Sistemas Distribuídos

Capítulo 4 - Comunicação

Universidade Federal do Pampa
Campus Alegrete

Exercícios de fixação

1. Teria sentido limitar a quantidade de *threads* em um processo servidor?
2. Falamos sobre servidor de arquivo *multi-thread* mostrando porque ele é melhor que um servidor *monothread* e um servidor de máquina de estados. Há alguma circunstância na qual um servidor monothread poderia ser melhor? Dê um exemplo.
3. Construir um servidor concorrente por meio da multiplicação de um processo tem algumas vantagens e desvantagens em comparação com servidores *multithread*. Cite algumas.

Introdução
Cronograma

Semana	Terça	Quarta	Observação	Síncrono - Terça (teórica)	Períodos	Síncrono - Quarta (prática)	Períodos	Assíncrono	P
1	21/03	22/03		Apresentação da disciplina	2	Setup do ambiente e avaliação diagnóstica	2		
2	28/03	29/03		Introdução (Cap. 1)	2	artigo 1	2		
3	04/04	05/04		Arquiteturas distribuídas (Cap. 2)	2	discussão artigo 1 e escolha artigo 2	2		
4	11/04	12/04		Redes P2P	2	discussão artigo 2	2		
5	18/04	19/04		Processos (Cap. 3)	2	--	2	Computação em nuvem 1	
6	25/04	26/04		Comunicação (Cap. 4)	2	--	2	Computação em nuvem 2	
7	02/05	03/05		Comunicação Continuação (Cap. 4)	2	Revisão para avaliação 1	2		
8	09/05	10/05		Avaliação 1	2	Correção avaliação 1	2		
9	16/05	17/05		Coordenação (Cap. 5)	2	Kubernets 1	2		
10	23/05	24/05		Nomeação (Cap. 6)	2	Kubernets 2	2		
11	30/05	31/05		Consistência e replicação (Cap. 7)	2	Rabbitmq	2		
12	06/06	07/06		Tolerância a falhas p1 (Cap. 8)	2	Zookeeper 1	2		
13	13/06	14/06		Tolerância a falhas p2 (Cap. 8)	2	Zookeeper 2	2		
14	20/06	21/06		Segurança (Cap. 9)	2	Revisão para avaliação 2	2		
15	27/06	28/06		Avaliação 2	2	Correção avaliação 2	2		
16	04/07	05/07		Recuperação teórica	2	Recuperação prática	2		
17	11/07	12/07		Encerramento	2				
18									

Revisão

- Como os diferentes tipos de processos desempenham papéis em sistemas distribuídos?
  - *Threads*
    - Modelos e compromissos envolvidos
  - Virtualização
    - Máquina de processos vs Monitor de máquina de processos
  - Clientes
    - Qualidade de experiência, transparência
  - Servidores
    - Balanceamento de carga

Introdução

Objetivos

- Aprender conceitos fundamentais e mecanismos utilizados para comunicação em sistemas distribuídos

unipampa

Agenda

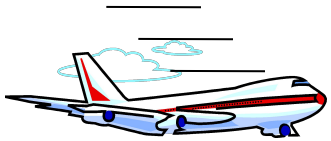
- ~~Introdução~~
- Comunicação
  - **Fundamentos**
  - Chamada de procedimento remoto
  - Comunicação orientada a mensagem
  - Comunicação orientada a fluxo
- Conclusão

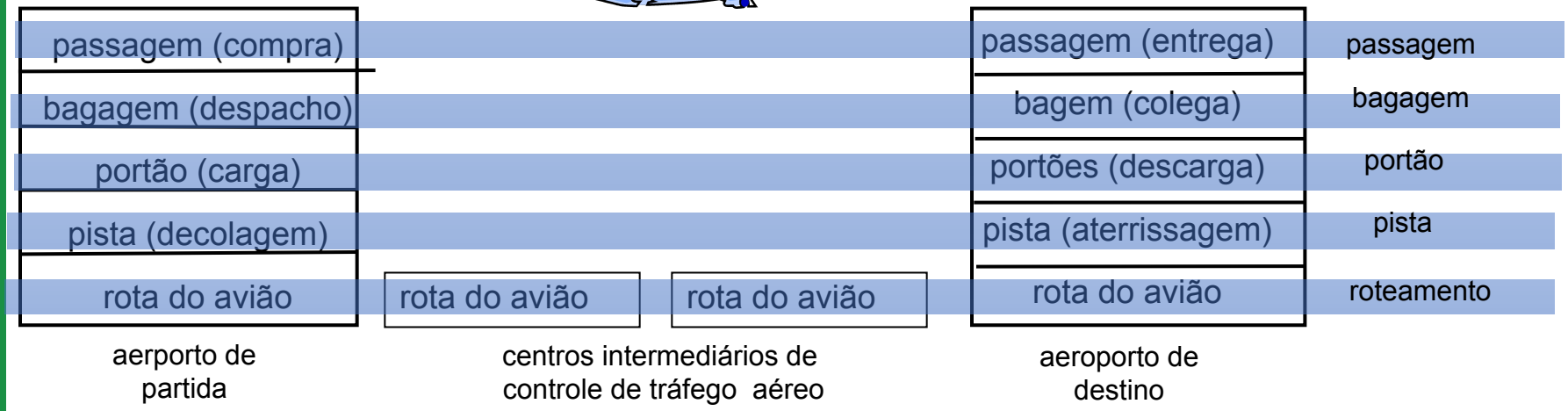
unipampa

Como lidar com sistemas complexos?

- Estrutura explícita permite identificação e relação entre partes complexas do sistema
- Modularização **facilita manutenção** e **atualização do sistema**
  - mudança de implementação do serviço da camada transparente ao restante do sistema
  - p.ex., mudanças em um procedimento específico não afeta o restante do sistema
- Exemplo?

Viagem de avião em "camadas"





camadas: cada uma implementa um serviço

- definindo ações internas
- baseada em serviços da camada inferior

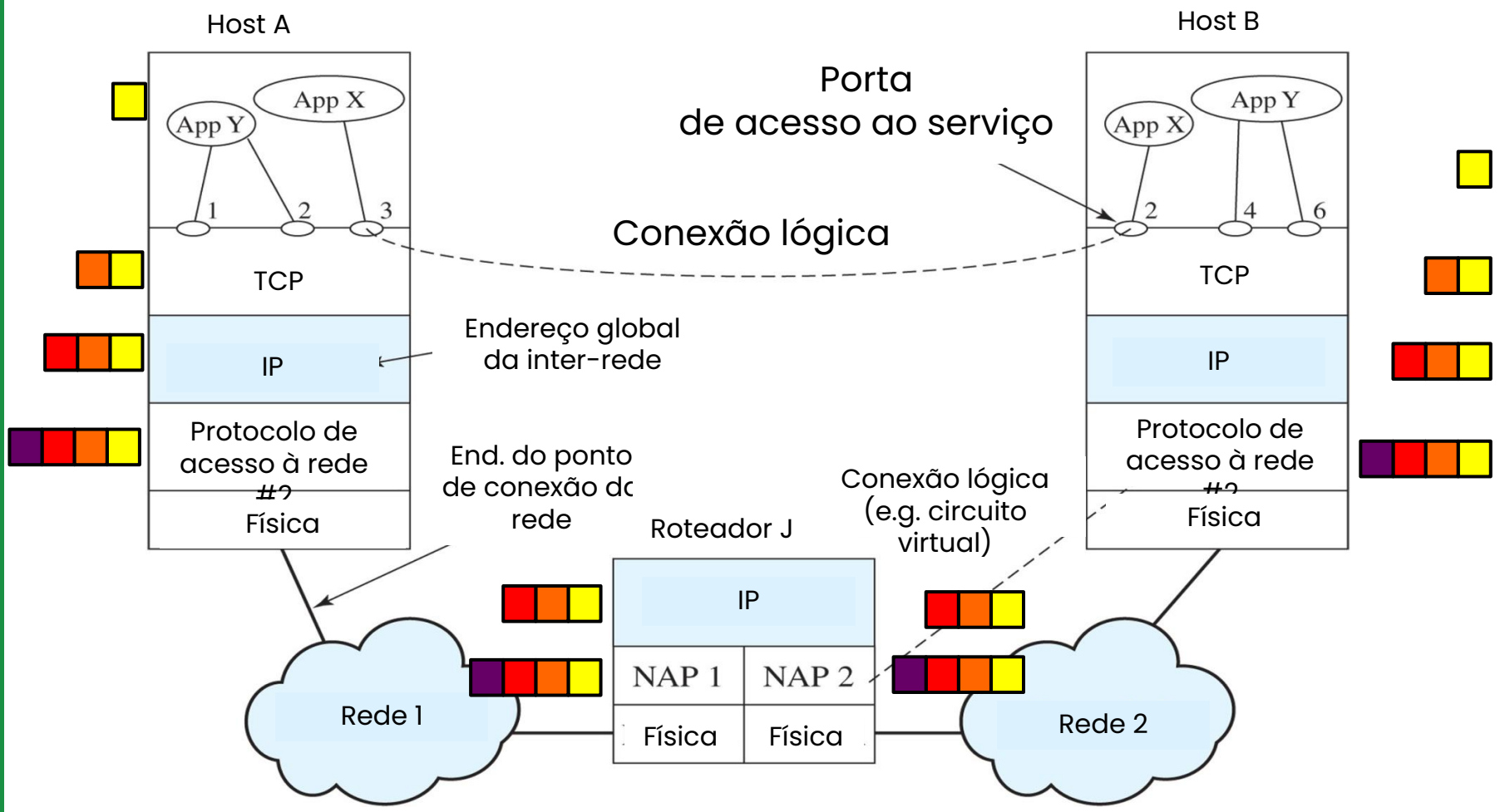
Pilha da Internet (5 camadas)

- *aplicação:* suporte p/ aplicações de rede
- *transporte:* transferência de dados entre processos
- *rede:* roteamento de datagramas da origem até o destino
- *enlace:* transferência de dados entre elementos de rede vizinhos
- *física:* bits transferidos no canal físico

- *Nasceu da reunião de protocolos*
  - *Originalmente, não definia camadas*
  - *Os protocolos foram organizados em camadas inspirados pelo modelo ISO/OSI*

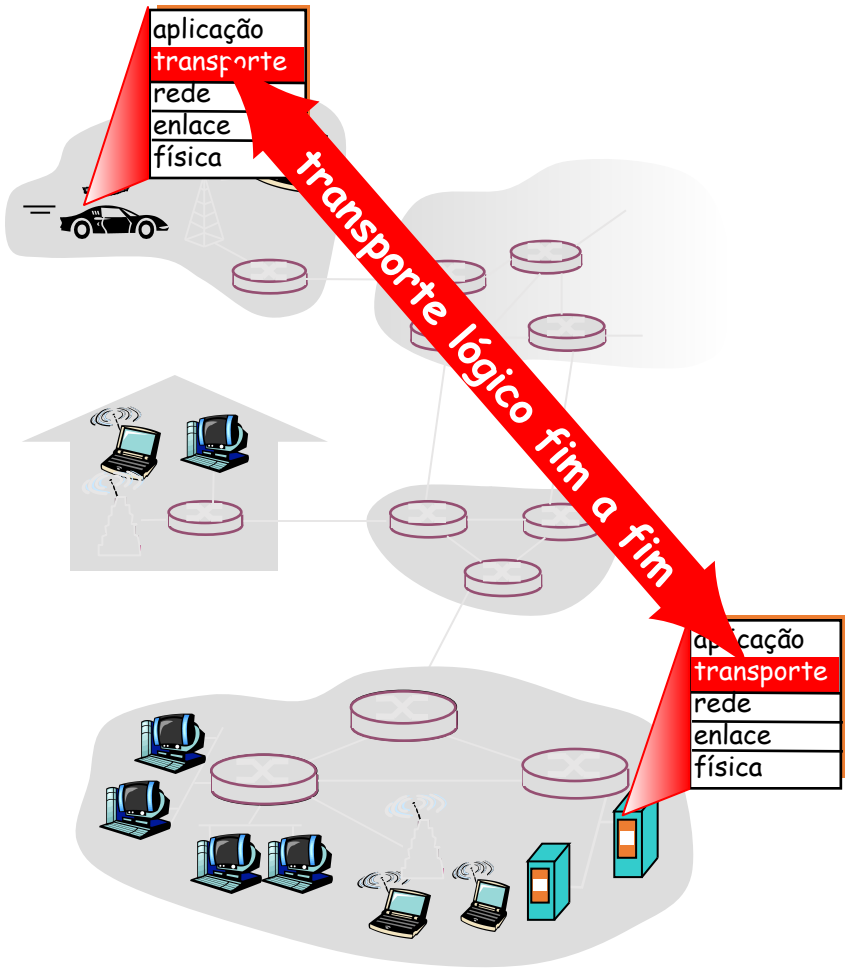
Fundamentos

Funcionamento da Pilha TCP/IP



Camada de rede vs. transporte

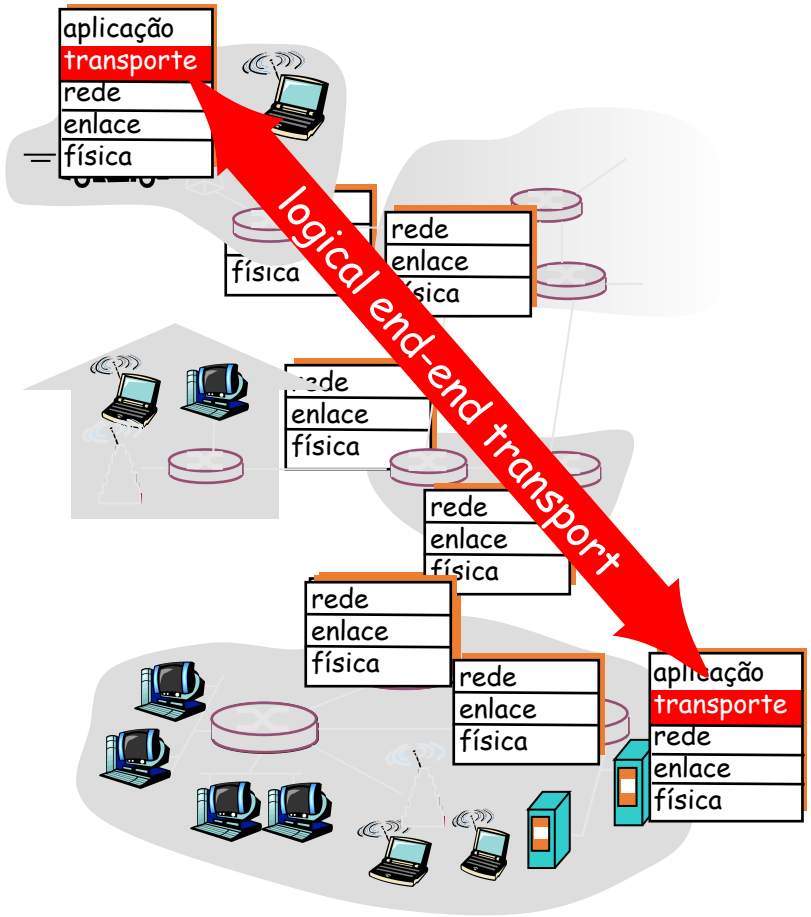
- *Camada de rede*: oferece comunicação lógica entre hosts
- *Camada de transporte*: oferece comunicação lógica entre processos
  - depende de, estende serviços da camada de rede
- Serviços da camada de transporte contam com e ampliam os serviços da camada de rede
- Existem dois protocolos de transporte na Internet
  - TCP e UDP



unipampa

Transporte: dois Protocolos

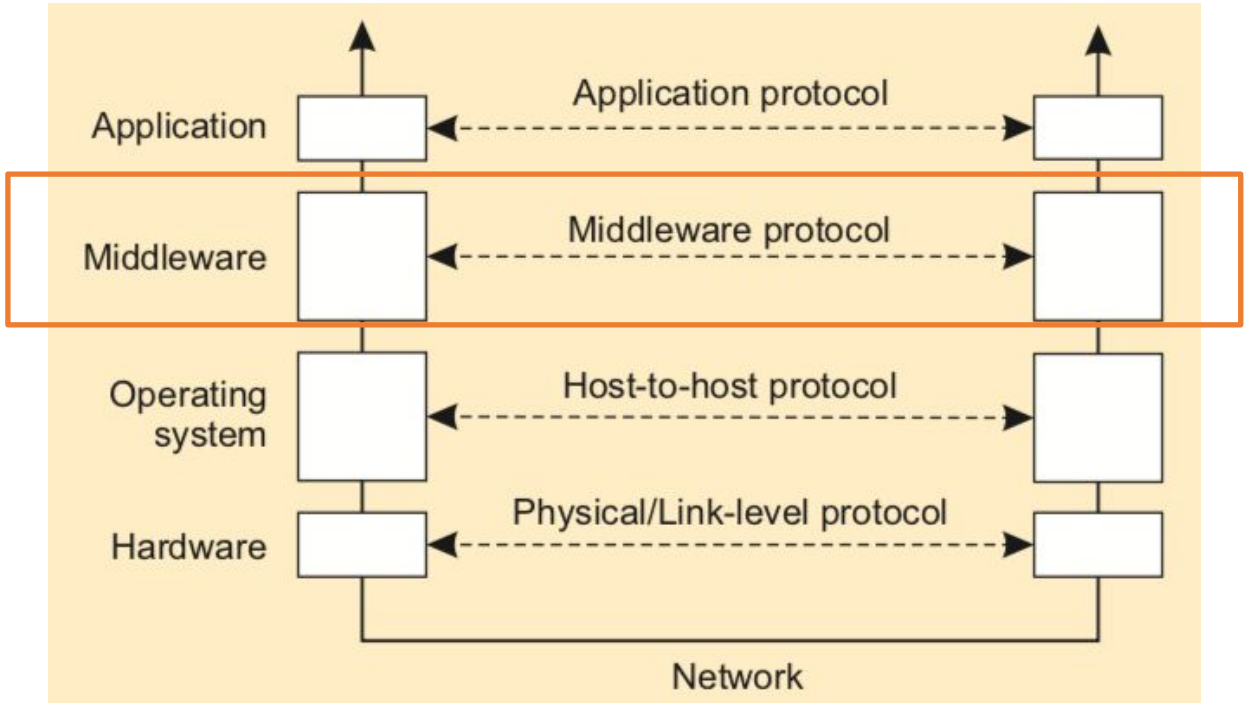
- TCP
  - *Transport Control Protocol*
  - Entrega *confiável e ordenada*
  - estabelecimento da conexão
  - controle de fluxo
  - controle de congestionamento
- UDP
  - *User Datagram Protocol*
  - Entrega *não confiável e sem garantia de ordem*
  - melhor esforço
- Serviços não disponíveis nativamente:
  - garantias de atraso máximo
  - garantias de largura de banda mínima



unipampa

Fundamentos
Middleware

- O middleware fornece serviços e protocolos comuns que podem ser usados por muitos aplicativos diferentes
- Exemplo: DNS (roda na camada de aplicação mas que serve a muitas outras aplicações)
- Nosso foco será nessas classes de serviços comuns



Fundamentos

Tipos de comunicação

- Duas propriedades principais
  1. Persistência
  2. Sincronismo
- Diversas combinações possíveis...

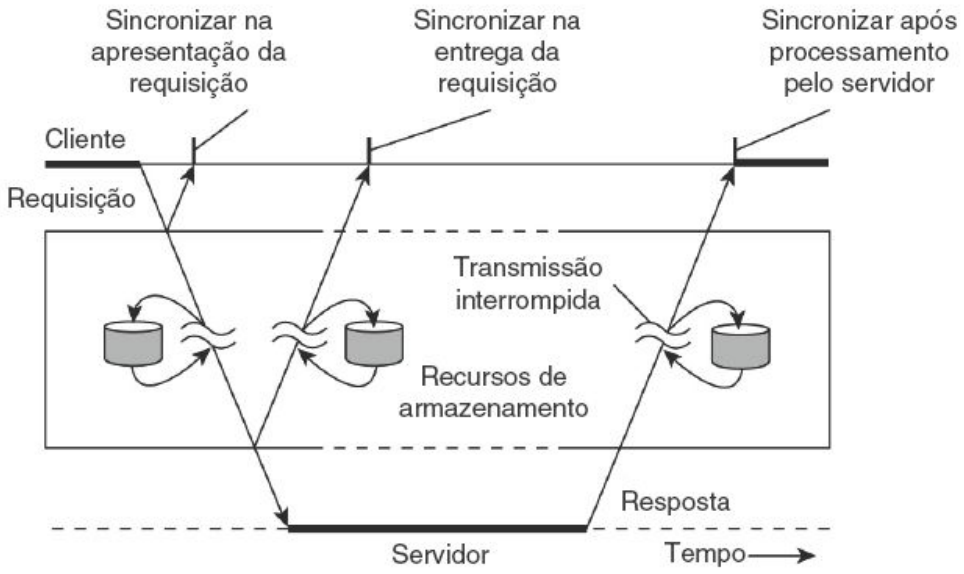


Figura 4.4 Middleware visto como serviço intermediário (distribuído) na comunicação de nível de aplicação.

unipampa

Propriedade 1: Persistência

- Ou a Comunicação é transiente
  - transmissão não é armazenada pelo middleware
  - persiste enquanto remetente e destinatário estiverem executando (exemplo: TCP)
- Ou a Comunicação é persistente
  - transmissão é armazenada pelo middleware (exemplo?)

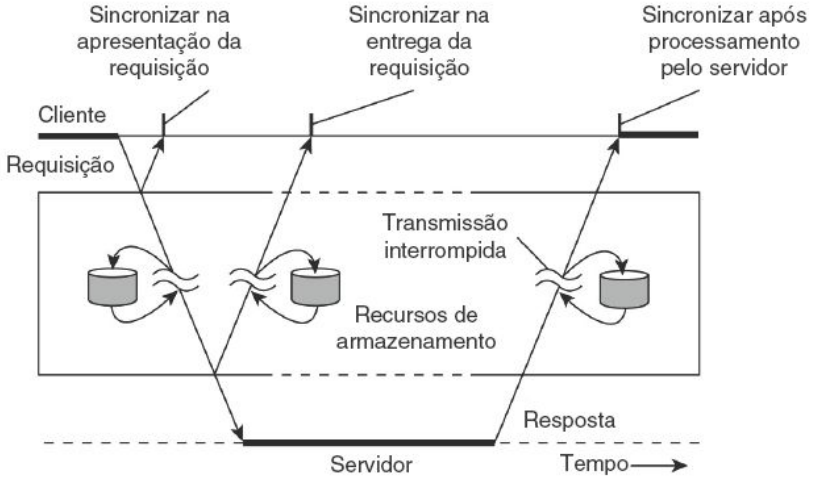


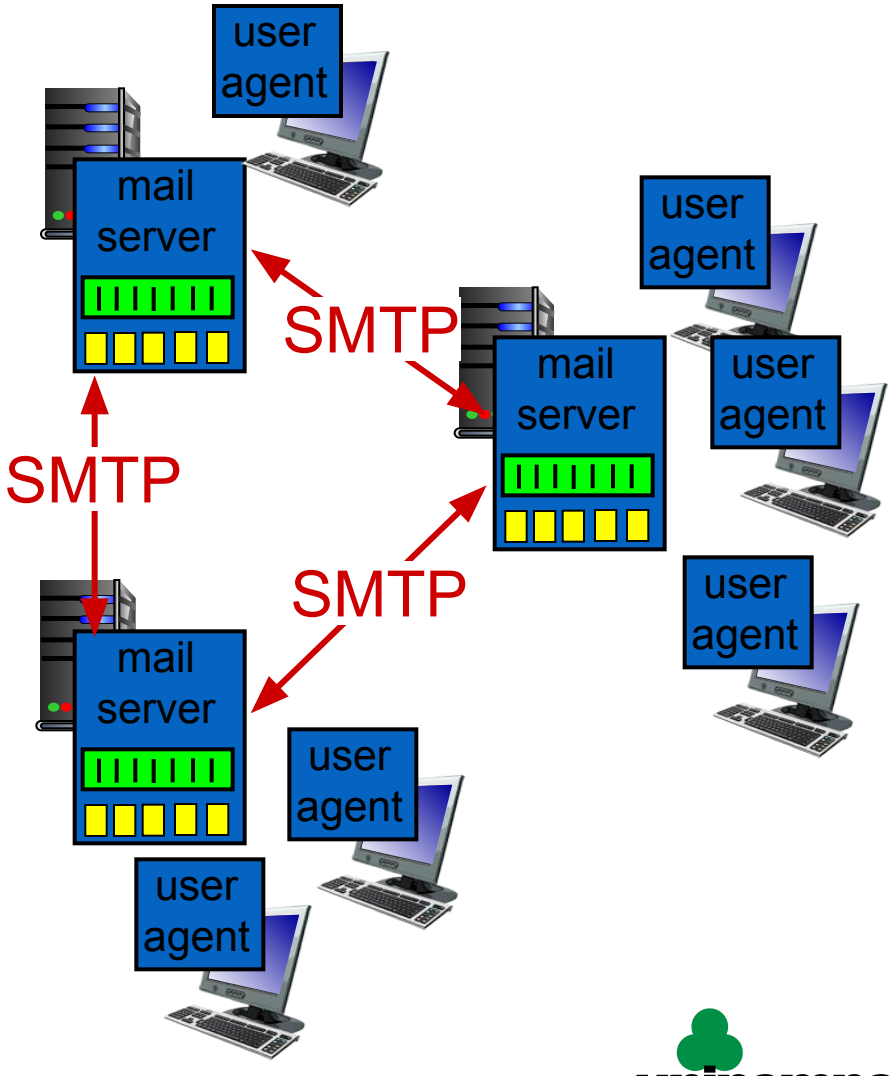
Figura 4.4 Middleware visto como serviço intermediário (distribuído) na comunicação de nível de aplicação.

unipampa

Comunicação persistente

Exemplo: E-mail

- Usuário remetente envia para servidor remetente
- Servidor remetente envia para servidor destinatário
- Usuário destinatário lê de servidor destinatário
- *Servidor de e-mail:*
  - *Possui um mailbox* (caixa de e-mail) armazena mensagens que chegam para usuário
  - *Possui uma fila de mensagens* de saída (a serem enviadas)
  - *Usa SMTP* para trocar mensagens de e-mail



unipampa

Fundamentos – tipos de comunicação
Propriedade 2: Sincronismo

- Ou Assíncrono: remetente continua sua execução imediatamente após enviar mensagem
- Ou Síncrono: remetente é bloqueado até saber que sua requisição foi aceita

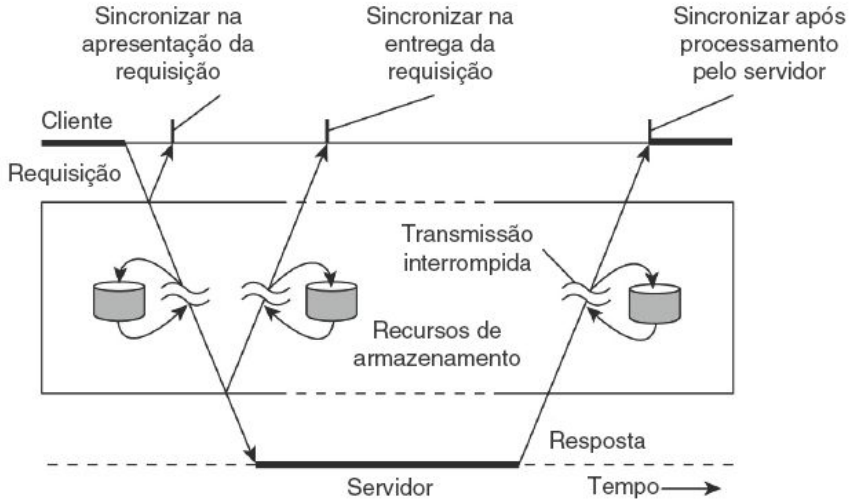


Figura 4.4 Middleware visto como serviço intermediário (distribuído) na comunicação de nível de aplicação.

Exemplo: comunicação síncrona com Sockets

Server

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

Client

```
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 same data
5  data = s.recv(1024)     # receive the response
6  print data              # print the result
7  s.close()               # close the connection
```

Agenda

- ~~Introdução~~
- Comunicação
  - Fundamentos
  - **Chamada de procedimento remoto**
  - Comunicação orientada a mensagem
  - Comunicação orientada a fluxo
- Conclusão

unipampa

Chamada de procedimento remoto
Motivação

- Como chamar um procedimento?
  - Localmente, pilha de execução, passagem por parâmetros ou por referência? A aplicação precisa saber detalhes sobre como cada processo é feito?
  - Remotamente? Passagem por referência?

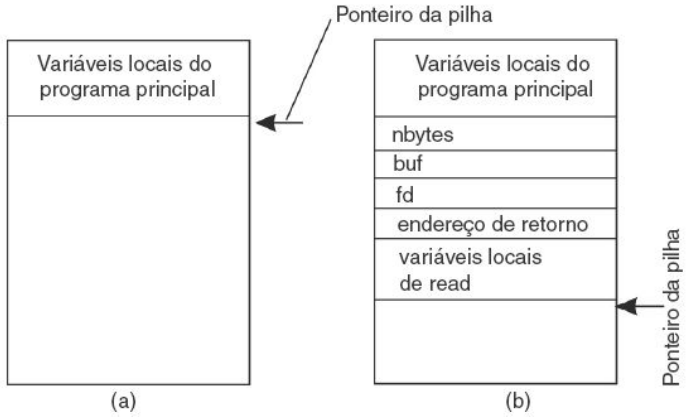


Figura 4.5 (a) Passagem de parâmetros em uma chamada de procedimento local: a pilha antes da chamada read. (b) A pilha enquanto o procedimento chamado está ativo.

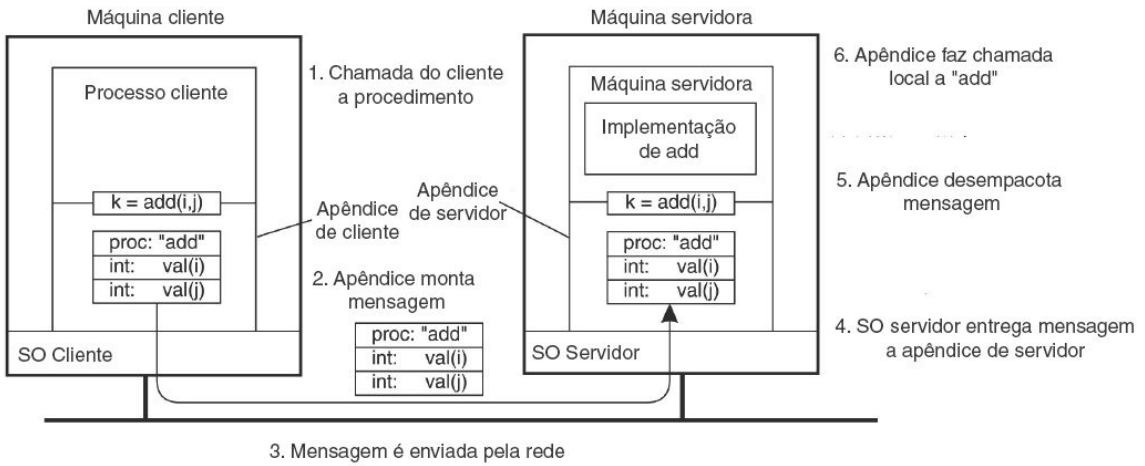


Figura 4.7 Etapas envolvidas para fazer um cálculo remoto por meio de RPC.

Chamada de procedimento remoto

Passagem de parâmetro

- Mais do que apenas encapsular parâmetros em mensagem:
  - Máquinas cliente e servidor podem ter diferentes representações de dados (pense em ordenação de bytes - Little Endian (e.g. Pentium) X Big Endian (e.g. SPARC)
  - Encapsular um parâmetro significa transformar um valor em uma sequência de bytes

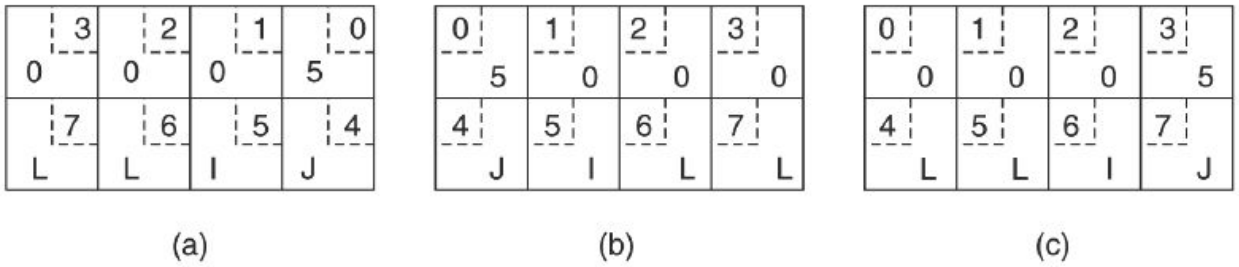


Figura 4.8 (a) Mensagem original no Pentium. (b) Mensagem após recebimento na SPARC. (c) Mensagem após ser invertida. Os pequenos números nos quadrados indicam o endereço de cada byte.

unipampa

Chamada de procedimento remoto
Passagem de parâmetro

- O cliente e o servidor precisam concordar com a mesma codificação:
  - Como os valores dos dados básicos são representados (números inteiros, flutuantes, caracteres)
  - Como os valores de dados complexos são representados (*arrays*, uniões)
- Enfim..
  - Cliente e servidor precisam interpretar corretamente as mensagens, transformando-as em representações dependentes da máquina!

Figura 4.9 (a) Procedimento. (b) Mensagem correspondente.

Chamada de procedimento remoto
Comunicação

- *Remote Process Call (RPC)*
- Permite que programas chamem procedimentos localizados em outras máquinas
- Oferece transparência de comunicação (em contraste com primitivas *send/request*)

Figura 4.6 Princípio de RPC entre um programa cliente e um programa servidor.

unipampa

Chamada de procedimento remoto
Chamada assíncrona

- Para situações em que não há resultado para retornar ao cliente, RPC podem permitir chamadas assíncronos.
- Com RPCs assíncronos, o servidor envia imediatamente uma resposta de volta ao cliente no momento em que a solicitação de RPC é recebida, e depois chama localmente o procedimento solicitado
- Resposta atua como um reconhecimento ao cliente de que o servidor processará o RPC
- Cliente continuará sem mais bloqueios assim que receber a confirmação do servidor

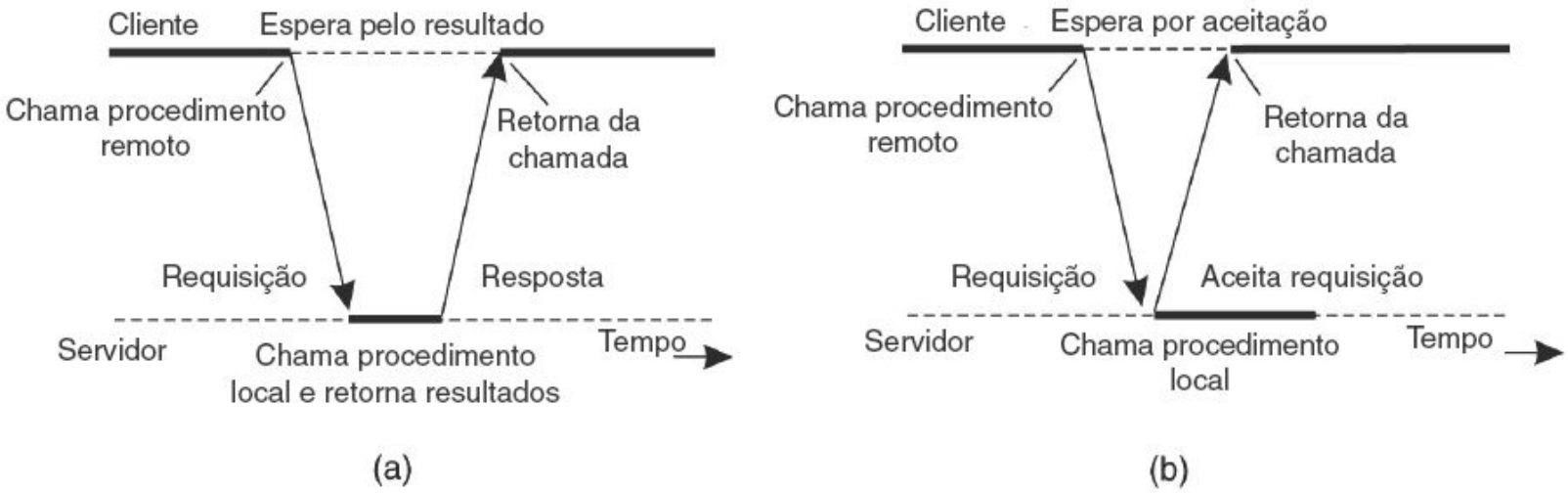


Figura 4.10 (a) Interação entre cliente e servidor em uma RPC tradicional. (b) Interação que usa RPC assíncrona.

Chamada de procedimento remoto

Chamada assíncrona no cliente

- RPCs assíncronos também podem ser úteis quando uma resposta será retornada, mas o cliente não está preparado para aguardar bloqueado
- Ex: quando o cliente precisa entrar em contato com vários servidores de forma independente. Após enviar todas as solicitações, o cliente pode começar a esperar que os vários resultados sejam retornados.
- Para esses casos, pode ser aplicado o chamado RPC síncrona adiada
  - Cliente chama o servidor, aguarda a aceitação e continua.
  - Quando os resultados ficam disponíveis, o servidor envia uma mensagem de resposta que leva a um retorno de chamada no lado do cliente.

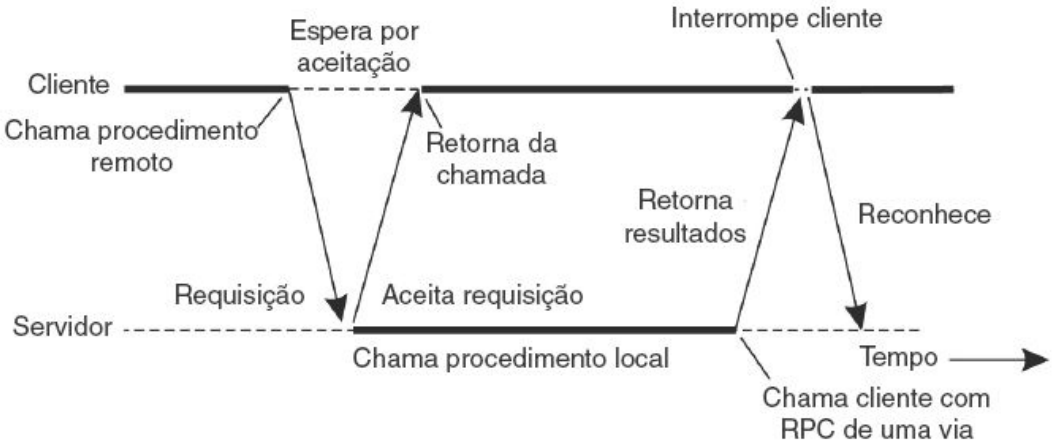
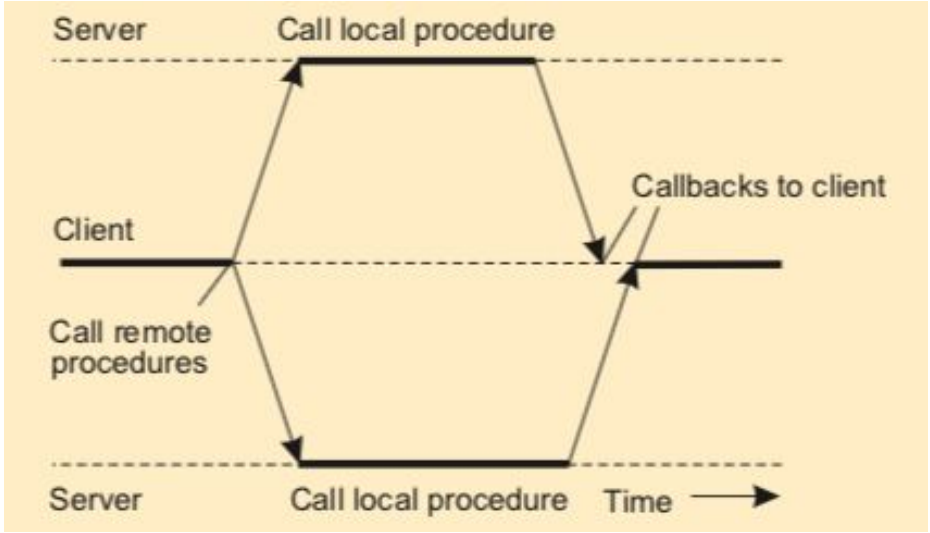


Figura 4.11 Cliente e servidor que interagem por meio de duas RPCs assíncronas.

Chamada de procedimento remoto

Chamada para múltiplos servidores

- Cliente pode não estar ciente do fato de que um RPC está sendo encaminhado para mais de um servidor
  - Para aumentar a tolerância a falhas, podemos decidir que todas as operações sejam executadas por um servidor de backup que possa assumir o controle quando o servidor principal falhar.
  - Cliente pode não estar ciente que um servidor foi replicado, por exemplo, porque estamos usando um endereço multicast no nível de transporte.
- O que fazer com as respostas? Cliente continuará após todas as respostas terem sido recebidas ou aguardará apenas uma?
  - Caso de tolerância a falhas, podemos decidir esperar apenas pela primeira resposta, ou talvez até que a maioria dos servidores retorne o mesmo resultado.
  - Caso de escalabilidade, os resultados podem precisar ser mesclados antes que o cliente possa continuar.



Chamada de procedimento remoto
Funcionamento na prática

- *Comunicação através de um middleware*

1. O procedimento do cliente chama o stub do cliente da maneira normal
2. O stub do cliente constrói uma mensagem e chama o sistema operacional local
3. O sistema operacional do cliente envia a mensagem ao sistema operacional remoto
4. O sistema operacional remoto envia a mensagem ao *stub* do servidor
5. O stub do servidor desempacota o (s) parâmetro(s) e chama o servidor
6. O servidor faz o trabalho e retorna o resultado para o *stub*
7. O stub do servidor empacota o resultado em uma mensagem e chama seu SO
8. O sistema operacional do servidor envia a mensagem ao sistema operacional do cliente
9. O sistema operacional do cliente envia a mensagem ao stub do cliente
10. O stub desempacota o resultado e o retorna ao cliente

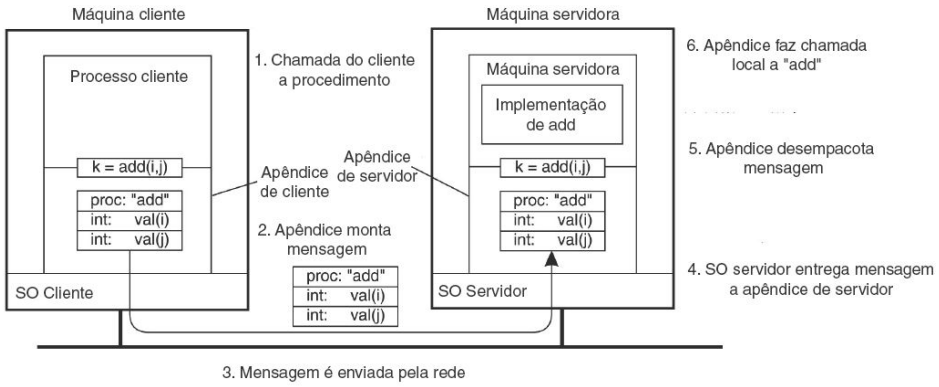


Figura 4.7 Etapas envolvidas para fazer um cálculo remoto por meio de RPC.

Chamada de procedimento remoto
Java RMI – exemplo

- *# Remote Method Invocation*
  http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html
- # Baixar arquivo compactado e descompactar
- # Acessar VM

```
$ vagrant ssh
```

- # Instalar JDK

```
vagrant@ubuntu-bionic:~$ sudo apt-get install default-jdk
```

- # Acessar VM e diretório, ex:

```
vagrant@ubuntu-bionic:~$ cd classroom-project/Cap_04_Comunicacao/01_Java_RMI_Hello
```

- # Compilar arquivos
- $ javac example/hello/*.java
- # Iniciar o Java RMI registry

```
$ rmiregistry
```

- # Em outro terminal, executar servidor

```
$ java example.hello.Server
```

- # Em outro terminal, executar Cliente

```
$ java example.hello.Client
```

unipampa

Chamada de procedimento remoto

Python RPC - exemplo

- # Baixar arquivo Cap_04_Comunicacao.zip e descompactar
- # Acessar VM

```
$ vagrant ssh
```

- # Instalar dependências

```
vagrant@ubuntu-bionic:~$ pip install rpyc
```

- # Acessar VM e diretório, ex:

```
vagrant@ubuntu-bionic:~$ cd
classroom-project/Cap_04_Comunicacao/02_Python_RPC_fig12
```

- # Executar servidor

```
$ python server.py
```

- # Executar Cliente

```
$ python client.py
```

unipampa

Agenda

- ~~Introdução~~
- Comunicação
  - Fundamentos
  - Chamada de procedimento remoto
  - **Comunicação orientada a mensagem**
  - Comunicação orientada a fluxo
- Conclusão

unipampa

Comunicação orientada a mensagem
Retomando...

- Comunicação por socket
  - Simples (send & receive)
  - Um para um
  - Inapropriado para sistemas de alto desempenho

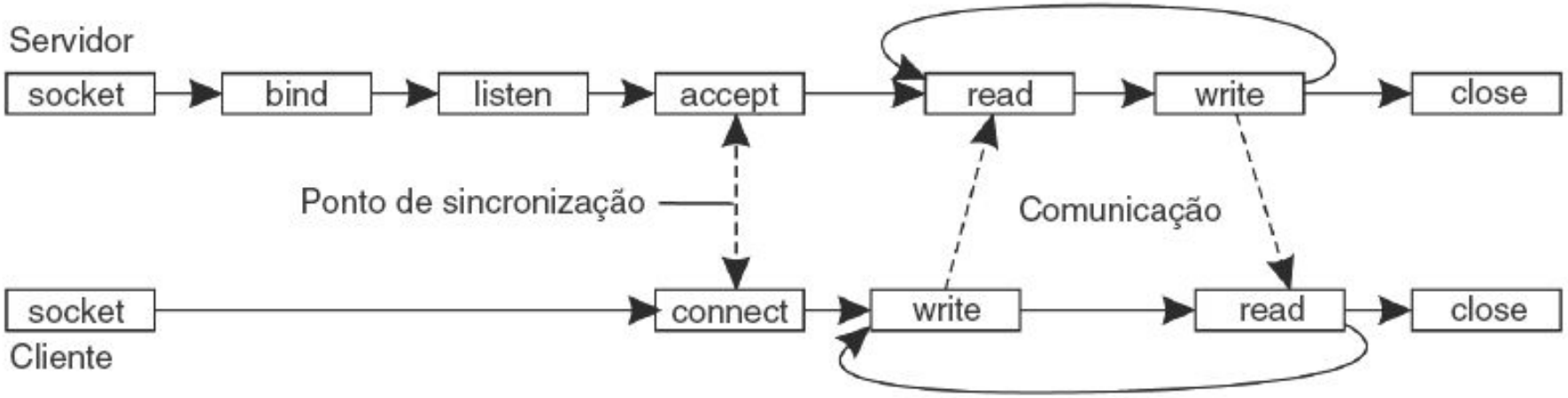


Figura 4.14 Padrão de comunicação orientada a conexão que usa interface Sockets.

Comunicação orientada a mensagem

Motivação

- RPC tem natureza síncrona (envia, aguarda)
- P.: alternativa?
- R.: troca de mensagens!
  1. Não persistente - transiente
  2. Persistente

unipampa

Comunicação orientada a mensagem

Comunicação transiente

- MPI
  - *Message Passing Interface*
  - Projetada para aplicações paralelas (local) e comunicação transiente (distribuída)
  - Exemplo local:
    - http://www.paulocollares.com.br/2013/05/ola-mundo-com-mip-em-c/
  - Exemplo em rede:
    - http://mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan/
- ZeroMQ
  - http://zeromq.org/
  - Um-para-muitos
  - Muitos-para-um
  - Middleware roda sobre TCP (1-para-1)

unipampa

Comunicação orientada a mensagem
ZMQ: Request-reply

Server

```
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
```

Client

```
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
```

unipampa

Comunicação orientada a mensagem

ZMQ – exemplo Request-Reply

- *# Figura 12*
- # Baixar arquivo compactado e descompactar
- # Acessar VM

```
vagrant ssh
```

- # Instalar dependecias

```
vagrant@ubuntu-bionic:~$ pip install zmq
```

- # Acessar VM e diretório, ex:

```
vagrant@ubuntu-bionic:~$ cd classroom-project/Cap_04_Comunicacao/03_Python_ZMQ_fig22
```

- # Executar servidor

```
$ python server.py
```

- # Executar Cliente

```
$ python client.py
```

unipampa

Comunicação orientada a mensagem
ZMQ: Publish-subscribe

Server

```
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
```

Client

```
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
```

Comunicação orientada a mensagem

ZMQ – exemplo Publish-Subscribe

- *# Figura 12*
- # Baixar arquivo compactado e descompactar
- # Acessar VM

```
vagrant ssh
```

- # Instalar dependecias

```
vagrant@ubuntu-bionic:~$ pip install zmq
```

- # Acessar VM e diretório, ex:

```
vagrant@ubuntu-bionic:~$ cd
classroom-project/Cap_04_Comunicacao/04_Python_ZMQ_fig23
```

- # Executar servidor

```
$ python server.py
```

- # Executar Cliente

```
$ python client.py
```

unipampa

Comunicação orientada a mensagem
ZeroMQ: 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

```
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
```

- Pipeline, processo que empurra não se importa qual outro processo puxa seus resultados: o primeiro disponível é OK.
- Da mesma forma, qualquer processo obtendo resultados de vários outros processos fará isso a partir do primeiro processo de envio, disponibilizando seus resultados.
- A intenção é manter o maior número possível de processos trabalhando

unipampa

Chamada de procedimento remoto

Python ZMQ – exemplo Pipeline

- *# Figura 12*
- # Baixar arquivo compactado e descompactar
- # Acessar VM

```
vagrant ssh
```

- # Instalar dependecias

```
vagrant@ubuntu-bionic:~$ pip install zmq
```

- # Acessar VM e diretório, ex:

```
vagrant@ubuntu-bionic:~$ cd classroom-project/Cap_04_Comunicacao/05_Python_ZMQ_fig24
```

- # Executar Source

```
$ python tasksrc.py a
```

- # Executar Worker

```
$ python taskwork.py a
```

unipampa

Comunicação orientada a mensagem

Comunicação persistente

- Visam suportar transferências de mensagens com duração na ordem de minutos (em contraste, RPC e MPI visam ordem de segundos)
- Filas correspondem a buffers em servidores
- Modelo de enfileiramento de mensagens
  - Aplicações se comunicam inserindo mensagens em filas específicas
  - Mensagens são repassadas por uma série de servidores até serem entregues
  - Garantia apenas sobre inserção na fila (nenhuma sobre quando e nem ao menos se a mensagem será entregue)
  - Comunicação "fracamente acoplada" em relação ao tempo (remetente e receptor podem rodar de maneira completamente independente)

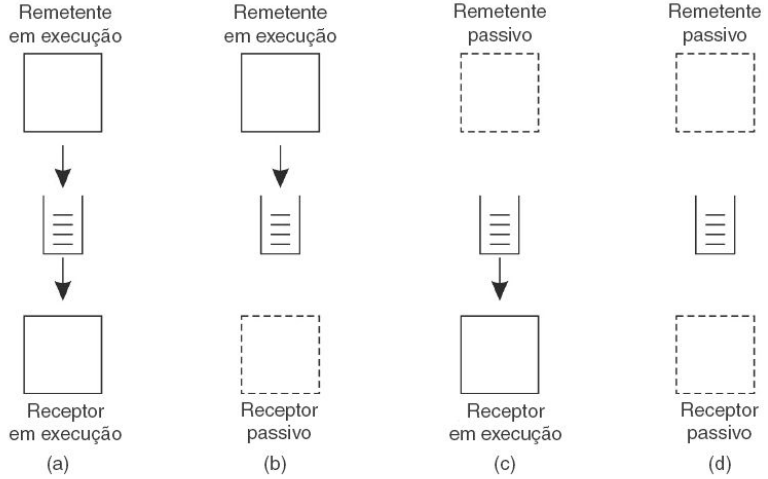
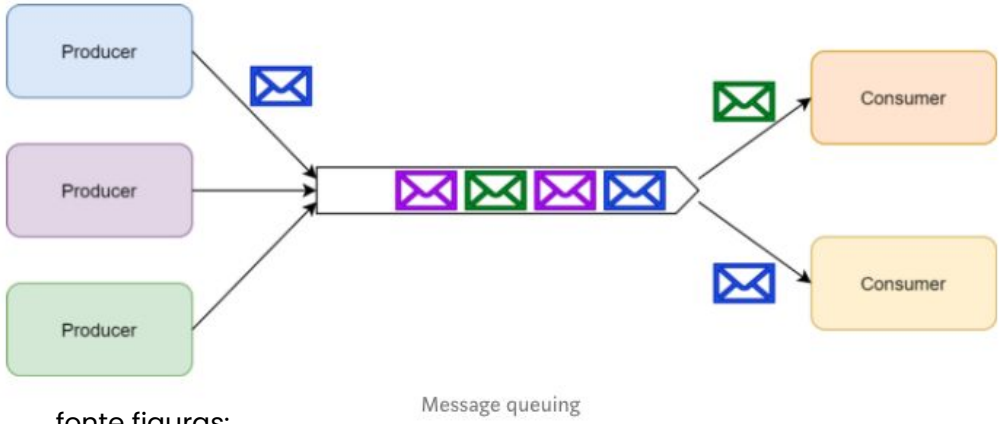


Figura 4.15 Quatro combinações para comunicações fracamente acopladas que utilizam filas.



fonte figuras: https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22a041793

unipampa

Comunicação orientada a mensagem

Exemplos de aplicações específicas

- Sistemas orientados a mensagem são utilizados para integrar sistemas
  - Conversão de mensagens (XML, JSON, Protobuf)
- *Brokers*
  - Ao invés de apenas, converter mensagens, um *broker* é responsável por combinar aplicações com base nas mensagens que são trocadas

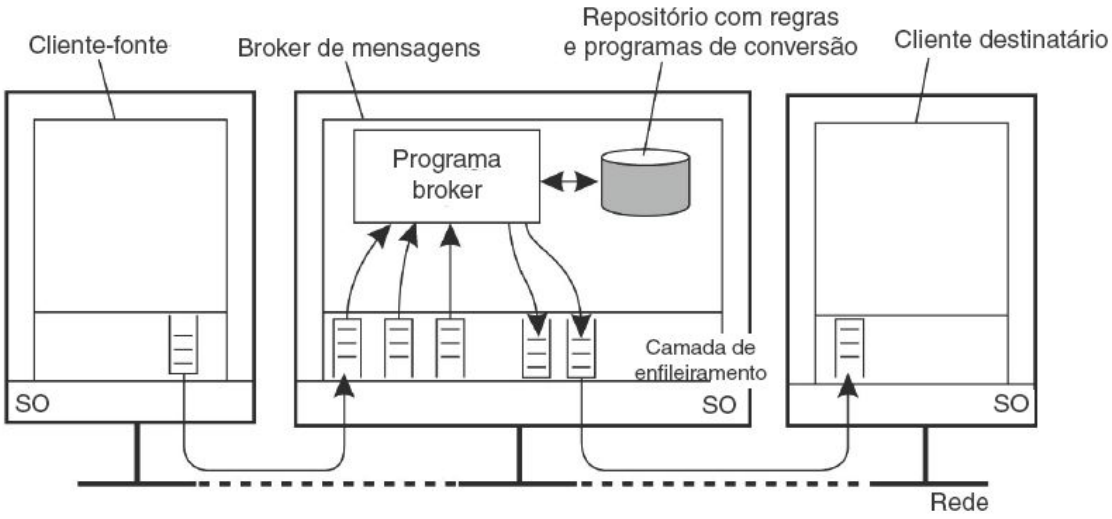
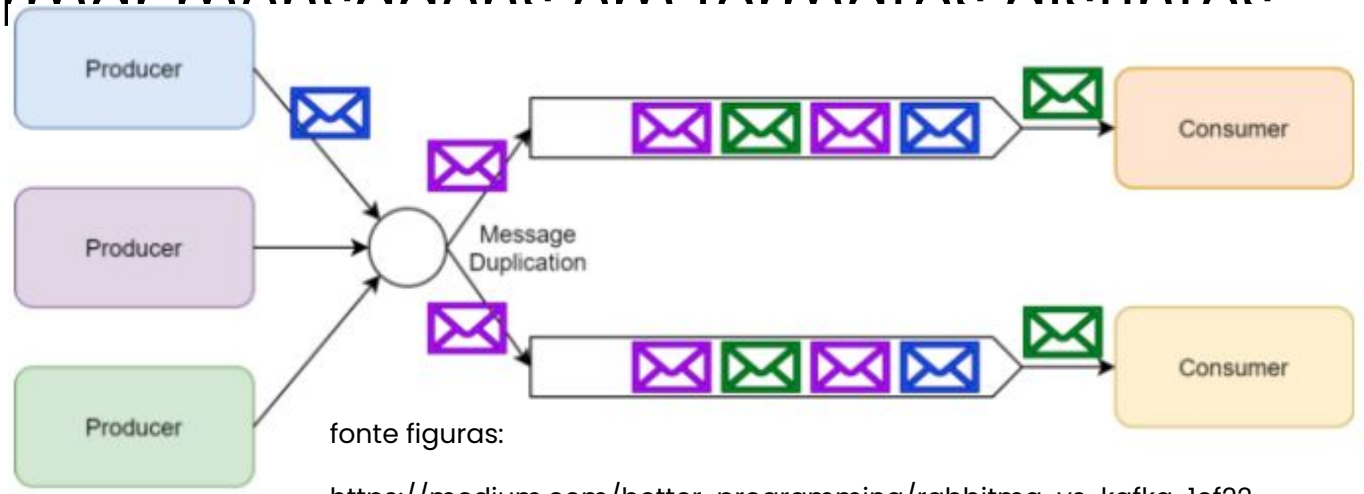


Figura 4.18 Organização geral de um broker de mensagens em um sistema de enfileiramento de mensagens.

unipampa

Comunicação orientada a mensagem
Publicar/Escrever

- Aplicações publicam mensagem sobre tópico X (publicam)
- Outras aplicações declararam interesse no tópico X (subscrevem)
- ***Broker*** pode prover repositório de regras para transformar mensagens em formatos distintos



fonte figuras:

https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22a041793

Exemplos:

- IBM MQ: https://www.ibm.com/products/mq
- Azure Service Bus: https://azure.microsoft.com/en-us/services/service-bus/
- Amazon Simple Queue Service (SQS): https://aws.amazon.com/pt/sqs/

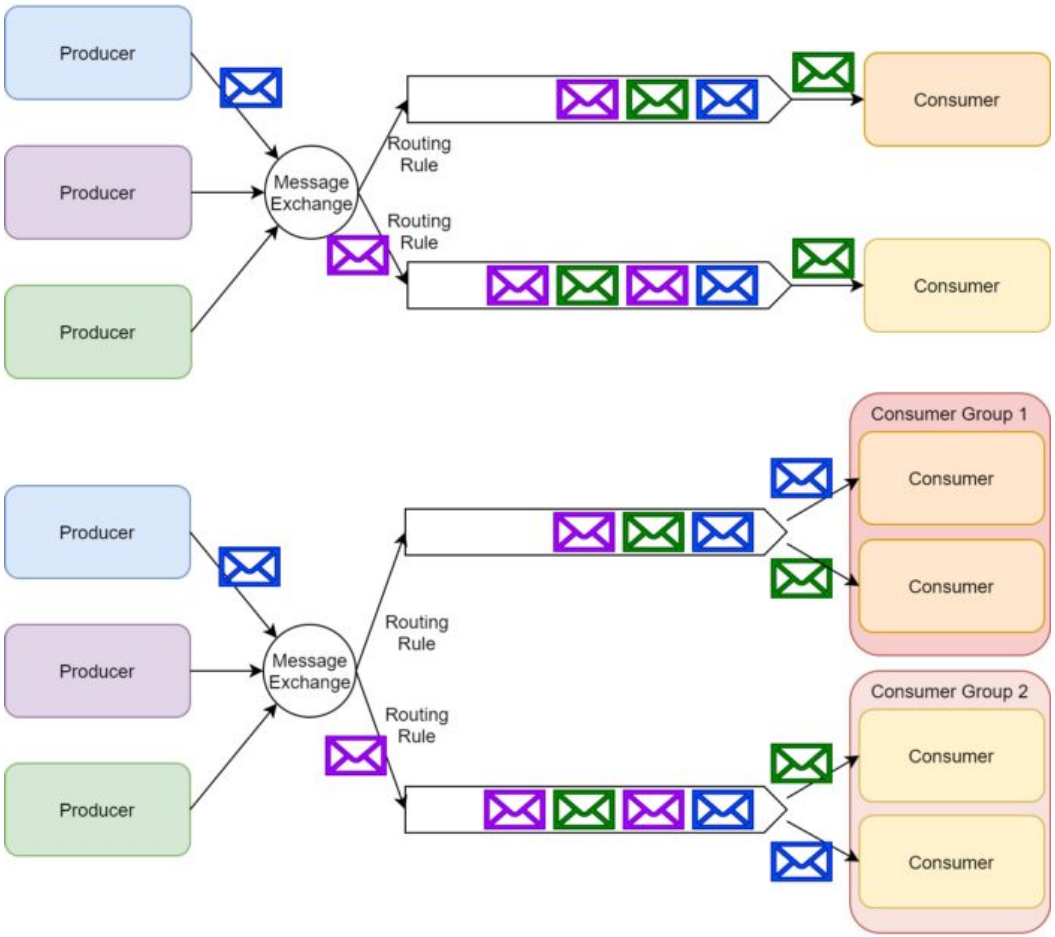
unipampa

Comunicação orientada a mensagem

Rabbit MQ

https://www.rabbitmq.com/#getstarted

- *Asynchronous Messaging*
- vários protocolos de mensagens
- fila de mensagens
- confirmação de entrega
- roteamento flexível para filas

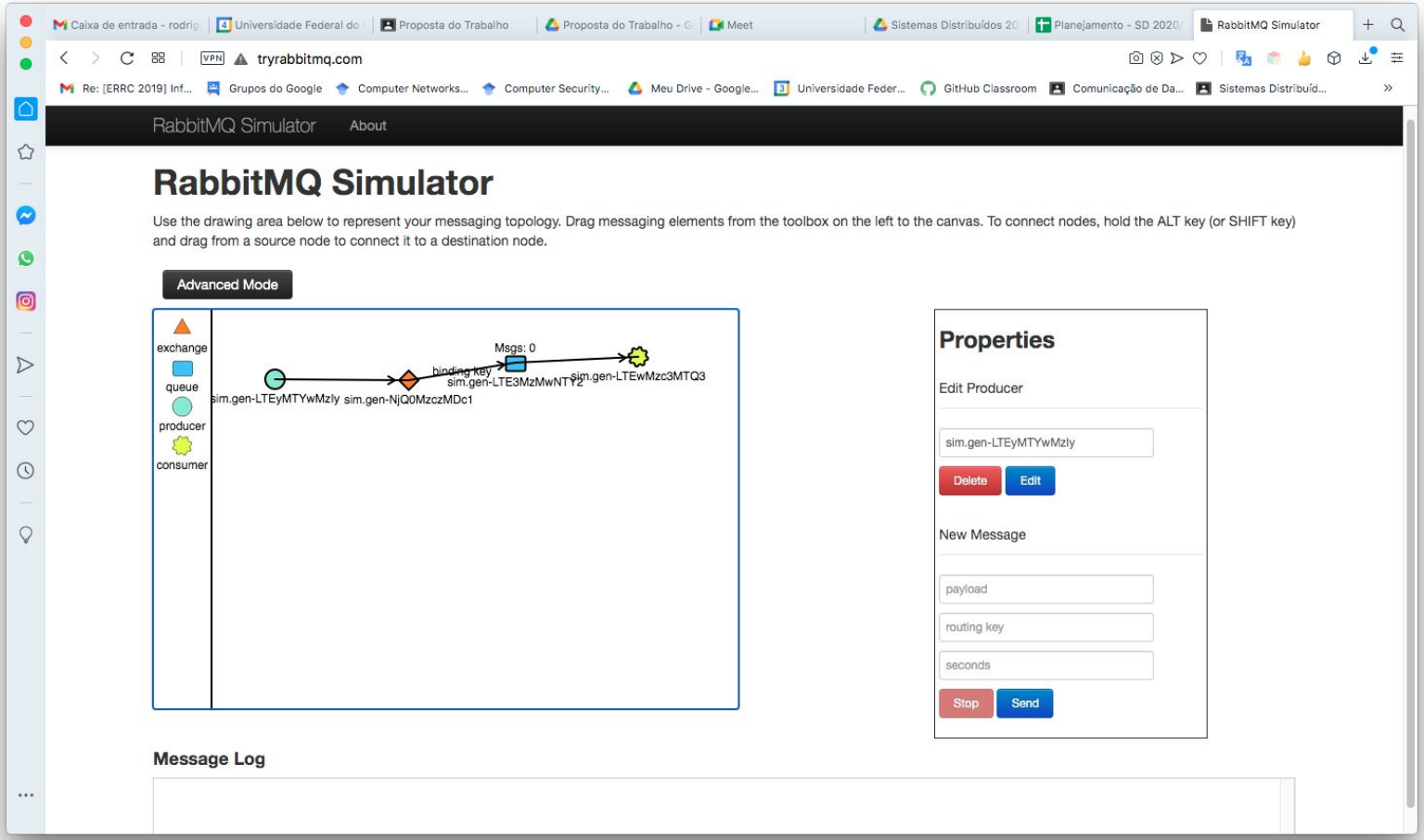


fonte figuras:

https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22

Exemplo

- http://tryrabbitmq.com



Agenda

- ~~Introdução~~
- Comunicação
  - Fundamentos
  - Chamada de procedimento remoto
  - Comunicação orientada a mensagem
  - **Comunicação orientada a fluxo**
- Conclusão

unipampa

Comunicação orientada a fluxo

Motivação

- Até agora, o aspecto temporal tem pouco efeito na correção do sistema:
  - mais rápido ou mais lento, a “corretude” do sistema não é afetada
- Na comunicação orientada a fluxo, a temporização tem papel importante
  - Fluxo de áudio de CD em uma taxa constante
  - Reproduzir em taxa diferente não será correto

unipampa

Comunicação orientada a fluxo

Aplicações

- *Streaming* de eventos é aplicado a uma ampla variedade de casos de uso em uma infinidade de setores e organizações
  - Processar pagamentos e transações financeiras em tempo real, como em bolsas de valores, bancos e seguros.
  - Rastrear e monitorar carros, caminhões, frotas e remessas em tempo real, como na logística e na indústria automotiva.
  - Capturar e analisar continuamente os dados do sensor de dispositivos IoT ou outros equipamentos, como fábricas e parques eólicos.
  - Coletar e reagir imediatamente às interações e pedidos do cliente, como no varejo, no setor de hotéis e viagens e em aplicativos móveis.
  - Monitorar pacientes em cuidados hospitalares e prever mudanças nas condições para garantir o tratamento oportuno em emergências.
  - Conectar, armazenar e disponibilizar dados produzidos por diferentes divisões de uma empresa.
  - Servir como base para plataformas de dados, arquiteturas orientadas a eventos e micro serviços.

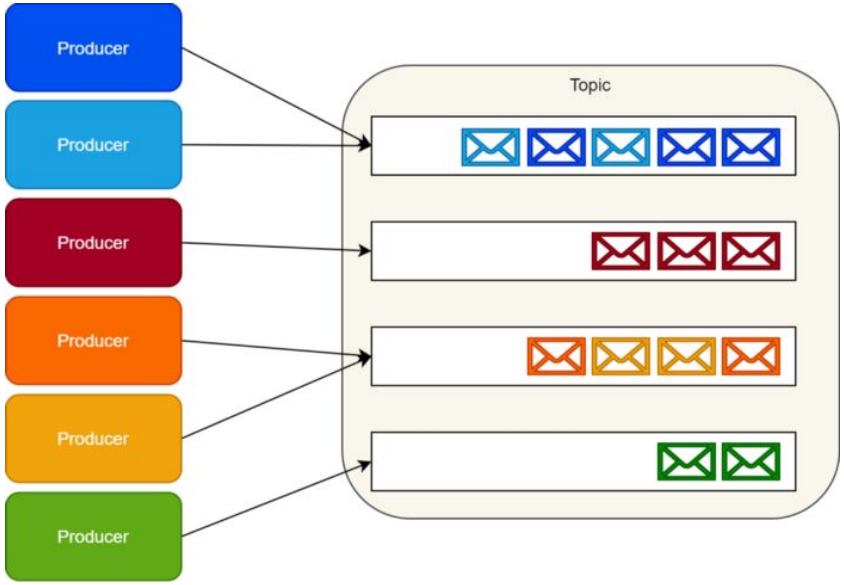
fonte:
https://kafka.apache.org/intro

unipampa

Comunicação orientada a fluxo

Visão geral

- Fluxo de dados
  - Uma sequencia de unidades de dados
  - transmissão isócrono: unidades de dados sejam transferidas no tempo certo
- Tipos
  - fluxo simples: uma sequência de dados
  - fluxo complexo: vários fluxos simples relacionados (subfluxos), podendo haver dependência temporal
- Exemplos
  - Azure Event Hubs
  - AWS Kinesis Data Streams
  - Apache Kafka



fonte figura:
https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22a041793

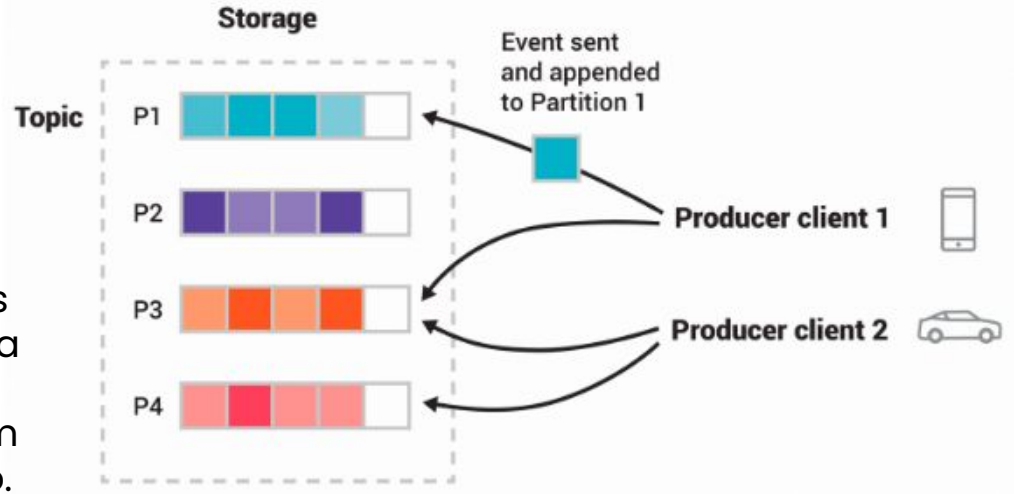
Comunicação orientada a mensagem

Kafka https://kafka.apache.org/intro

- Kafka combina três recursos principais para que você possa implementar seus casos de uso para streaming de eventos de ponta a ponta com solução única:
  - Publicar e assinar fluxos de eventos, incluindo importação/exportação contínua de dados oriundos de outros sistemas
  - Armazenar fluxos de eventos de forma durável e confiável
  - Processar fluxos de eventos conforme eles ocorrem ou retrospectivamente.

Exemplo:

- 4 partições P1 – P4.
- Dois clientes produtores diferentes estão publicando, independentemente um do outro, novos eventos no tópico, gravando eventos na rede nas partições do tópico
- Eventos com a mesma chave (denotados por suas cores na figura) são gravados na mesma partição
- Observe que ambos os produtores podem gravar na mesma partição, se necessário.



fonte:
https://kafka.apache.org/intro

Agenda

- ~~Introdução~~
- ~~Comunicação~~
- **Conclusão**

Conclusão

Revisão

- Fundamentos
  - Importância de middleware entre transporte e aplicação
  - Classificação de comunicação
    - Comunicação persistente vs transiente
    - Comunicação síncrona vs assíncrona
- Chamada de procedimento remoto
  - Um-para-um síncrona (exemplos: RPC, RMI)
- Comunicação orientada a mensagem (assíncrono)
  - Um-para-muitos ou muitos-para-um não persistente (MPI, ZMQ)
  - Persistente através de enfileiramento (RabbitMQ)
- Comunicação orientada a fluxo
  - Streaming de dados (Kafka)

unipampa

Conclusão

Exercícios de fixação

1. Por que serviços de comunicação providos pelos protocolos de transporte frequentemente são considerados inadequados para construir aplicações distribuídas?
2. Descreva como ocorre a comunicação sem conexão entre um cliente e um servidor usando a interface Sockets
3. Suponha que você possa utilizar apenas primitivas de comunicação transiente síncrona. Como você implementaria primitivas para comunicação transiente assíncrona?
4. Explique por que a comunicação transiente síncrona tem problemas inerentes de escalabilidade e como eles podem ser resolvidos.

unipampa

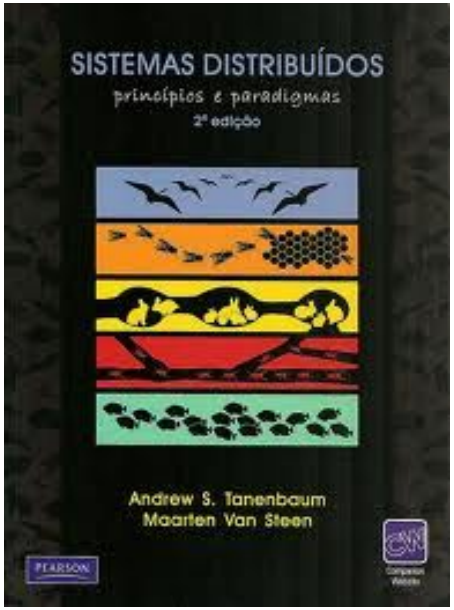
Conclusão
Referências

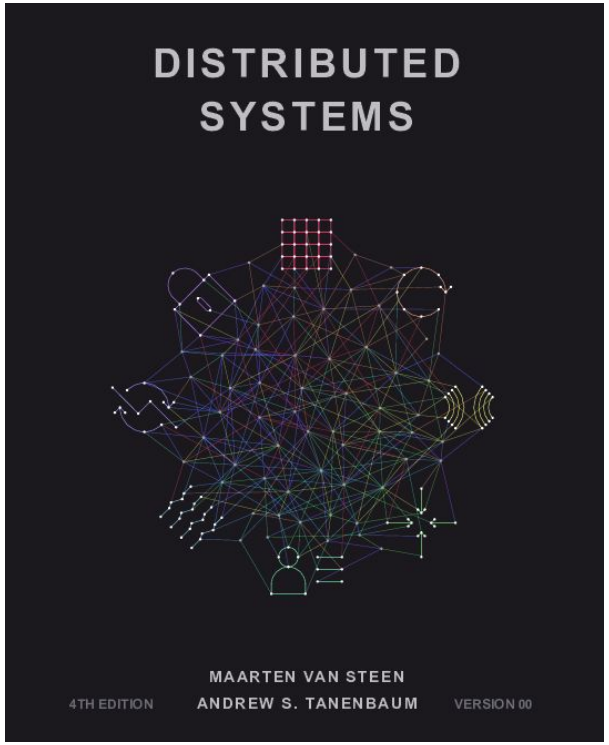
https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22a041793

- Slides (modificados) dos capítulo 4

- TANENBAUM, A. S.; STEEN, M.V. **Sistemas Distribuídos: Princípios e Paradigmas**. 2a ed. São Paulo: Pearson Prentice Hall, 2007.

4º edição, lançada em 2023!
https://www.distributed-systems.net/index.php/books/ds4/





unipampa