

MCTA025-13 - SISTEMAS DISTRIBUÍDOS

COMUNICAÇÃO

Emilio Francesquini

02 de julho de 2018

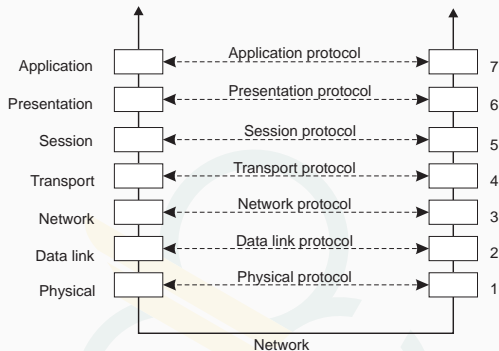
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



- 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>.

- Camadas de baixo nível
- Camada de transporte
- Camada de aplicação
- Camada do middleware

MODELO DE COMUNICAÇÃO BÁSICO



Desvantagens:

- Funciona apenas com passagem de mensagens
- Frequentemente possuem funcionalidades desnecessárias
- Viola a transparência de acesso

Camada física: contém a especificação e implementação dos bits em um quadro, e como são transmitidos entre o remetente e destinatário

Camada de enlace: determina o envio de séries de bits em um quadro, permite detecção de erro e controle de fluxo

Camada de rede: determina como pacotes são roteados em uma rede de computadores

Observação:

Em muitos sistemas distribuídos, a interface de mais baixo nível é a interface de rede.

Importante:

A camada de transporte fornece as ferramentas de comunicação efetivamente utilizadas pela maioria dos sistemas distribuídos.

Protocolos padrões da Internet

TCP: orientada a conexão, confiável, comunicação orientada a fluxo de dados

UDP: comunicação de datagramas não confiável (*best-effort*)

Nota:

IP multicasting é normalmente considerado um serviço padrão (mas essa é uma hipótese perigosa)

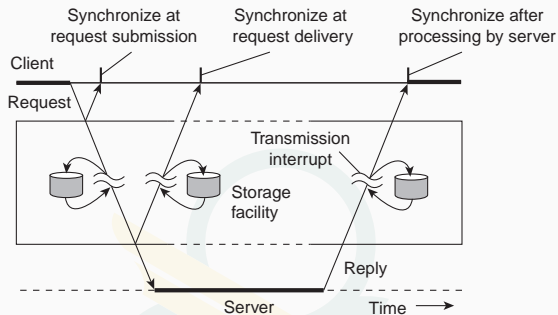
Middleware foi inventado para prover serviços e protocolos **frequentemente usados** que podem ser utilizados por várias aplicações **diferentes**.

- Um conjunto rico de **protocolos de comunicação**
- **(Des)empacotamento** [(un)marshaling] de dados, necessários para a integração de sistemas
- **Protocolos de gerenciamento de nomes**, para auxiliar o compartilhamento de recursos
- **Protocolos de segurança** para comunicações seguras
- **Mecanismos de escalabilidade**, tais como replicação e caching

Observação:

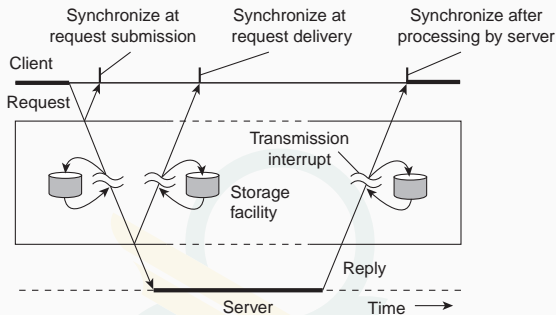
O que realmente sobra são protocolos específicos de aplicação.

TIPOS DE COMUNICAÇÃO



- Comunicação **transiente** vs. **persistente**
- Comunicação **assíncrona** vs. **síncrona**

TIPOS DE COMUNICAÇÃO

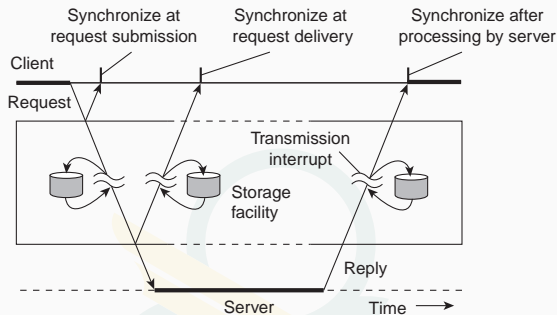


Transiente vs. persistente

Comunicação transiente: remetente descarta a mensagem se ela não puder ser encaminhada para o destinatário

Comunicação persistente: uma mensagem é guardada no remetente pelo tempo que for necessário, até ser entregue no destinatário

TIPOS DE COMUNICAÇÃO



Pontos de sincronização

- No envio da requisição
- Na entrega da requisição
- Após o processamento da requisição

Computação Cliente/Servidor geralmente é baseada em um modelo de **comunicação transiente síncrona**:

- Cliente e servidor devem estar ativos no momento da comunicação
- Cliente envia uma requisição e bloqueia até que receba sua resposta
- Servidor essencialmente espera por requisições e as processa

Computação Cliente/Servidor geralmente é baseada em um modelo de **comunicação transiente síncrona**:

- Cliente e servidor devem estar ativos no momento da comunicação
- Cliente envia uma requisição e bloqueia até que receba sua resposta
- Servidor essencialmente espera por requisições e as processa

Desvantagens de comunicação síncrona:


- o cliente não pode fazer nenhum trabalho enquanto estiver esperando por uma resposta
- falhas precisam ser tratadas imediatamente (afinal, o cliente está esperando)
- o modelo pode não ser o mais apropriado (mail, news)

Middleware orientado a mensagens

tem como objetivo prover **comunicação persistente assíncrona**:

- Processos trocam mensagens entre si, as quais são armazenadas em uma fila
- O remetente não precisa esperar por uma resposta imediata, pode fazer outras coisas enquanto espera
- Middleware normalmente assegura tolerância a falhas

RPC — CHAMADAS A PROCEDIMENTOS REMOTOS

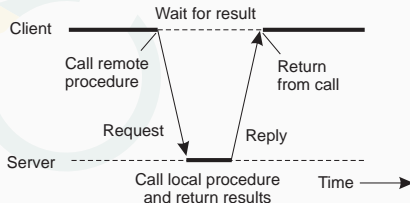
- Funcionamento básico de RPCs
 - Passagem de parâmetros
 - Variações
- 

FUNCIONAMENTO BÁSICO DE RPC

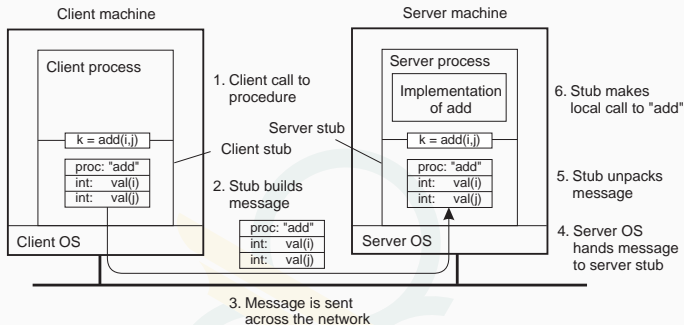
- Desenvolvedores estão familiarizados com o modelo de procedimentos
- Procedimentos bem projetados operam isoladamente (*black box*)
- Então não há razão para não executar esses procedimentos em máquinas separadas

Conclusão

Comunicação entre o chamador & chamado podem ser escondida com o uso de mecanismos de chamada a procedimentos.



FUNCIONAMENTO BÁSICO DE RPC



1. Procedimento no cliente chama o *stub* do cliente
2. *Stub* constrói mensagem; chama o SO local
3. SO envia msg. para o SO remoto
4. SO remoto repassa mensagem para o *stub*
5. *Stub* desempacota parâmetros e chama o servidor
6. Servidor realiza chamada local e devolve resultado para o *stub*
7. *Stub* constrói mensagem; chama SO
8. SO envia mensagem para o SO do cliente
9. SO do cliente repassa msg. para o *stub*
10. *Stub* do cliente desempacota resultado e devolve para o cliente

Empacotamento de parâmetros

há mais do que apenas colocá-los nas mensagens:

- As máquinas cliente e servidor podem ter **representação de dados diferentes** (ex: ordem dos bytes)
- Empacotar um parâmetro significa **transformar um valor em uma sequência de bytes**
- Cliente e servidor precisam concordar com a mesma regra de codificação (*encoding*):
 - Como os **valores dos dados básicos** (inteiros, números em ponto flutuante, caracteres) são representados?
 - Como os **valores de dados complexos** (vetores, *unions*) são representados?
- Cliente e servidor precisam **interpretar corretamente as mensagens**, transformando seus valores usando representações dependentes da máquina

Algumas suposições:

- semântica de **copy in/copy out**: enquanto um procedimento é executado, nada pode ser assumido sobre os valores dos parâmetros
- **Todos** os dados são passado apenas por parâmetro. Exclui passagem de *referências para dados (globais)*.

Algumas suposições:

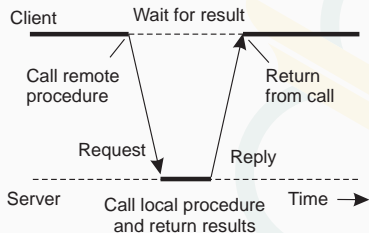
- semântica de **copy in/copy out**: enquanto um procedimento é executado, nada pode ser assumido sobre os valores dos parâmetros
- **Todos** os dados são passado apenas por parâmetro. Exclui passagem de *referências para dados (globais)*.

Conclusão

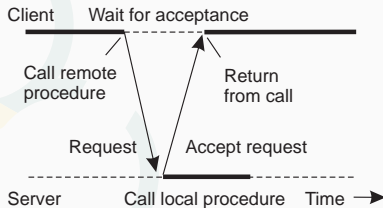
Não é possível assumir transparência total de acesso.

Ideia geral

Tentar se livrar do comportamento estrito de requisição-resposta, mas permitir que o cliente continue sem esperar por uma resposta do servidor.

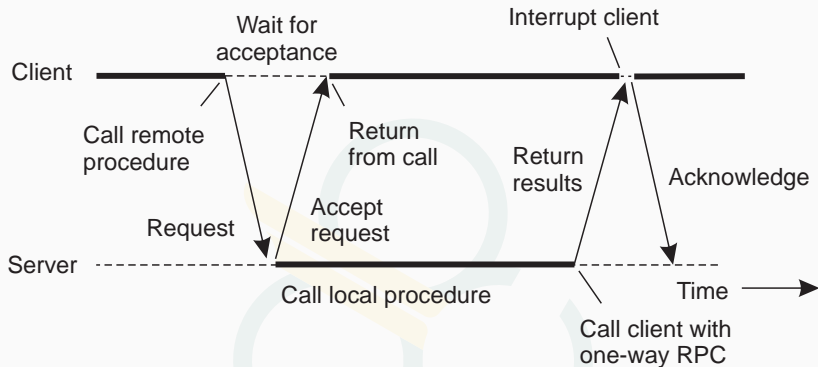


(a)



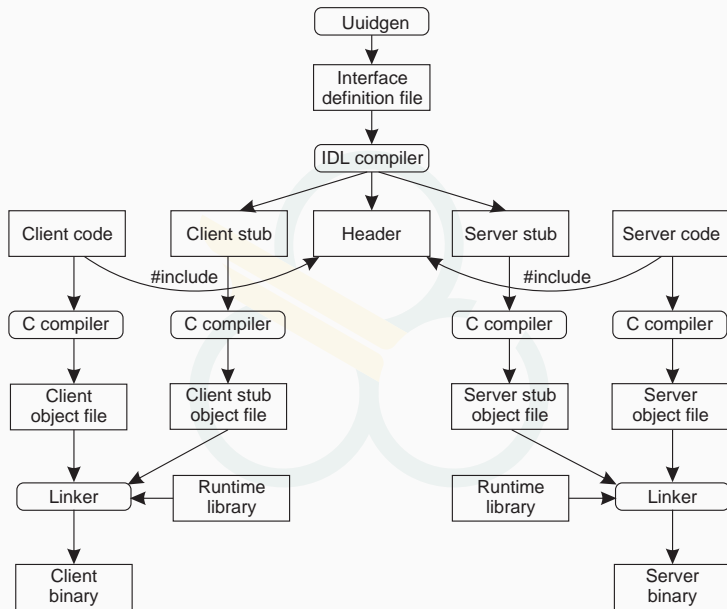
(b)

RPC SÍNCRONO DIFERIDO



Variação

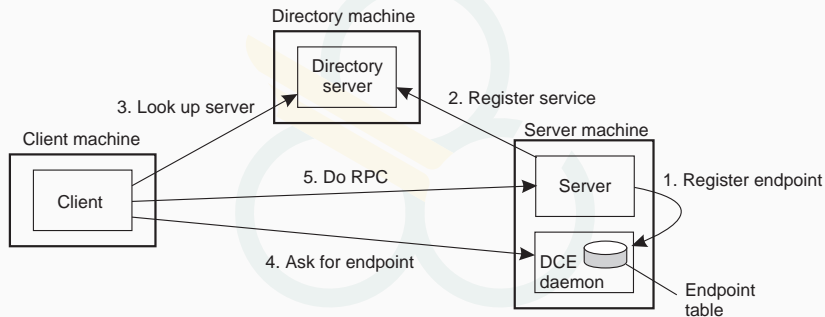
Cliente pode também realizar uma consulta (*poll*) (bloqueante ou não) para verificar se os resultados estão prontos.



VINCULAÇÃO CLIENTE-SERVIDOR (DCE)

Problemas

- (1) Cliente precisa localizar a máquina com o servidor e,
- (2) precisa localizar o servidor.



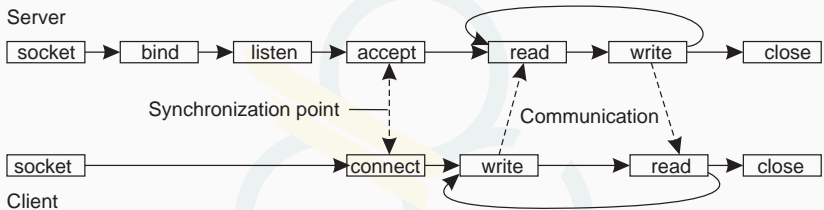
COMUNICAÇÃO ORIENTADA A MENSAGENS

- Mensagens transientes
- Sistema de enfileiramento de mensagens
- *Message brokers*
- Exemplo: IBM Websphere

Berkeley socket interface

SOCKET	Cria um novo ponto de comunicação
BIND	Especifica um endereço local ao socket
LISTEN	Anuncia a vontade de receber N conexões
ACCEPT	Bloqueia até receber um pedido de estabelecimento de conexão
CONNECT	Tenta estabelecer uma conexão
SEND	Envia dados por uma conexão
RECEIVE	Recebe dados por uma conexão
CLOSE	Libera a conexão

MENSAGENS TRANSIENTES: SOCKETS



```
import socket
HOST = socket.gethostname()           # e.g. 'localhost'
PORT = SERVERPORT                      # e.g. 80
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(N)                            # listen to max N queued connection
conn, addr = s.accept()                # new socket + addr client
while 1: # forever
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

Ideia geral

Comunicação assíncrona e persistente graças ao uso de **filas** pelo middleware. Filas correspondem a buffers em servidores de comunicação.

PUT	Adiciona uma mensagem à fila especificada
GET	Bloqueia até que a fila especificada tenha alguma mensagem e remove a primeira mensagem
POLL	Verifica se a fila especificada tem alguma mensagem e remove a primeira. Nunca bloqueia
NOTIFY	Instala um tratador para ser chamado sempre que uma mensagem for inserida em uma dada fila

Observação:

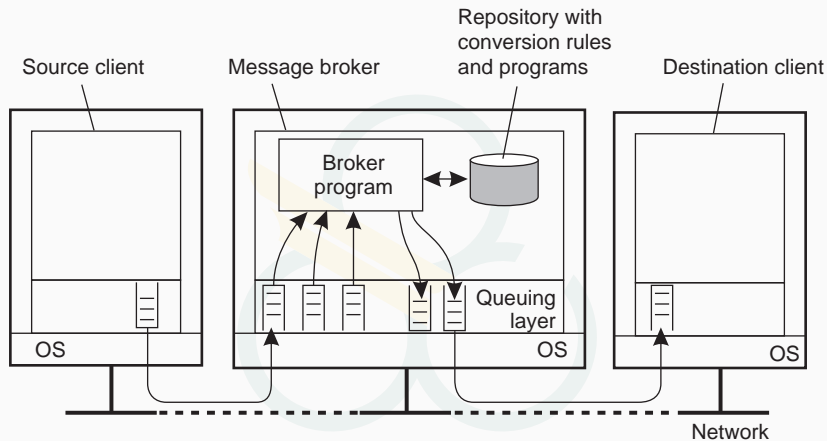
Sistemas de filas de mensagens assumem um **protocolo comum de troca de mensagens**: todas as aplicações usam o mesmo formato de mensagem (i.e., estrutura e representação de dados)

Message broker

Componente centralizado que lida com a heterogeneidade das aplicações:

- transforma as mensagens recebidas para o formato apropriado
- frequentemente funciona como um **application gateway**
- podem rotear com **base no conteúdo** ⇒ **Enterprise Application Integration**

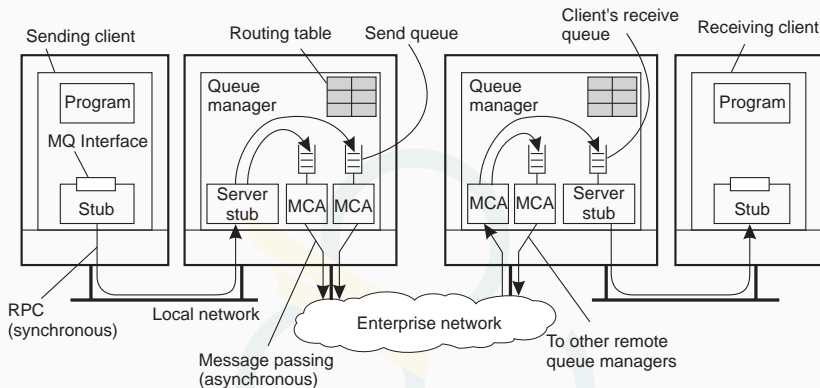
MESSAGE BROKER



- Mensagens específicas da aplicação são colocadas e removidas de filas
- As filas são controladas por um gerenciador de filas
- Processos podem colocar mensagens apenas em filas locais, ou usando um mecanismo de RPC

Transferência de mensagens

- Mensagens são transferidas entre filas
- Mensagens transferidas entre filas em diferentes processos requerem um canal
- Em cada ponta do canal existe um agente de canal, responsável por:
 - configurar canais usando ferramentas de rede de baixo nível (ex: TCP/IP)
 - (Des)empacotar mensagens de/para pacotes da camada de transporte
 - Enviar/receber pacotes



- Canais são unidirecionais
- Agentes de canais são automaticamente iniciados quando uma mensagem chega
- Pode-se criar redes de gerenciadores de filas
- Rotas são configuradas manualmente (pelo admin do sistema)

Roteamento

O uso de **nomes lógicos**, combinados com resolução de nomes para filas locais, permitem que uma mensagem seja colocada em uma **fila remota**.

