ACH 2147 — Desenvolvimento de Sistemas de Informação Distribuídos

Aula 10: Comunicação (parte 2)

Prof. Renan Alves

Escola de Artes, Ciências e Humanidades — EACH — USP

05/04/2024

Comunicação orientada a mensagens

RPC

- Modelo transiente e síncrono
- Pode não ser adequado para todos os cenários

Comunicação orientada a mensagens

- Alternativa ao modelo RPC
- Tornar o sistema assíncrono e persistente

Mensagens transientes: sockets

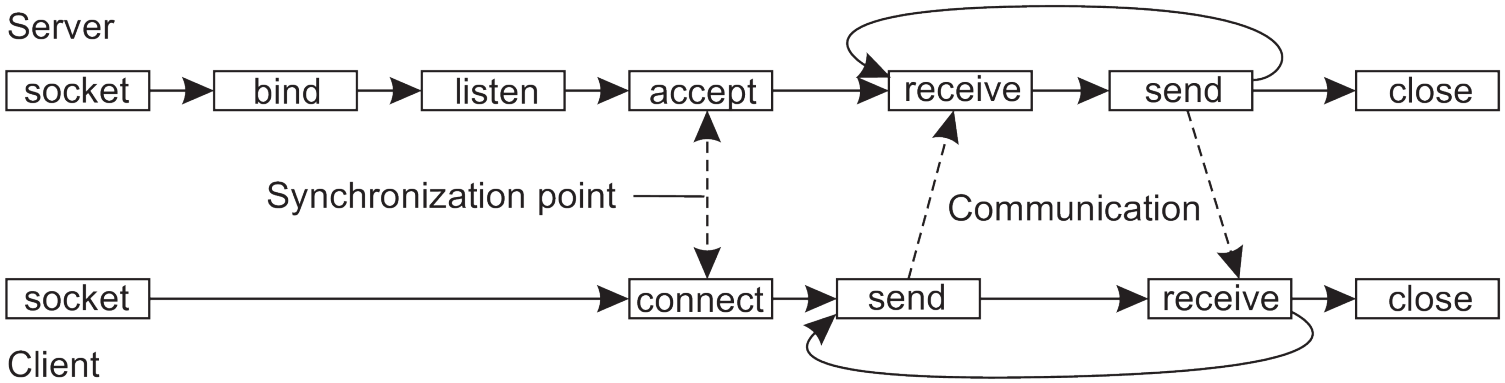
Sockets

- Vamos começar detalhando o funcionamento de uma interface transiente e síncrona
- Definição: terminal de comunicação através do qual é possível ler e escrever dados, que são enviados via rede
- Abstração da porta da camada de transporte

Interface socket POSIX

Operação	Descrição
`socket`	Cria um novo ponto de comunicação
`bind`	Associa um endereço a um socket
`listen`	Informa ao sistema operacional qual deve ser o número máximo de solicitações de conexão pendentes
`accept`	Bloqueia o chamador até que uma solicitação de conexão chegue
`connect`	Tenta ativamente estabelecer uma conexão
`send`	Envia alguns dados pela conexão
`receive`	Recebe alguns dados pela conexão
`close`	Libera a conexão

Interface socket POSIX: interação cliente-servidor



Sockets: Código em Python

Servidor

```
1  from socket  import *
2
3  class Server:
4    def run(self):
5      s = socket(AF_INET, SOCK_STREAM)
6      s.bind((HOST, PORT))
7      s.listen(1)
8      (conn, addr) = s.accept()  # returns new socket and addr. client
9      while True:                # forever
10       data = conn.recv(1024)   # receive data from client
11       if not data: break       # stop if client stopped
12       conn.send(data+b"*")     # return sent data plus an "*"
13     conn.close()               # close the connection
```

Cliente

```
1  class Client:
2    def run(self):
3      s = socket(AF_INET, SOCK_STREAM)
4      s.connect((HOST, PORT)) # connect to server (block until accepted)
5      s.send(b"Hello, world") # send same data
6      data = s.recv(1024)     # receive the response
7      print(data)             # print what you received
8      s.send(b"")             # tell the server to close
9      s.close()               # close the connection
```

Facilitando o uso de sockets

Observação

Os sockets são bastante de baixo nível e erros de programação são facilmente cometidos. No entanto, a maneira como são usados é muitas vezes a mesma (como em um ambiente cliente-servidor).

Alternativa: ZeroMQ

Fornece um nível mais alto de abstração ao parear sockets: um para enviar mensagens no processo P e um correspondente no processo Q para receber mensagens. Toda a comunicação é assíncrona.

Três padrões

- Requisição-resposta
- Publicação-assinatura (publish-subscribe)
- Pipeline

Requisição-resposta

- Funciona de forma parecido aos sockets tradicionais
- Servidor usa um socket do tipo REP (resposta)
- Cliente usa um socket to tipo REQ (requisição)
- Não há `listen` ou `accept` no servidor
- Não é necessário informar a quantidade de bytes a se receber
- Cliente pode ser inicializado antes do servidor!

Requisição-resposta

```
1  import zmq
2
3  def server():
4    context = zmq.Context()
5    socket  = context.socket(zmq.REP)        # create reply socket
6    socket.bind("tcp://*:12345")             # bind socket to address
7
8    while True:
9      message = socket.recv()                # wait for incoming message
10     if not "STOP" in str(message):         # if not to stop...
11       reply = str(message.decode())+'*'    # append "*" to message
12       socket.send(reply.encode())          # send it away (encoded)
13     else:
14       break                                # break out of loop and end
15
16 def client():
17   context = zmq.Context()
18   socket  = context.socket(zmq.REQ)        # create request socket
19
20   socket.connect("tcp://localhost:12345") # block until connected
21   socket.send(b"Hello world")             # send message
22   message = socket.recv()                 # block until response
23   socket.send(b"STOP")                    # tell server to stop
24   print(message.decode())                 # print result
```

Publish-subscribe

- Servidor usa um socket do tipo PUB (publish)
- Cliente usa um socket to tipo SUB (subscribe)
- Padrão de comunicação um-para-muitos
- Limita recebimento de mensagens dos clientes para o que foi pedido
- Não há inscrição padrão por parte do cliente
- Se não houver ninguém inscrito para receber, mensagens publicadas são perdidas

Publish-subscribe

```python
import multiprocessing
import zmq, time

def server():
  context = zmq.Context()
  socket = context.socket(zmq.PUB)           # create a publisher socket
  socket.bind("tcp://*:12345")               # bind socket to the address
  while True:
    time.sleep(5)                            # wait every 5 seconds
    t = "TIME " + time.asctime()
    socket.send(t.encode())                  # publish the current time

def client():
  context = zmq.Context()
  socket = context.socket(zmq.SUB)           # create a subscriber socket
  socket.connect("tcp://localhost:12345")    # connect to the server
  socket.setsockopt(zmq.SUBSCRIBE, b"TIME")  # subscribe to TIME messages

  for i in range(5):       # Five iterations
    time = socket.recv()   # receive a message related to subscription
    print(time.decode())   # print the result
```

Pipeline

- "Linha de produção"
- Um (conjunto de) processo(s) atua gerando resultados
- Um (conjunto de) processo(s) atua recebendo resultados
- O par transmissor-receptor formado não é importante (pode ser qualquer um, o primeiro disponível)
- Produtor usa um socket do tipo PUSH
- Consumidor usa um socket to tipo PULL

Pipeline

```
1  def producer():
2    context = zmq.Context()
3    socket  = context.socket(zmq.PUSH)        # create a push socket
4    socket.bind("tcp://127.0.0.1:12345")      # bind socket to address
5
6    while True:
7      workload = random.randint(1, 100)       # compute workload
8      socket.send(pickle.dumps(workload))     # send workload to worker
9      time.sleep(workload/NWORKERS)           # balance production by waiting
10
11 def worker(id):
12   context = zmq.Context()
13   socket  = context.socket(zmq.PULL)        # create a pull socket
14   socket.connect("tcp://localhost:12345")   # connect to the producer
15
16   while True:
17     work = pickle.loads(socket.recv())      # receive work from a source
18     time.sleep(work)                        # pretend to work
```

MPI

MessagePassing Interface

- Criado para unificar bibliotecas de alto desempenho proprietárias
- Busca ser mais flexível que usar sockets diretamente
- Assume grupo de processos conhecidos, cada grupo e cada processo com um com um ID
- Funciona como um middleware

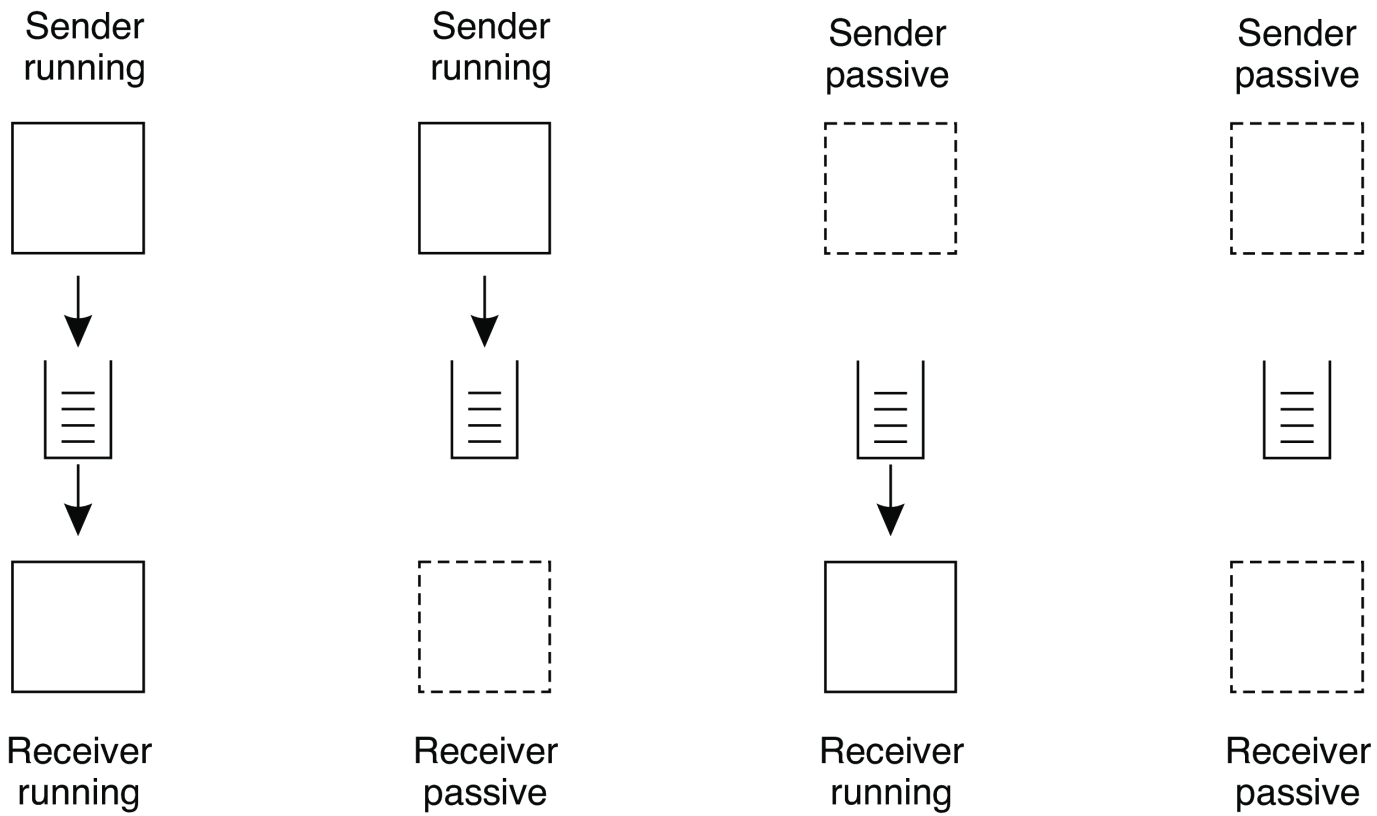
MPI

Operações representativas

Operação	Descrição
MPI_BSEND	Anexa uma mensagem de saída a um buffer de envio local
MPI_SEND	Envia uma mensagem e espera até que seja copiada para um buffer remoto
MPI_SSEND	Envia uma mensagem e espera até a transmissão começar
MPI_SENDRECV	Envia uma mensagem e espera pela resposta (como RPC)
MPI_ISEND	Passa referência para mensagem de saída e continua
MPI_ISSEND	Passa referência para mensagem de saída e espera até o recebimento começar
MPI_RECV	Recebe uma mensagem; bloqueia se não houver nenhuma
MPI_IRECV	Verifica se há uma mensagem de entrada, mas não bloqueia

Mensagens baseadas em filas

Quatro combinações possíveis



Sem garantias a respeito do atraso de entrega.

Middleware orientado a mensagens

Essência

Comunicação persistente assíncrona por meio do suporte a filas no nível de middleware. As filas correspondem a buffers nos servidores de comunicação.

Operações

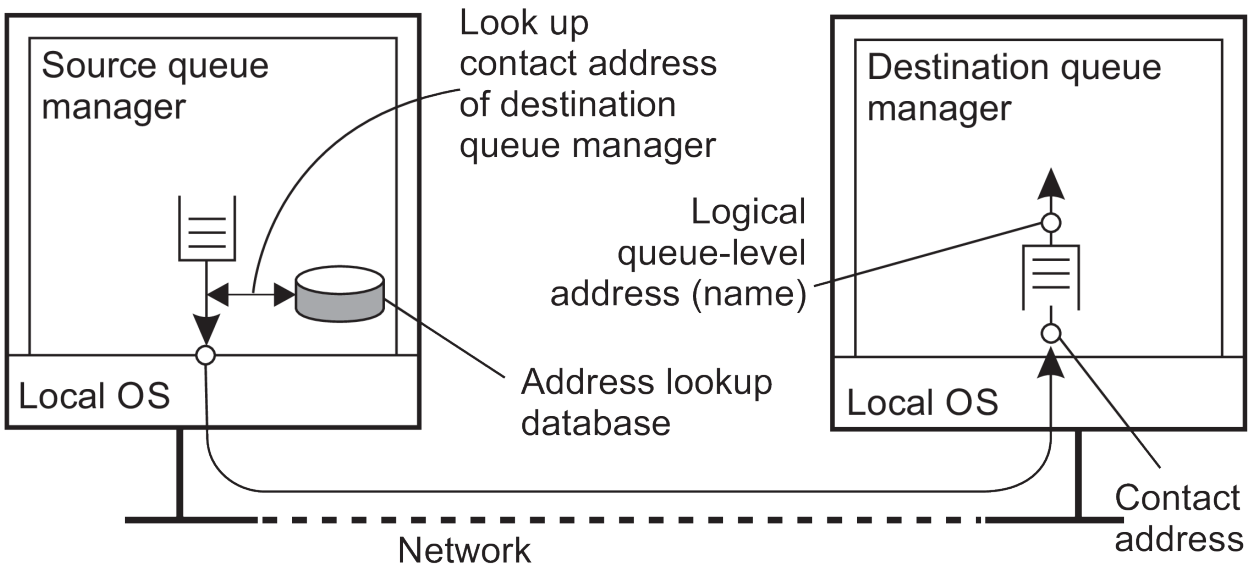
Operação	Descrição
`PUT`	Anexa uma mensagem a uma fila especificada
`GET`	Bloqueia até que a fila especificada não esteja vazia e remove a primeira mensagem
`POLL`	Verifica se fila especificada possui mensagens e remove a primeira. Nunca bloqueia
`NOTIFY`	Instala um handler para ser chamado quando uma mensagem é colocada na fila especificada

Modelo geral

Gerenciadores de filas

As filas são gerenciadas pelos gerenciadores de filas. Uma aplicação pode colocar mensagens apenas em filas locais. Receber uma mensagem somente é possível ao extraí-la de uma fila local $\Rightarrow$ os gerenciadores de filas precisam rotear mensagens.

Roteamento



Intermediário de mensagens (broker)

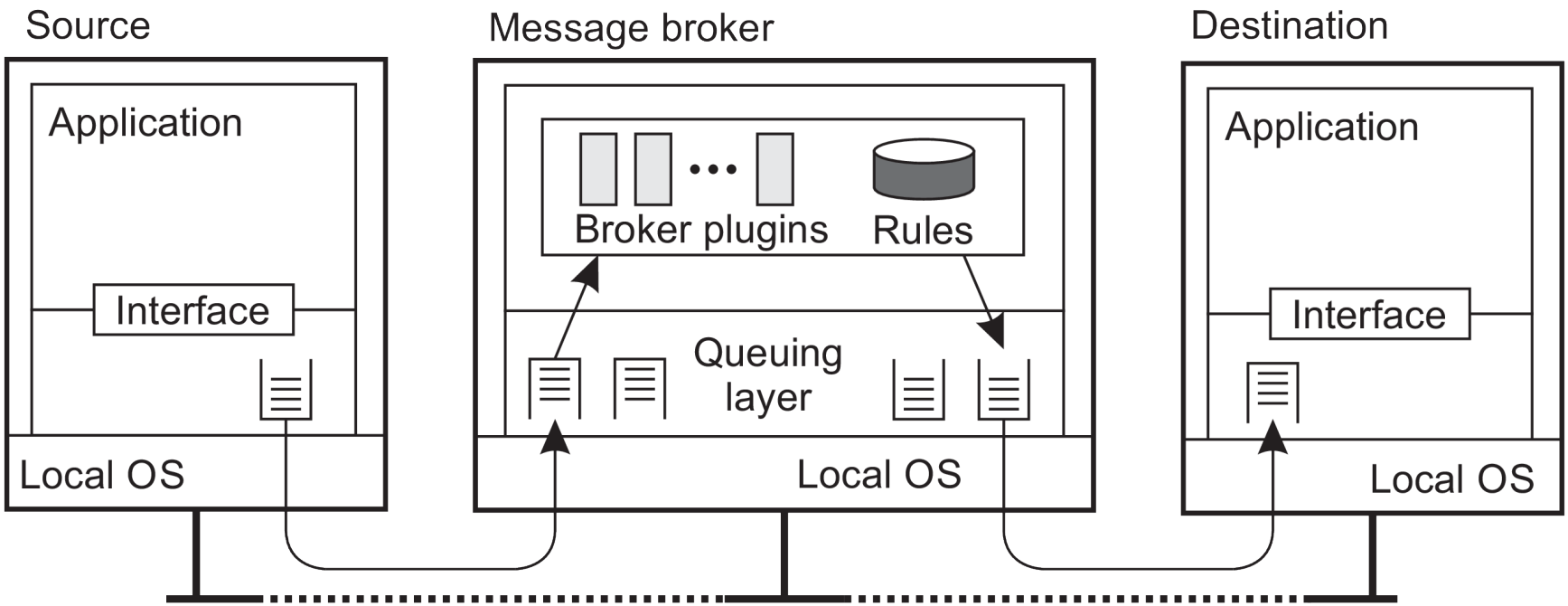
Observação

Sistemas de filas de mensagens pressupõem um protocolo de mensagens comum: todas as aplicações concordam com o formato da mensagem (ou seja, estrutura e representação de dados)

Broker lida com heterogeneidade de aplicativos em um sistema MQ

- Funciona como uma aplicação no sistema MQ (message queue)
- Transforma mensagens de entrada para o formato de destino
- Muito frequentemente atua como um gateway de aplicação
- Pode fornecer funcionalidade de roteamento baseado em assunto (ou seja, funcionalidade de publish-subscribe)

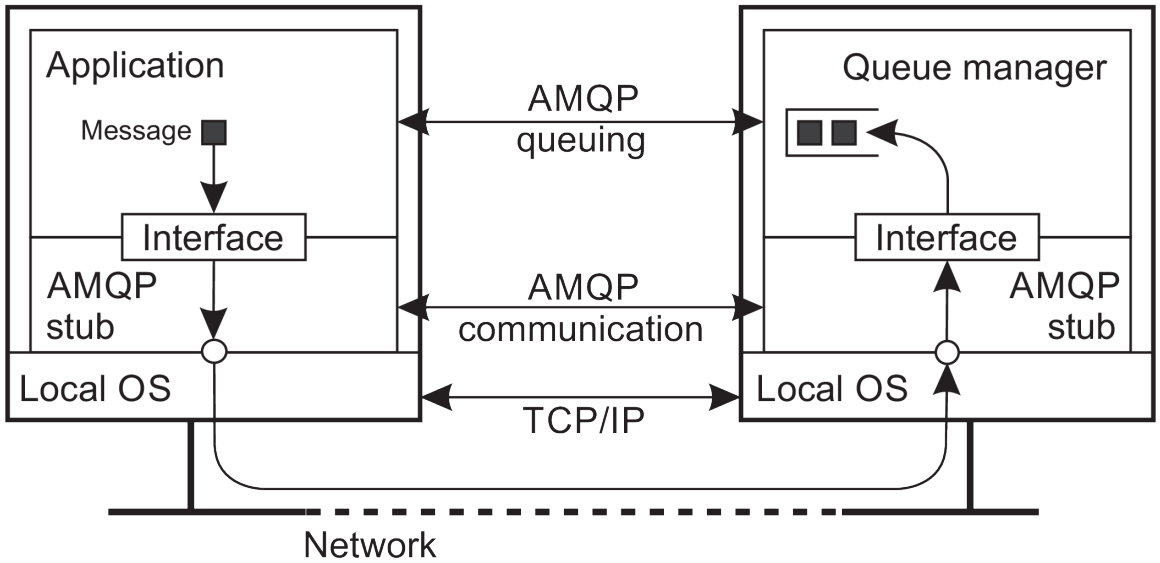
Arquitetura geral do broker



Exemplo: AMQP

Falta de padronização

O Advanced Message Queuing Protocol foi projetado para desempenhar o mesmo papel que, por exemplo, o TCP em redes: um protocolo para mensagens em alto nível com diferentes implementações.



Modelo básico

O cliente configura uma conexão (estável), que é um container para vários canais (possivelmente efêmeros) de uma via. Dois canais de uma via podem formar uma sessão. Um link é semelhante a um socket e mantém o estado sobre transferências de mensagem.

Exemplo: Produtor baseado em AMQP

```
1  import rabbitpy
2
3  def producer():
4    connection = rabbitpy.Connection() # Connect to RabbitMQ server
5    channel = connection.channel()     # Create new channel on the connection
6
7    exchange = rabbitpy.Exchange(channel, 'exchange') # Create an exchange
8    exchange.declare()
9
10   queue1 = rabbitpy.Queue(channel, 'example1') # Create 1st queue
11   queue1.declare()
12
13   queue2 = rabbitpy.Queue(channel, 'example2') # Create 2nd queue
14   queue2.declare()
15
16   queue1.bind(exchange, 'example-key') # Bind queue1 to a single key
17   queue2.bind(exchange, 'example-key') # Bind queue2 to the same key
18
19   message = rabbitpy.Message(channel, 'Test message')
20   message.publish(exchange, 'example-key') # Publish the message using the key
21   exchange.delete()
```

Exemplo: Consumidor baseado em AMQP

```python
import rabbitpy

def consumer():
  connection = rabbitpy.Connection()
  channel = connection.channel()

  queue = rabbitpy.Queue(channel, 'example1')

  # While there are messages in the queue, fetch them using Basic.Get
  while len(queue) > 0:
    message = queue.get()
    print('Message Q1: %s' % message.body.decode())
    message.ack()

  queue = rabbitpy.Queue(channel, 'example2')

  while len(queue) > 0:
    message = queue.get()
    print('Message Q2: %s' % message.body.decode())
    message.ack()
```