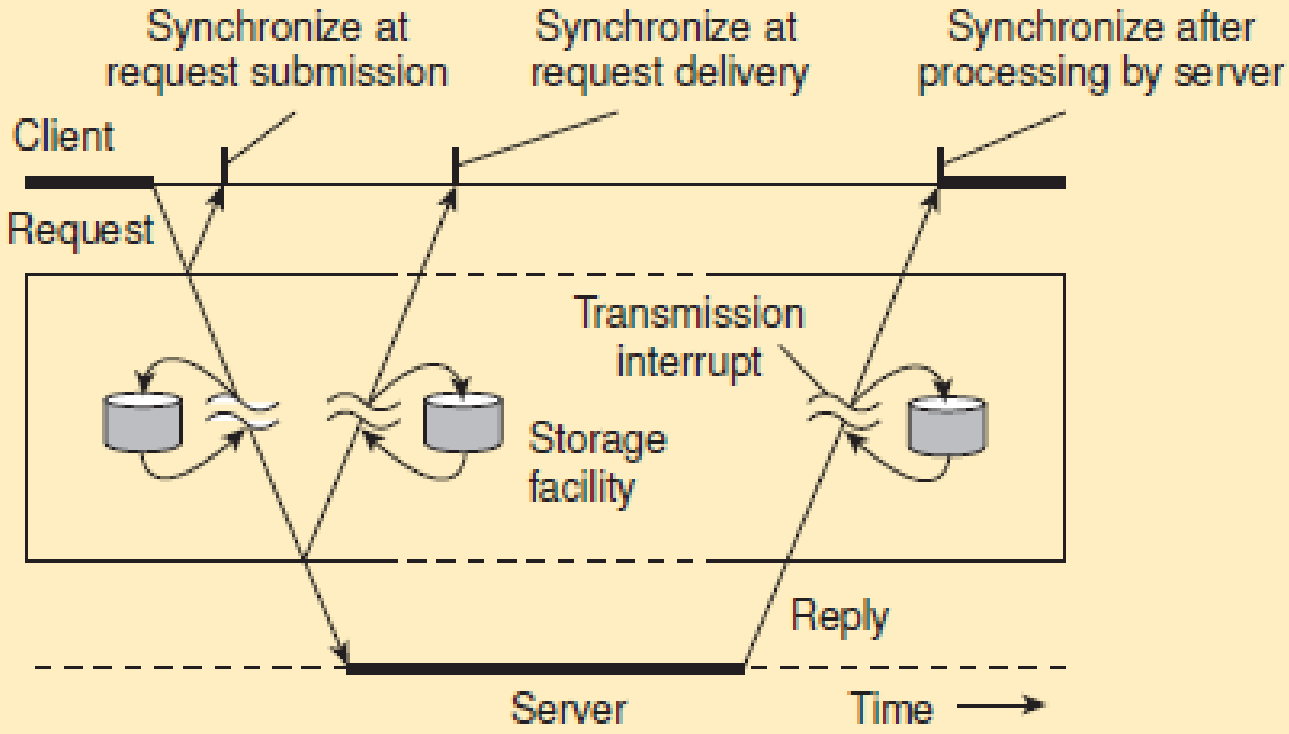
- O que sobra são protocolos específicos de aplicação ... Tais como?

UM ESQUEMA ADAPTADO DE CAMADAS



TIPOS DE COMUNICAÇÃO

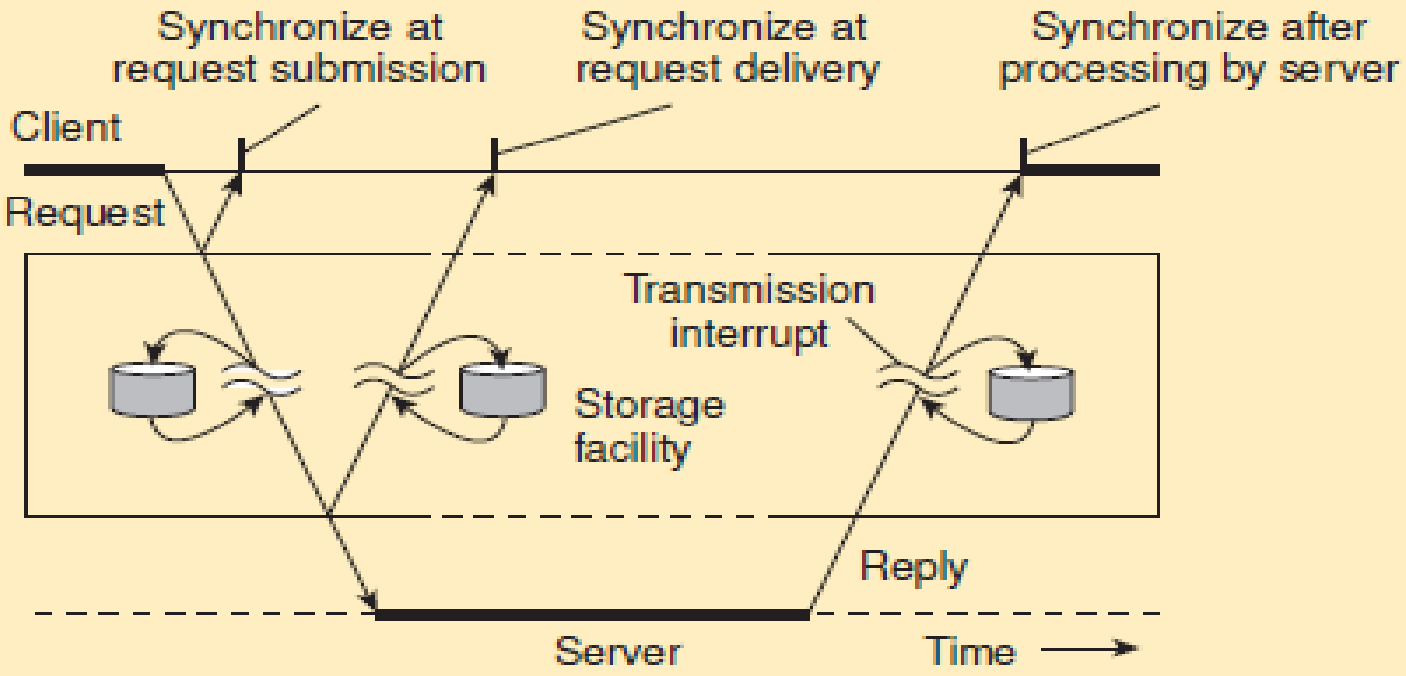
DISTINÇÃO ..



- Comunicação Transiente X persistente
- Comunicação Assíncrona X síncrona

TIPOS DE COMUNICAÇÃO

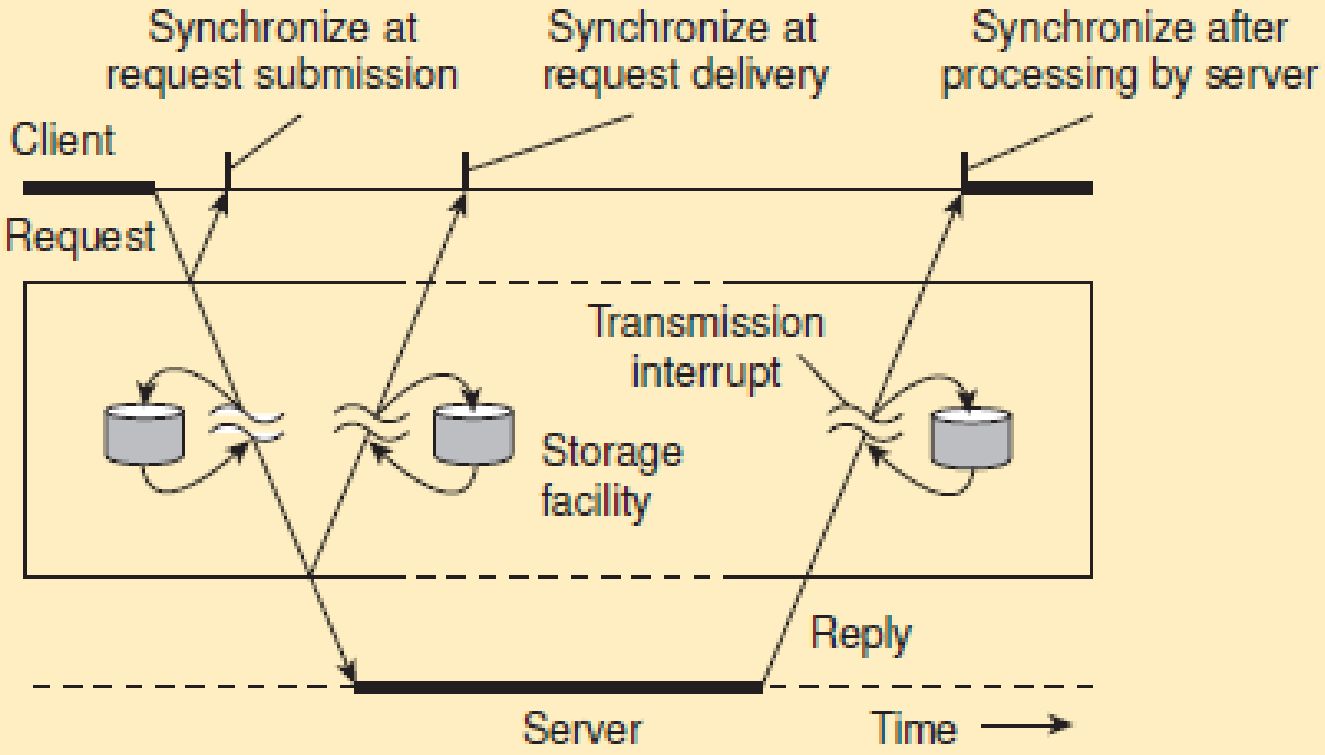
TRANSIENTE X PERSISTENTE



- Comunicação Transiente: Servidor de comunicação descarta a mensagem quando ela não pode ser entregue para o próximo servidor ou destinatário.
- Comunicação Persistente: Uma mensagem é armazenada no servidor de comunicação pelo tempo que demorar pra ela ser entregue.

TIPOS DE COMUNICAÇÃO

LOCAIS PARA SINCRONIZAÇÃO



- Na submissão da requisição
- Na entrega da requisição
- Depois do processamento da requisição

CLIENTE SERVIDOR

ALGUMAS OBSERVAÇÕES ..

Computação cliente/servidor é geralmente baseada no modelo de comunicação transiente síncrona:

- Cliente e servidor precisam estar ativos no momento da comunicação
- Cliente emite reqisições e bloqueia até receber resposta
- Servidor essencialmente espera somente por requisições entrantes, e subsequentemente as processa.

PROBLEMAS DE COMUNICAÇÃO SÍNCRONA

- Cliente não pode fazer nenhum outro trabalho enquanto espera por resposta
- Falhas tem que ser manuseadas imediatamente: o cliente está esperando
- O modelo pode simplesmente não ser apropriado (mail, news)

PASSANDO MENSAGENS

MIDDLEWARE ORIENTADO A MENSAGENS

Foca na comunicação de alto nível assíncrona persistente:

- Processos enviam mensagens uns para outros, que são enfileirados (queued)
- Remetente não precisa esperar por uma resposta imediata, e pode fazer outras coisas
- Middleware normalmente garante tolerância a falhas

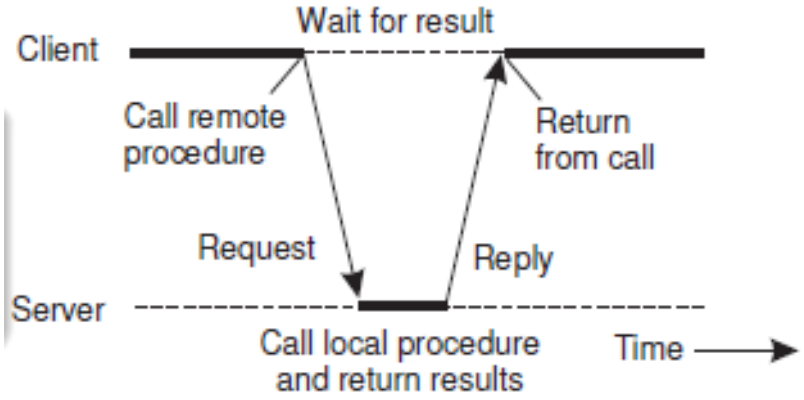
RPC - OPERAÇÃO BÁSICA

OBSERVAÇÕES

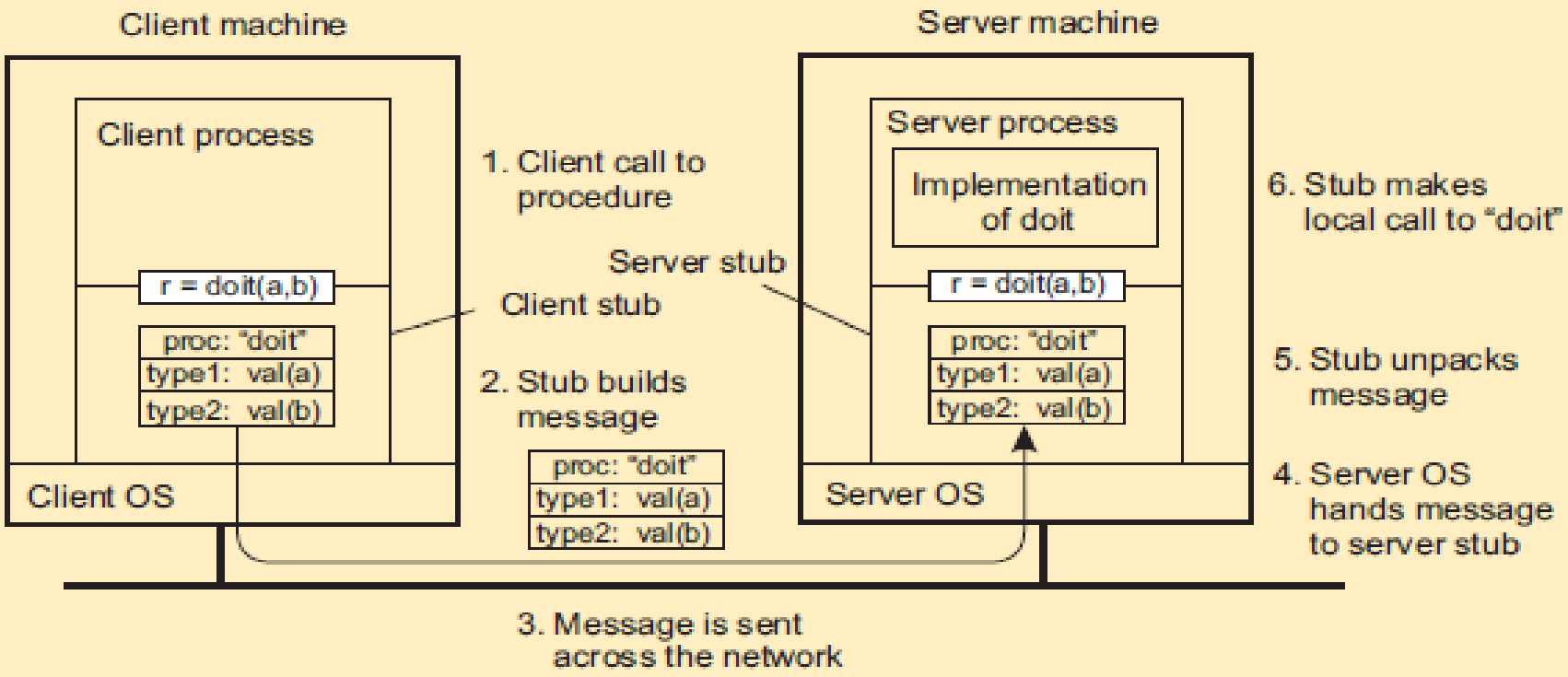
- Desenvolvedores normalmente usam o modelo procedural simples
- Procedimentos bem engenhados operam isolados (black box)
- Não existe uma razão fundamental para não executar procedimentos em máquinas separadas

CONCLUSÃO

Comunicação entre chamador e chamado pode ser escondida usando um mecanismo de chamada de procedimento.



RPC OPERAÇÃO BÁSICA



1. Procedimento Cliente chama sutb cliente
2. Stub monta mensagem; chama SO local
3. SO envia mensagem para SO remoto.
4. SO remote passa a mensagem para o stub
5. Stub desempacota parâmetros e chama servidor
6. Servidor faz uma chamada local; retorna resultado para stub.
7. Stub monta mensagem; chama o SO.
8. SO envia mensagem para o SO do cliente.
9. SO do cliente passa a mensagem para o stub.
10. Stub do cliente desempacota o resultado; retorna para o cliente.

RPC PASSAGEM DE PARÂMETROS

OBSERVAÇÕES

- As máquinas cliente e servidor podem ter diferentes representações de dados (big/little endian)
- Wrapping de parâmetros significa transformer um valor em uma sequência de bytes
- Cliente e servidor tem que concordar em uma mesma codificação:
  - Como os valores de dados básicos são representados (integers, floats, characters)
  - Como os valores complexos são representados (arrays, unions)

CONCLUSÃO

Cliente e servidor precisam interpreter corretamente mensagens, e transformá-las em representaçoes dependentes de máquina.

RPC PASSAGEM DE PARÂMETROS

ALGUMAS SUPOSIÇÕES

- Copy in/copy out semântica: enquanto o procedimento é executado, nada pode ser assumido sobre os valores dos parâmetros
- Todos dados que precisam ser operados são passados como parâmetros. Excluí passagem de referência para (global) dados.

CONCLUSÃO

Transparência de acesso total não pode ser realizada

...

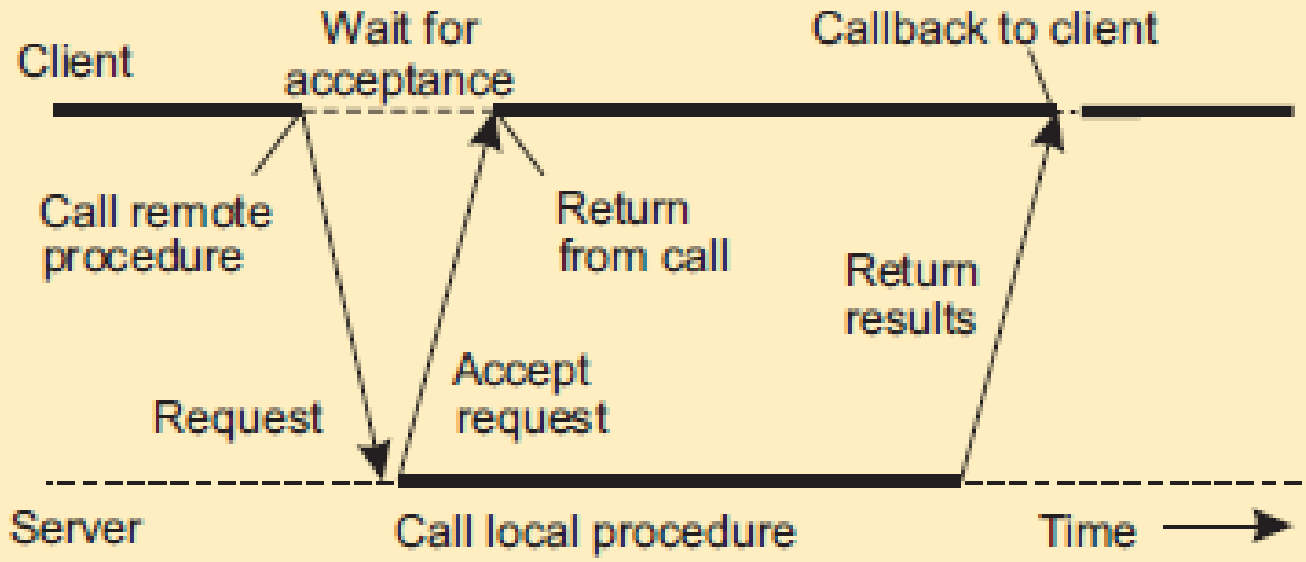
UM MECANISMO DE REFERÊNCIA REMOTA MELHORA A TRANSPARÊNCIA DE ACESSO

- Referências remotas oferecem acesso unificado a dados remotos
- Referências remotas podem ser passadas como parâmetros em RPCs
- Nota: stubs podem as vezes serem usadso como tais referências

RPC ASSÍNCRONA

ESSÊNCIA

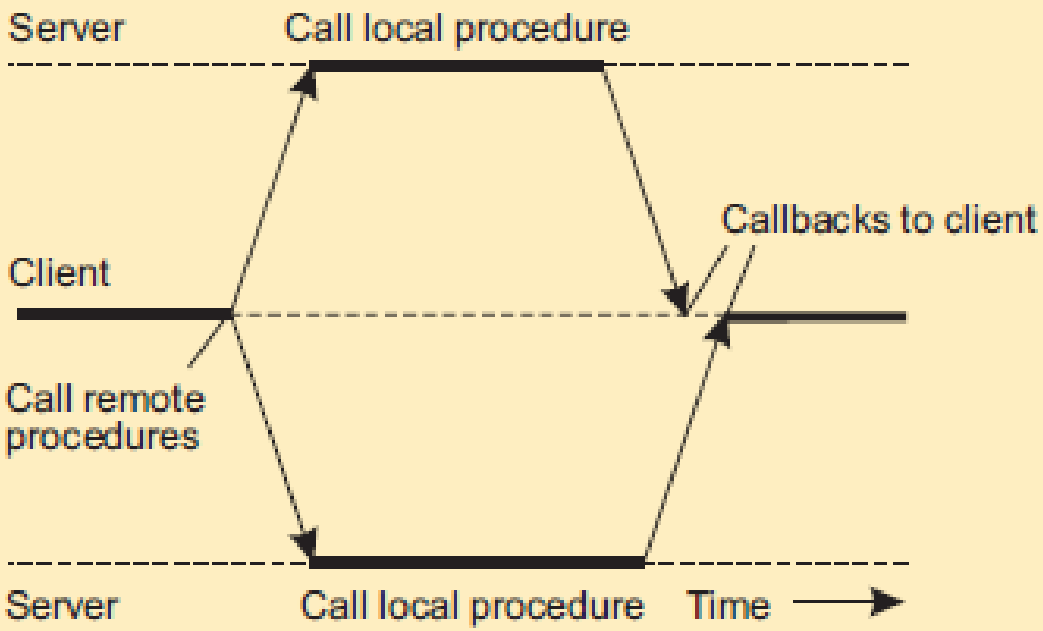
- Tente se livrar do comportamento estrito requisição resposta, mas deixe o cliente continuar sem esperar por uma resposta do servidor.



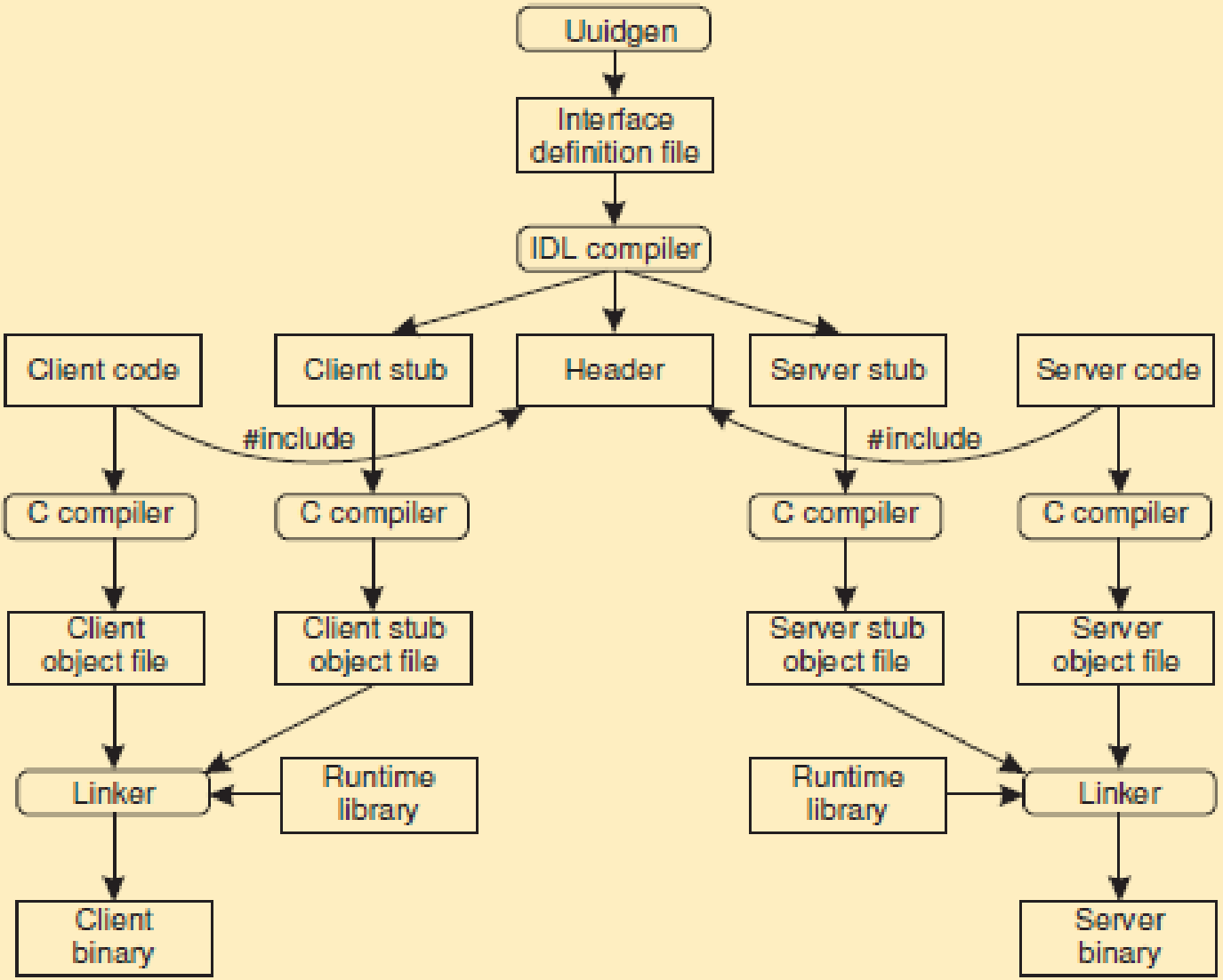
ENVIANDO MÚLTIPLAS RPC

ESSÊNCIA

- Enviando uma requisição RPC para um grupo de servidores.



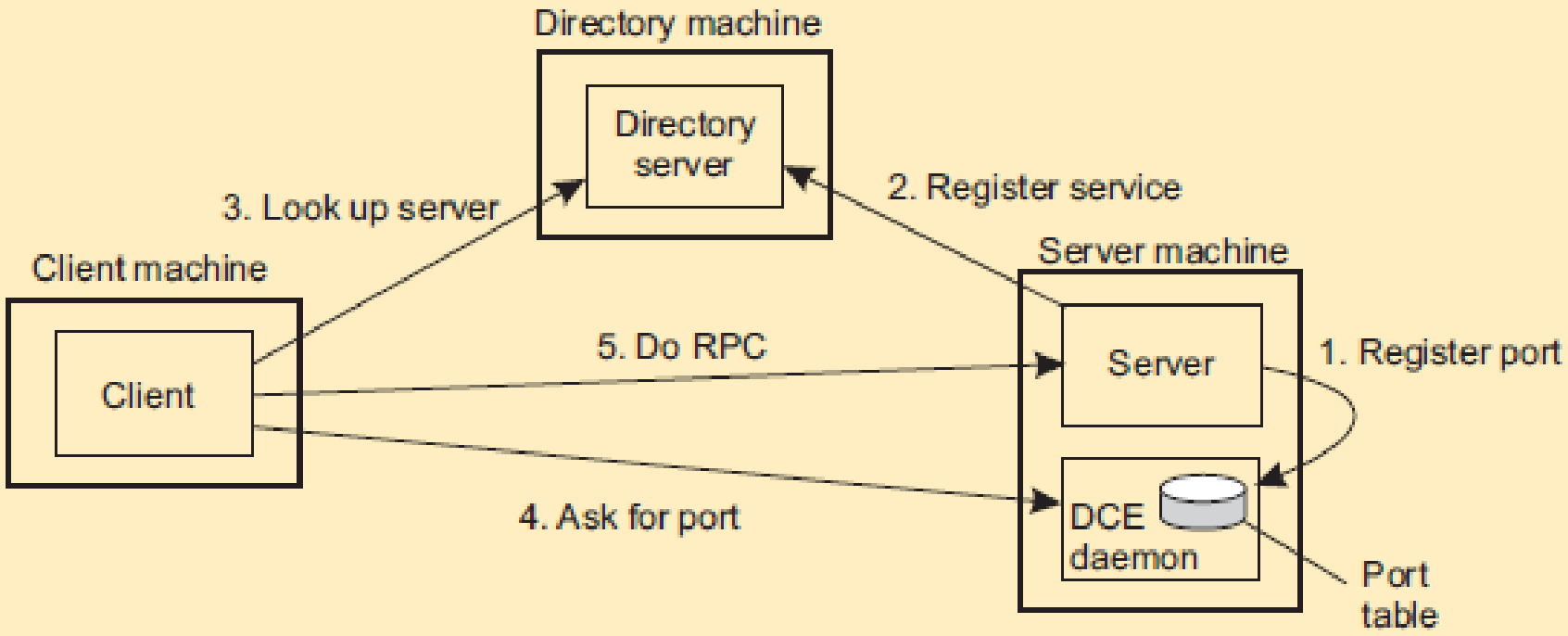
RPC NA PRÁTICA



BIDING CLIENTE/SERVIDOR (DCE)

QUESTÕES

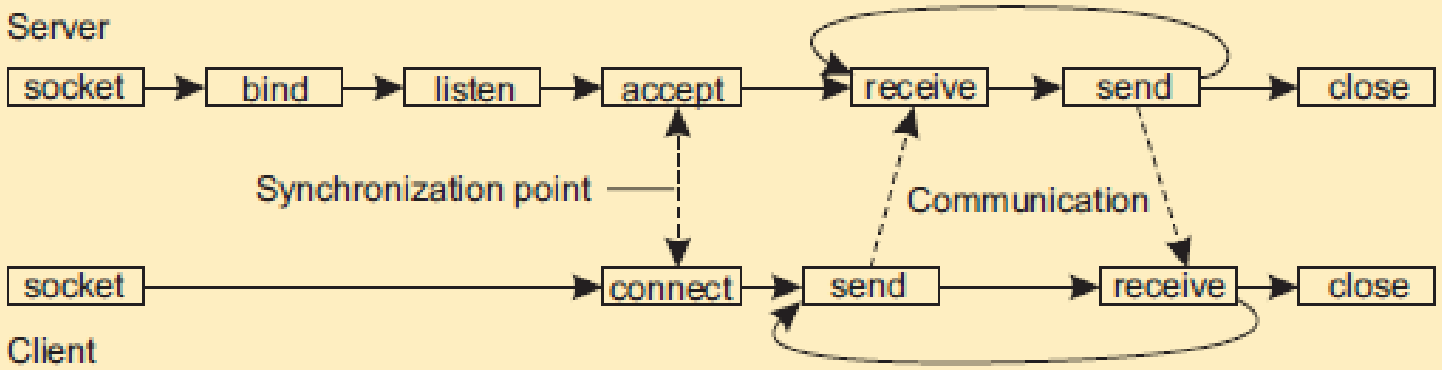
(1) Cliente deve localizar a máquina servidor, e (2) localizar o servidor.



MENSAGENS TRANSIENTES: SOCKETS

API SOCKETS BERKELEY

Operation	Description
socket	Create a new communication end point
bind	Attach a local address to a socket
listen	Tell operating system what the maximum number of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some data over the connection
receive	Receive some data over the connection
close	Release the connection



SOCKETS: CÓDIGO PYTHON

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
```

FACILITANDO O USO DE SOCKETS

OBSERVAÇÃO

- Sockets representa um nível bem baixo de programação e erros podem acontecer facilmente. Entretanto, a forma como eles são usados não muda (como exmplo em ajustes de cliente-servidor).

ALTERNATIVA: ZeroMQ

- Provê um nível mais alto para expressar o pareamento de sockets: um para enviar mensagens do processo P e um correspondente para receber mensagens em Q. Toda comunicação é assíncrona.

TRÊS PADRÕES

- Request-reply
- Publish-subscribe
- Pipeline

REQUEST REPLY

SERVIDOR

```
1  import zmq
2  context = zmq.Context()
3
4  p1 = "tcp://"+ HOST +":"+ PORT1 # how and where to connect
5  p2 = "tcp://"+ HOST +":"+ PORT2 # how and where to connect
6  s  = context.socket(zmq.REP)    # create reply socket
7
8  s.bind(p1)                      # bind socket to address
9  s.bind(p2)                      # bind socket to address
10 while True:
11   message = s.recv()            # wait for incoming message
12   if not "STOP" in message:     # if not to stop...
13     s.send(message + "*")       # append "*" to message
14   else:                         # else...
15     break                       # break out of loop and end
```

CLIENTE

```
1  import zmq
2  context = zmq.Context()
3
4  php = "tcp://"+ HOST +":"+ PORT # how and where to connect
5  s   = context.socket(zmq.REQ)   # create socket
6
7  s.connect(php)                  # block until connected
8  s.send("Hello World")           # send message
9  message = s.recv()              # block until response
10 s.send("STOP")                  # tell server to stop
11 print message                   # print result
```

PUBLISH SUBSCRIBE

SERVIDOR (PUBLICADOR)

```
1  import zmq, time
2
3  context = zmq.Context()
4  s = context.socket(zmq.PUB)          # create a publisher socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.bind(p)                            # bind socket to the address
7  while True:
8    time.sleep(5)                      # wait every 5 seconds
9    s.send("TIME " + time.asctime()) # publish the current time
```

CLIENTE (ASSINANTE)

```
1  import zmq
2
3  context = zmq.Context()
4  s = context.socket(zmq.SUB)          # create a subscriber socket
5  p = "tcp://"+ HOST +":"+ PORT        # how and where to communicate
6  s.connect(p)                         # connect to the server
7  s.setsockopt(zmq.SUBSCRIBE, "TIME")  # subscribe to TIME messages
8
9  for i in range(5):  # Five iterations
10   time = s.recv()   # receive a message
11   print time
```

PIPELINE

SOURCE

```
1  import zmq, time, pickle, sys, random
2
3  context = zmq.Context()
4  me  = str(sys.argv[1])
5  s   = context.socket(zmq.PUSH)          # create a push socket
6  src = SRC1  if me == '1' else SRC2      # check task source host
7  prt = PORT1 if me == '1' else PORT2     # check task source port
8  p   = "tcp://"+ src +":"+ prt           # how and where to connect
9  s.bind(p)                               # bind socket to address
10
11 for i in range(100):                    # generate 100 workloads
12   workload = random.randint(1, 100)     # compute workload
13   s.send(pickle.dumps((me,workload)))   # send workload to worker
```

WORKER (TRABALHADOR)

```
1  import zmq, time, pickle, sys
2
3  context = zmq.Context()
4  me = str(sys.argv[1])
5  r  = context.socket(zmq.PULL)       # create a pull socket
6  p1 = "tcp://"+ SRC1 +":"+ PORT1     # address first task source
7  p2 = "tcp://"+ SRC2 +":"+ PORT2     # address second task source
8  r.connect(p1)                       # connect to task source 1
9  r.connect(p2)                       # connect to task source 2
10
11 while True:
12   work = pickle.loads(r.recv())     # receive work from a source
13   time.sleep(work[1]*0.01)          # pretend to work
```

MPI QUANDO É NECESSÁRIO FLEXIBILIDADE

OPERAÇÕES REPRESENTATIVAS

Operation	Description
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until transmission starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

MESSAGE ORIENTED MIDDLEWARE

ESSÊNCIA

Comunicação assíncrona persistente com suporte de enfileiramento em nível de middleware. Filas correspondem a buffers em servidores de comunicação.

OPERAÇÕES

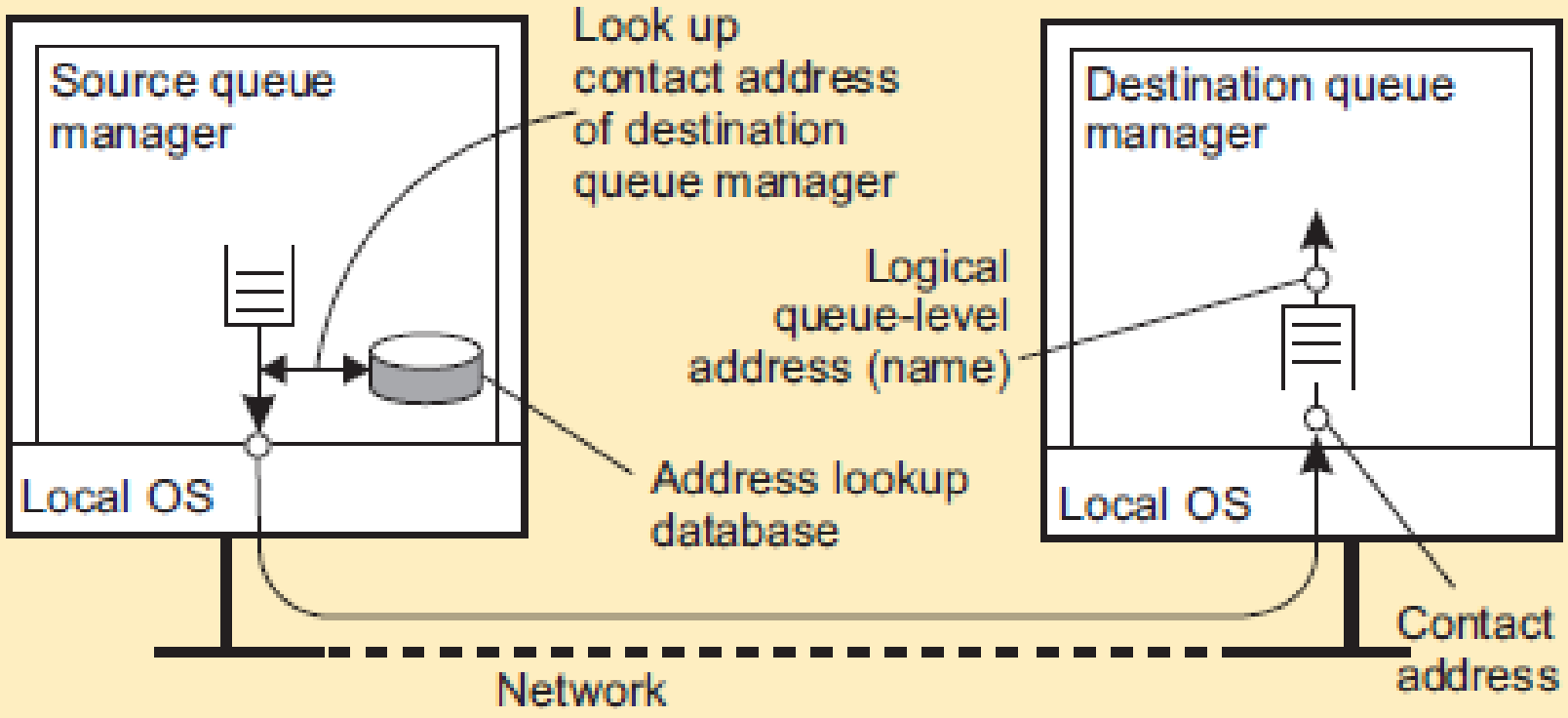
Operation	Description
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

MODELO GERAL

GERENCIADORES DE FILA

Comunicação assíncrona persistente com suporte de enfileiramento em nível de middleware. Filas correspondem a buffers em servidores de comunicação.

ROTEAMENTO



MESSAGE BROKER

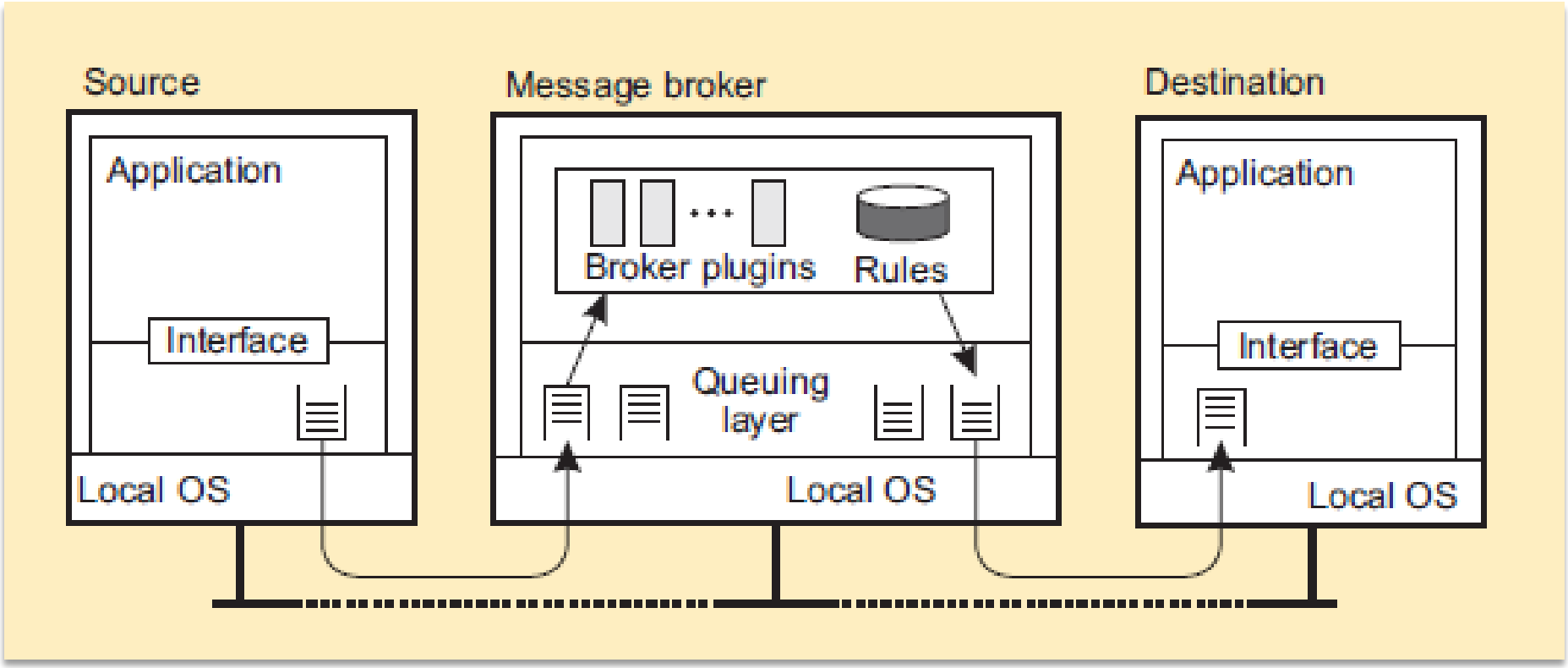
OBSERVAÇÃO

Sistemas de enfileiramento de mensagens assumem um protocolo comum de mensageria: todas as aplicações concordam com o formato da mensagem (i.e. representação e estrutura de dados)

BROKER MANIPULA A HETEROGENEIDADE DE UM SISTEMA MQ

- Transforma mensagens entrantes em fomato do destinatário alvo
- Age como um gateway de aplicação de forma frequente
- Pode prover capacidade de roteamento baseado no tópico (subject-based) (i.e., capacidades publish-subscribe)

MESSAGE BROKER ARQUITETURA GERAL



IBM´s WebSphere MQ

CONCEITO BÁSICO

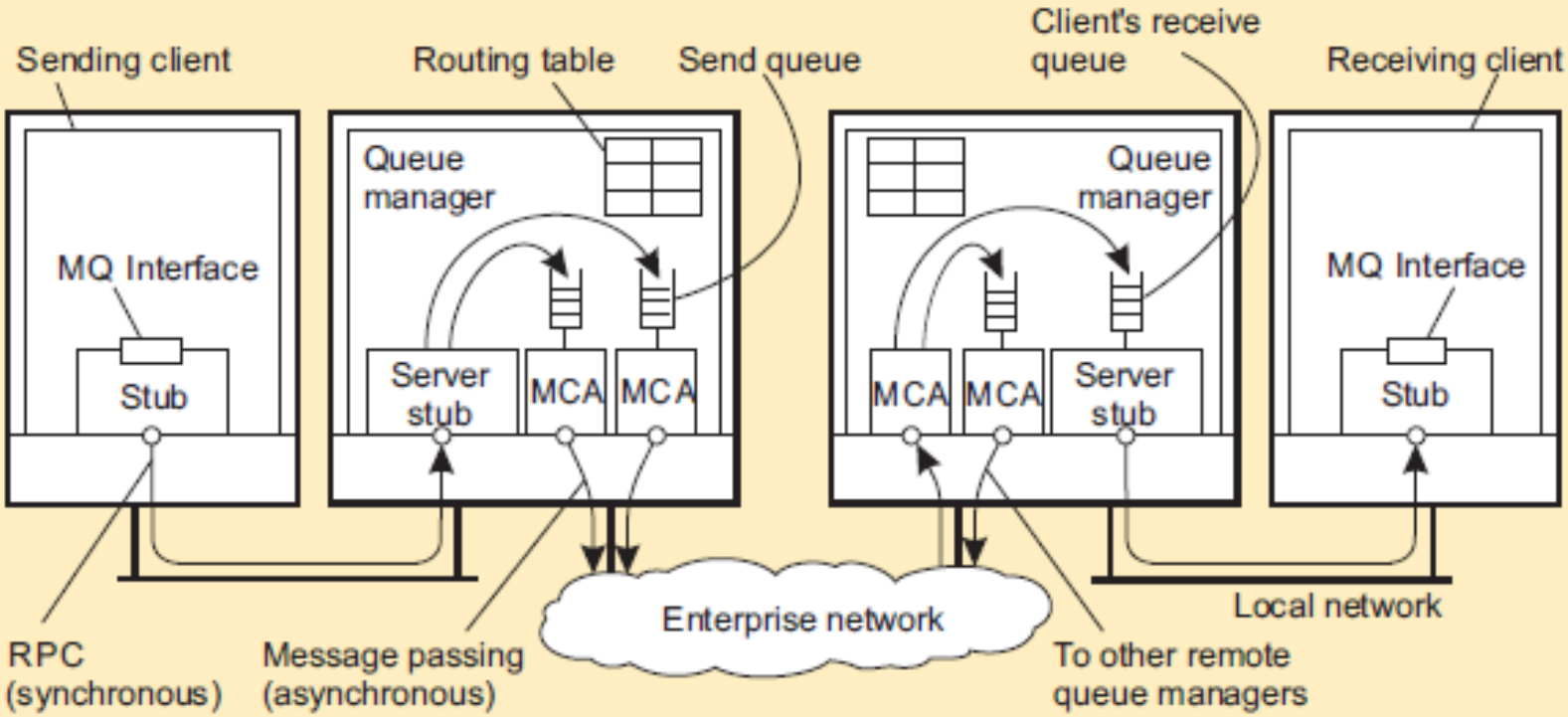
- Mensagens específicas da aplicação a são colocadas e removidas de filas
- Filas reside em um regime do gerenciador da fila
- Processos podem colocar mensagens somente em filas locais, ou através de um mecanismo RPC

TRANSFERÊNCIA DE MENSAGEM

- Mensagens são transferidas entre filas
- transferência de mensagem entre filas em diferentes processos, demandam o uso de um canal (channel)
- Em cada end point de um canal existe um agente de mensagem do canal (message channel agent)
- Message channel agents são responsáveis por:
  - Criação (setup) de canais usando facilidades de comunicação de baixo nível da rede (ex. TCP/IP)
  - *(Un)wrapping* de mensagens de/em pacotes no nível do transporte
  - Envio / Recebimento de pacotes

IBM´s WebSphere MQ

TRANSFERÊNCIA DE MENSAGEM



- Canais são inerentemente unidirecionais
- Automaticamente iniciam MCA´s (message channel agent) quando uma mensagem chega
- Qualquer rede de gerenciadores de filas pode ser criada
- Roteadores são criados manualmente (administração do sistema)

MCA Message Channel Agent

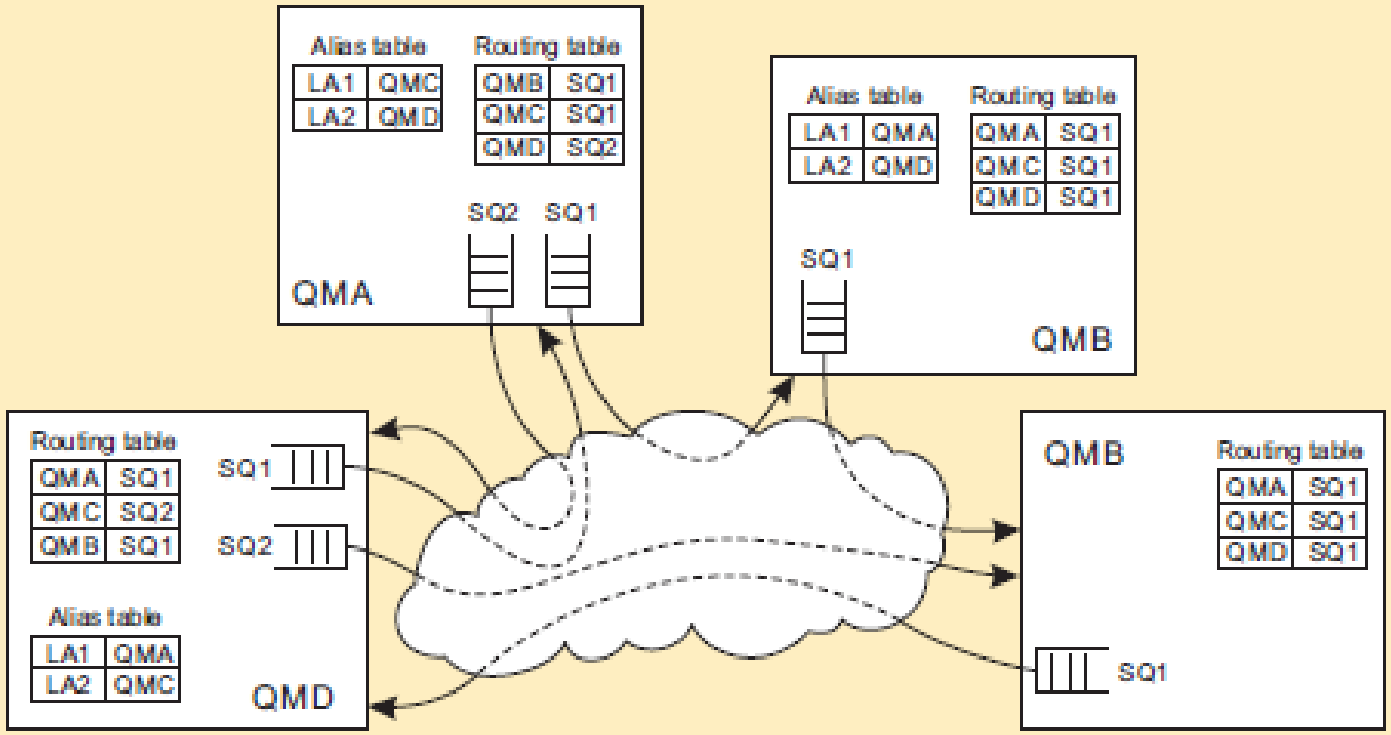
ALGUNS ATRIBUTOS ASSOCIADOS COM MCA

Attribute	Description
Transport type	Determines the transport protocol to be used
FIFO delivery	Indicates that messages are to be delivered in the order they are sent
Message length	Maximum length of a single message
Setup retry count	Specifies maximum number of retries to start up the remote MCA
Delivery retries	Maximum times MCA will try to put received message into queue

IBM´s WebSphere MQ

ROTEAMENTO

Usando nomes lógicos, em combinação com a resolução do nome para filas locais, é possível colocar uma mensagem em uma fila remota



MULTICASTING EM NÍVEL APLICAÇÃO

ESSÊNCIA

Organiza nós de um sistema distribuído em uma rede overlay e usa esta rede para disseminar dados:

- Muitas vezes uma árvore, levando a rotas únicas
- Alternativamente, também redes Mesh, que demanda uma forma de roteamento

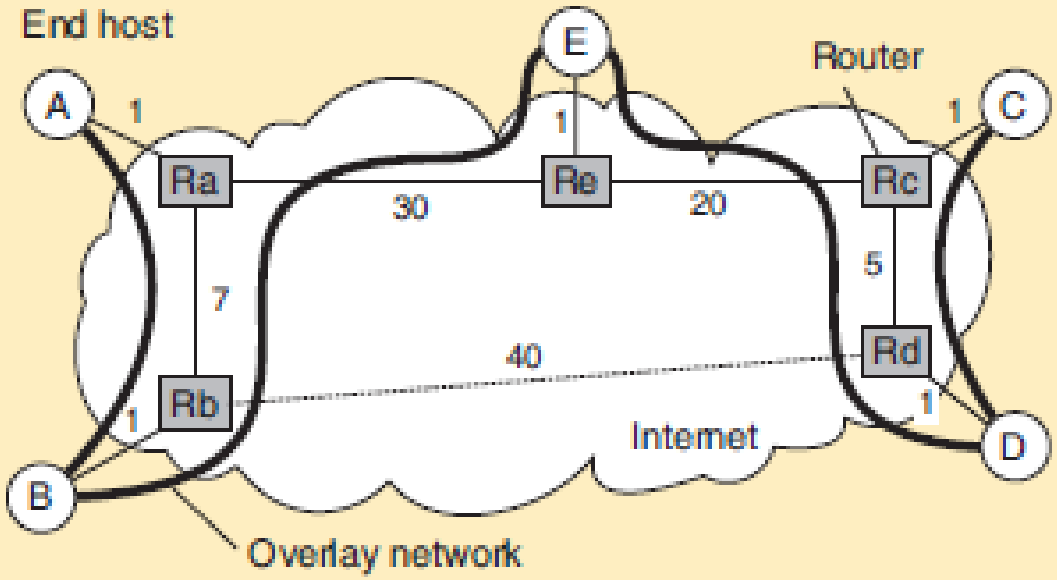
CHORD MULTICASTING EM NÍVEL APLICAÇÃO

ABORDAGEM BÁSICA

1. Iniciador gera um identificador multicast *mid*
2. Lookup *succ(mid)*, o nó responsável por *mid*
3. A requisição é roteada para *succ(mid),* que vai se tornar a root
4. Se *P* quer se juntar, então *P* envia uma requisição join para root
5. Quando a requisição chega em *Q*:
   - *Q* ainda não conhece uma requisição join → e se torna um forwarder; *P* se torna filho de *Q*. A requisição *join* continua a ser repassada.
   - *Q* sabe sobre a árvore → *P* se torna filho de *Q*. Não existe mais necessidade de repassar a requisição *join*.

CUSTOS DE MENSAGERIA EM NÍVEL APLICAÇÃO

DIFERENTES MÉTRICAS



- Stress no link: com qual frequência uma ALM (application level message) atravessa o mesmo link físico ? Exemplo: mensagem de *A* para *D* precisa atravessar <*Ra, Rb*> duas vezes.
- Stretch: Razão em atraso entre caminho ALM e caminho de rede. Exemplo: mensagens de *B* para *C* seguem um caminho de comprimento 73 em ALM, mas 47 em nível de rede → stretch = 73/47

COMUNICAÇÃO MULTICAST FLOODING

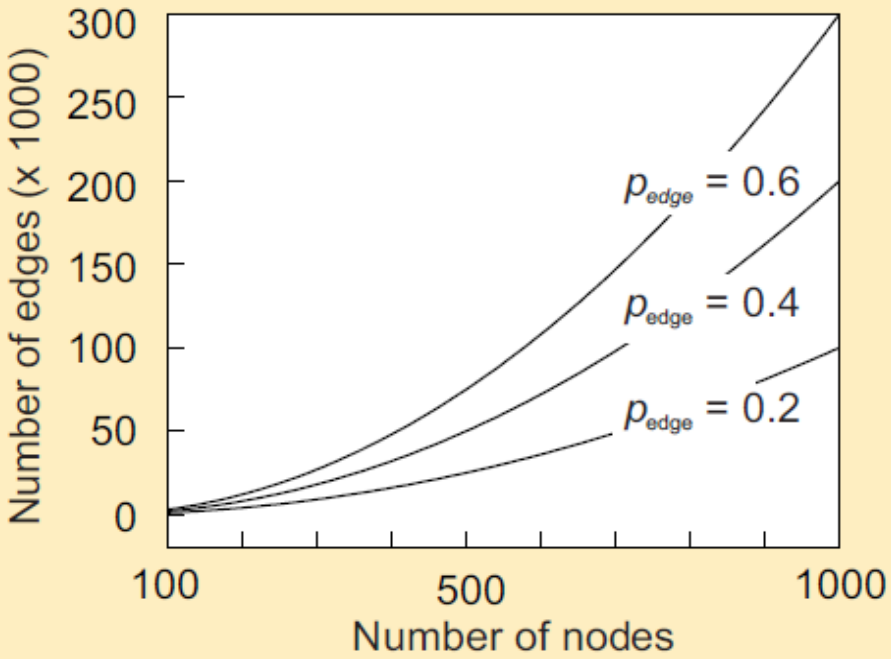
ESSÊNCIA

P envia mensagem *m* para todos seus vizinhos. Cada vizinho repassa esta mensagem, exceto para *P*, e somente se não viu a mesma mensagem *m* antes.

DESEMPENHO

Quanto mais bordas, mais custoso !

O TAMANHO DE OVERLAYS



VARIAÇÃO

Se *Q* repassar uma mensagem com uma certa probabilidade *Pflood*, possivelmente dependente de seu próprio número de vizinhos (i.é. node degree) ou o grau de seus vizinhos.

PROTOCOLOS EPIDÊMICOS

ASSUMA QUE NÃO EXISTE CONFLITO ESCRITA-ESCRITA

- Operações de atualização (update) são executadas em um único servidor
- Uma replica passa o estado da atualização para somente alguns vizinhos
- A propagação da atualização é preguiçosa, i.é., não imediata
- Finalmente cada atualização chega em cada réplica

DUAS FORMAS DE EPIDEMIA

- Anti-entropia: Cada replica regularmente escolhe outra replica de forma aleatória e troca as diferenças de estado, o que leva a estados idênticos em ambos depois da troca
- Espalhamento de rumor: Uma replica que foi recentemente atualizada (i.é., foi contaminada), conta para outras replicas sobre sua atualização (contaminando elas também)

ANTI ENTROPIA

PRINCIPAIS OPERAÇÕES

- Um nó *P* seleciona outro nó *Q* de um Sistema de forma aleatória
- Pull: *P* puxa somente dados novos de *Q* (de atualizações)
- Push: *P* somente empurra sua própria atualização para *Q*
- Push-Pull: *P* e *Q* enviam atualizações para ambos

OBSERVAÇÃO

- Para push-pull, demora $\mathcal{O}(log(N))$ rodadas para atualizar todos os N nós (rodada = quando cada nó tomou a iniciativa de iniciar uma troca)

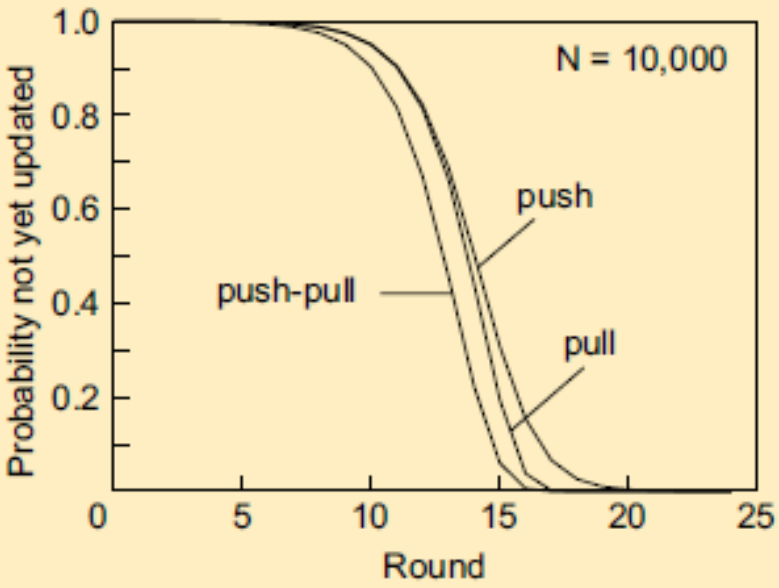
ANTI-ENTROPIA: ANÁLISE

BÁSICO

Considere uma única fonte propaganda suas atualizações. Faça *Pj* ser a probabilidade de um nó não ter recebido a atualização depois da *i*-enésima rodada.

ANÁLISE: SENDO IGNORANTE

- With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i^{th} round and should contact another ignorant node during the next round.
- With push, $p_{i+1} = p_i(1 - \frac{1}{N})^{N(1-p_i)} \approx p_i e^{-1}$ (for small p_i and large N): the node was ignorant during the i^{th} round and no updated node chooses to contact it during the next round.
- With push-pull: $(p_i)^2 \cdot (p_i e^{-1})$



ESPALHAMENTO DE RUMOR

MODELO BÁSICO

Um servidor *S* possuíndo uma atualização para reporter contato outros servidores. Se um servidor é contatado e o mesmo já recebeu uma atualização propaganda, S para de contatar outros servidores com probabilidade *Pstop*

OBSERVAÇÃO

Se S é uma fração de de servidores ignorantes (i.é, não sabem da atualização), pode ser mostrado que para muitos servidores

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

ANÁLISE FORMAL

NOTAÇÕES

Faça *S* denotar uma fração de nós que ainda não foram atualizados (i.é. Suscetível), i a fração de atualizados (infetados) e nós ativos; e *r* a fração de nós atualizados que desistíram (removidos)

DA TEORIA DA EPIDEMIA

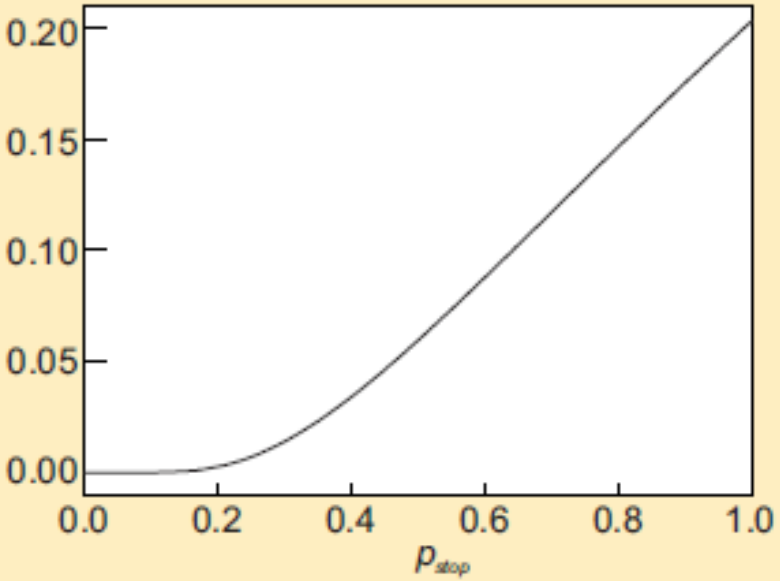
$$
\begin{aligned}
(1) \quad & ds/dt &=& \; -s \cdot i \\
(2) \quad & di/dt &=& \; s \cdot i - p_{stop} \cdot (1-s) \cdot i \\
\Rightarrow \quad & di/ds &=& \; -(1+p_{stop}) + \frac{p_{stop}}{s} \\
\Rightarrow \quad & i(s) &=& \; -(1+p_{stop}) \cdot s + p_{stop} \cdot \ln(s) + C
\end{aligned}
$$

RESUMINDO

$i(1) = 0 \Rightarrow C = 1 + p_{stop} \Rightarrow i(s) = (1 + p_{stop}) \cdot (1 - s) + p_{stop} \cdot \ln(s)$. We are looking for the case $i(s) = 0$, which leads to $s = e^{-(1/p_{stop}+1)(1-s)}$

ESPALHAMENTO DE RUMOR

O EFEITO DE PARAR



Consider 10,000 nodes		
$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

OBSERVAÇÃO

Se tivermos que garantir que todos servidores serão atualizados, somente o espalhamento de rumor não é sufuciente.

DELETANDO VALORES

PROBLEMA FUNDAMENTAL

Não podemos remover um valor antigo de um servidor e esperar a remoção se propagar. Ao invés disto, a mera remoção será desfeita no tempo devido usando algorítmos epidêmicos

SOLUÇÃO

A remoção tem que ser registrada como uma atualização especial através da inserção de um certificado de morte

DELETANDO VALORES

QUANDO REMOVER UM CERTIFICADO DE MORTE (NÃO É PERMITIDO FICAR PRA SEMPRE)

- Rode um algoritmo global para detetar se a remoção é conhecida em todos lugares, e então colete os certificados de morte (se parece com coleta de lixo)
- Assuma que certificados de morte se propagam em tempo finite, e associe um tempo de vida máximo para um certificado (pode ser feito mas com risco de não atingir todos servidores)

NOTA

É necessário que a remoção de fato atinja todos servidores