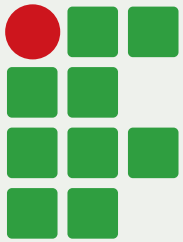
Comunicação orientada a mensagens

STD29006 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

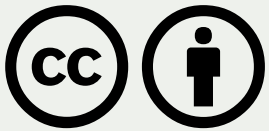
mello@ifsc.edu.br



INSTITUTO FEDERAL
Santa Catarina

Câmpus
São José

Licenciamento



Revisão das aulas anteriores

Alguns pontos na comunicação cliente/servidor

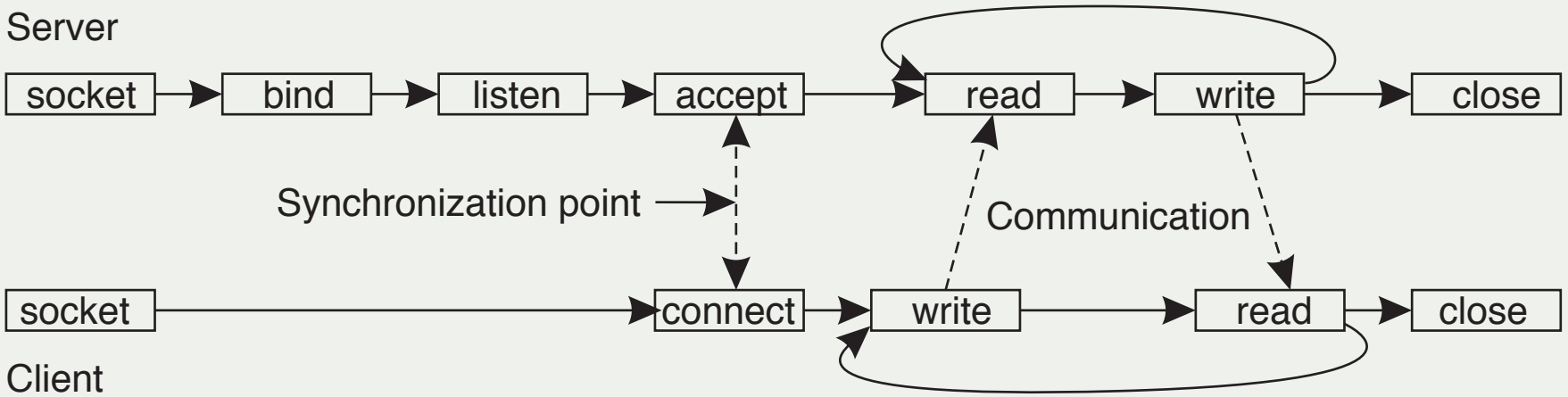
- **Protocolo de comunicação**
  - Podem ser definidos como acordos/regras para comunicação
  - Podem ser orientados a conexão ou não (Ex: TCP x UDP)
- **Modelo de comunicação cliente/servidor**
  - Servidores oferecem serviços aos clientes
  - Paradigma pedido e resposta
  - Implementação: Socket, RPC e RMI
- **Servidor concorrente x sequencial**
  - Sequencial: servidor atende um pedido por vez
  - Concorrente: servidor dispara uma *thread* ou processo filho para lidar com cada pedido que chega

Alguns pontos na comunicação cliente/servidor

- **Endereçamento**
  - Como localizar o servidor?
    - Conhecimento prévio, *broadcast* ou serviço de nomes
- **Bloqueante x não bloqueante**
  - Síncrono – emissor/receptor fica bloqueado até enviar/receber mensagem
  - Assíncrono – emissor não fica bloqueado e o retorno também não bloqueia
- **Área de armazenamento temporário (buffer) - com x sem**
  - Servidor precisa ficar esperando uma mensagem antes que o cliente possa enviá-la (consumo imediato)
  - Cliente envia para uma área de armazenamento temporário e o servidor depois consulta esta área

Comunicação **transitória** por mensagens

- **RPC e RMI contribuem** para esconder a comunicação no sistema distribuídos (**transparência**)
- **RPC e RMI assumem** que ambas as **partes estarão ativas no mesmo instante** para permitir a comunicação



Comunicação **persistente** por mensagens

Comunicação **persistente** por mensagens

- **Middleware orientado a mensagem** (MOM) – sistema de fila de mensagens
  - Provê suporte a comunicação assíncrona e persistente
  - Cliente e servidor não precisam estar ativos no mesmo instante para permitir a troca de mensagens
- **Comunicação** entre aplicações é feita **por meio de filas de mensagens**
  - **Mensagens** podem ser **roteadas** por diversos **servidores intermediários** e eventualmente ser entregue ao destino

Comunicação **persistente** por mensagens

- **Middleware orientado a mensagem** (MOM) – sistema de fila de mensagens
  - Provê suporte a comunicação assíncrona e persistente
  - Cliente e servidor não precisam estar ativos no mesmo instante para permitir a troca de mensagens
- **Comunicação** entre aplicações é feita **por meio de filas de mensagens**
  - **Mensagens** podem ser **roteadas** por diversos **servidores intermediários** e eventualmente ser entregue ao destino

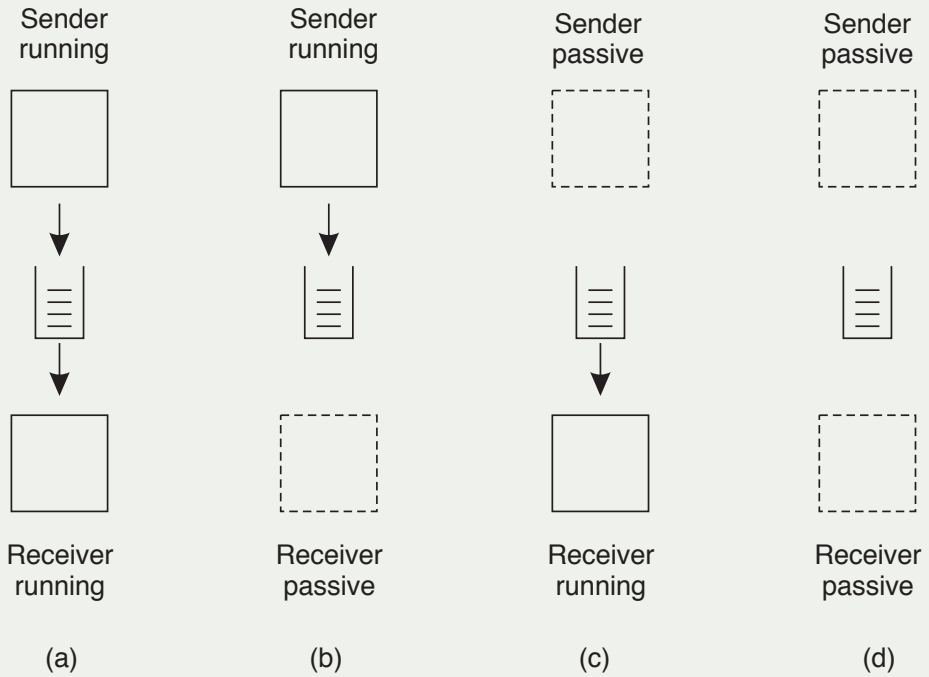
Quando usar?

Quando as partes estiverem interconectadas por meio de uma WAN, cuja probabilidade de desconexão temporária não for desprezível

Modelo de fila de mensagens

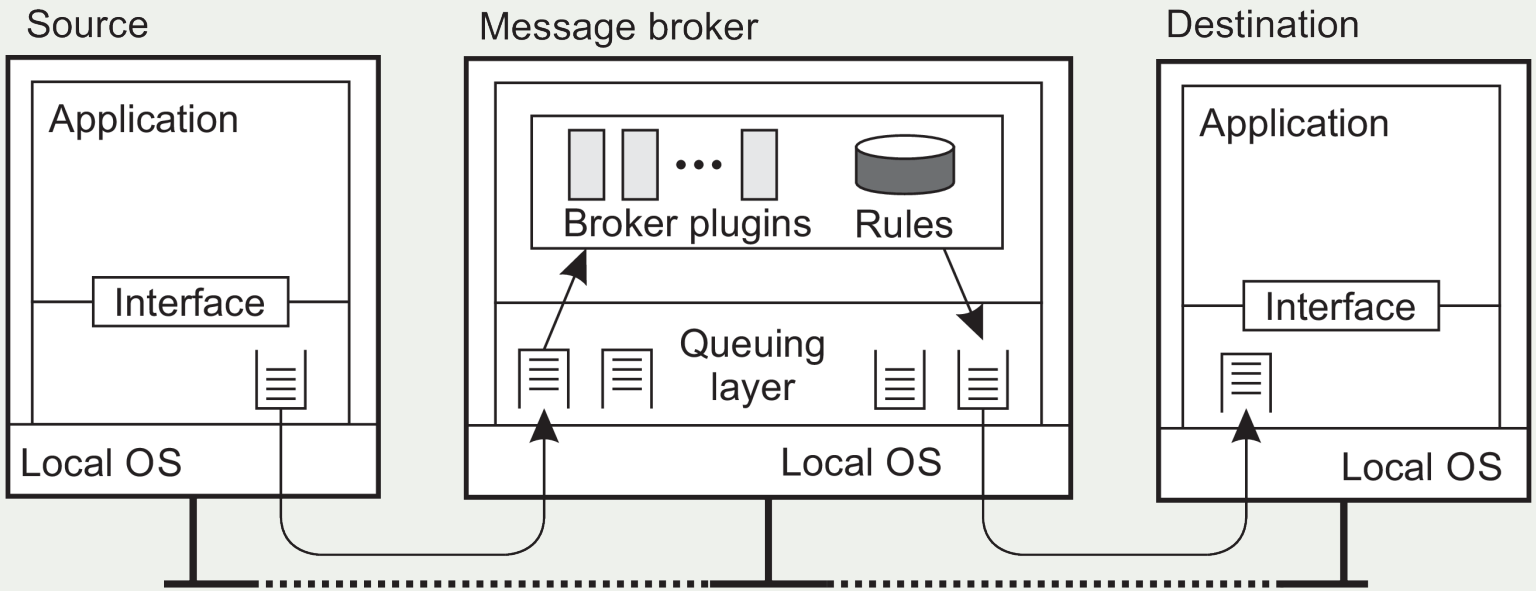
Comunicação persistente e assíncrona – desacoplamento temporal

- Mensagens ficam armazenadas até que o receptor possa consumi-las
  - Emissor só tem garantia que a mensagem será eventualmente inserida na fila do receptor
  - Quando a mensagem será consumida ou se será consumida, isto depende do comportamento do receptor



Message broker – intermediador

- O *broker* é responsável por lidar com heterogeneidade de aplicações
- Transforma as mensagens que chegam para o formato da aplicação destino
- Pode implementar um conjunto de facilidades para as aplicações, por exemplo, roteamento de mensagens, serviço de assinatura e publicação (*publish* & *subscribe*)



Interface básica para sistemas de filas de mensagens

- **PUT**
  - Colocar uma mensagem em uma fila
- **GET**
  - Se a fila estiver vazia ficará bloqueada até chegar mensagem. Remove a primeira mensagem que estiver na fila
- **POLL**
  - Verificar se existe mensagem em uma fila específica e remover a primeira. Neste caso, não fica bloqueado
- **NOTIFY**
  - Definir uma função que será invocada sempre que uma mensagem for inserida na fila

Arquitetura básica de um sistema de filas de mensagens

- Cada **fila possui um identificador único** em todo o sistema distribuído e as **mensagens** são **destinadas para uma fila**
  - Filas são distribuídas por diversas máquinas
  - Identificar cada fila implica em fazer um mapeamento para endereços de rede e portas
- **Cada máquina** oferece uma **interface** para o **envio** e **recepção** de mensagens
- Máquinas **clientes** podem estar **ligadas** a **uma ou mais** máquinas **servidoras** responsáveis pelo encaminhamentos de mensagens (roteamento)

Algumas implementações de comunicações por mensagens

Advanced Message Queuing Protocol – AMQP

Padrão aberto para *middleware* orientado à mensagens – enfileiramento, roteamento ponto-a-ponto ou pub-sub, confiabilidade e segurança

- **Apache ActiveMQ**
- **Apache Qpid**
- **RabbitMQ**

Message Queue Telemetry Transport – MQTT

Padrão ISO baseado em uma versão leve do *publish-subscribe* para ser usado sobre TCP/IP – ambientes que exigem código pequeno & taxa de transmissão baixa

- **Mosquitto**

Comunicação transitória

Tornando mais fácil o uso do *sockets*

- Desenvolver com *sockets* consiste em uma tarefa de baixo nível e erros de programação podem ser comuns
- A forma de uso do sockets geralmente segue o modelo cliente/servidor

Tornando mais fácil o uso do *sockets*

- Desenvolver com *sockets* consiste em uma tarefa de baixo nível e erros de programação podem ser comuns
- A forma de uso do sockets geralmente segue o modelo cliente/servidor

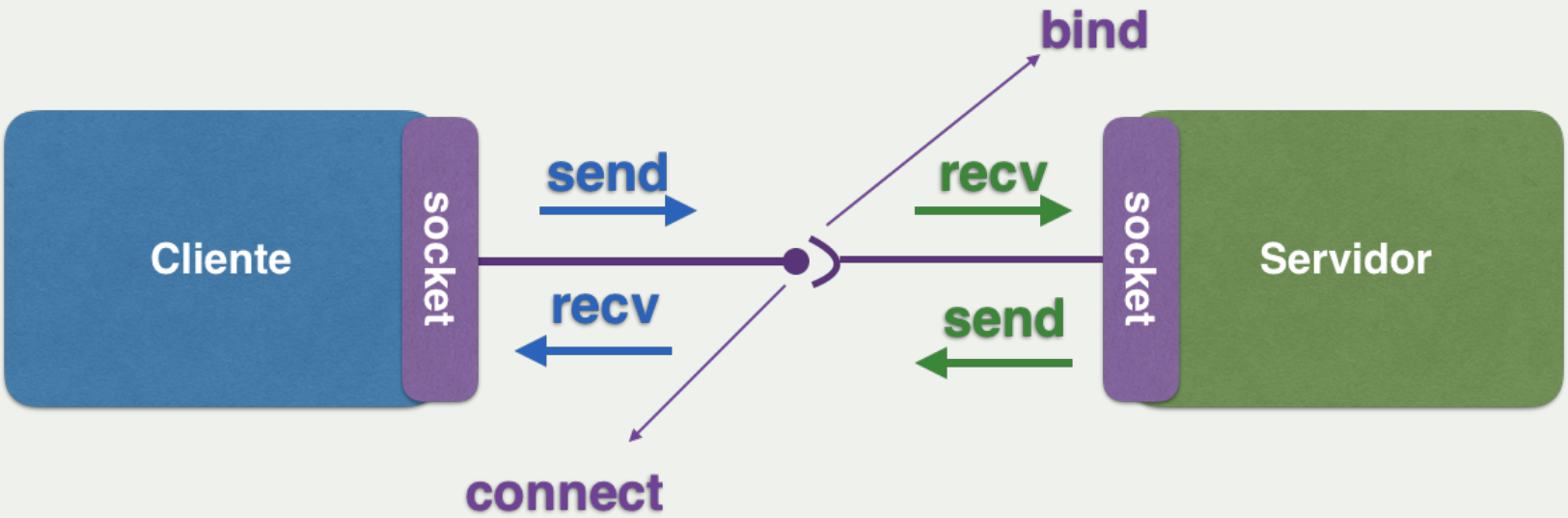
ZeroMQ – uma alternativa para desenvolvimento com sockets

- Biblioteca com abstração de alto nível para trabalhar com **pares de sockets**
- Um socket para enviar mensagens do processo P e um par correspondente no processo Q para receber mensagens
- Toda **comunicação é assíncrona**

ZeroMQ ou ∅MQ

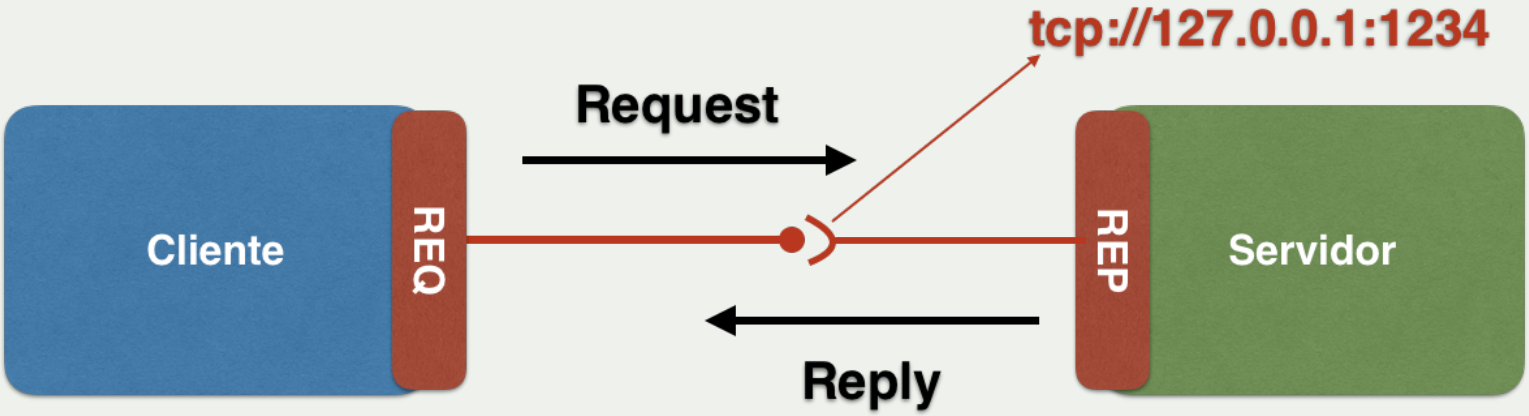
- Suporte a comunicação *many-to-one* ao invés do *one-to-one*
- Suporte a comunicação *one-to-many* (*multicast*)
- Protocolos de transporte
  - *Threads* em um único processo – inproc://nome
  - Processos em uma mesma máquina – ipc:///tmp/arquivo
  - Processos em máquinas diferentes – tcp://host:port
  - Comunicação multicast – epgm://;239.0.0.1:1234
- Implementa os seguintes padrões
  - **Request-reply**
  - **Publish & Subscribe**
  - **Pipeline**

ZeroMQ: Request-reply



- Comunicação tradicional via sockets TCP

ZeroMQ: Request-reply



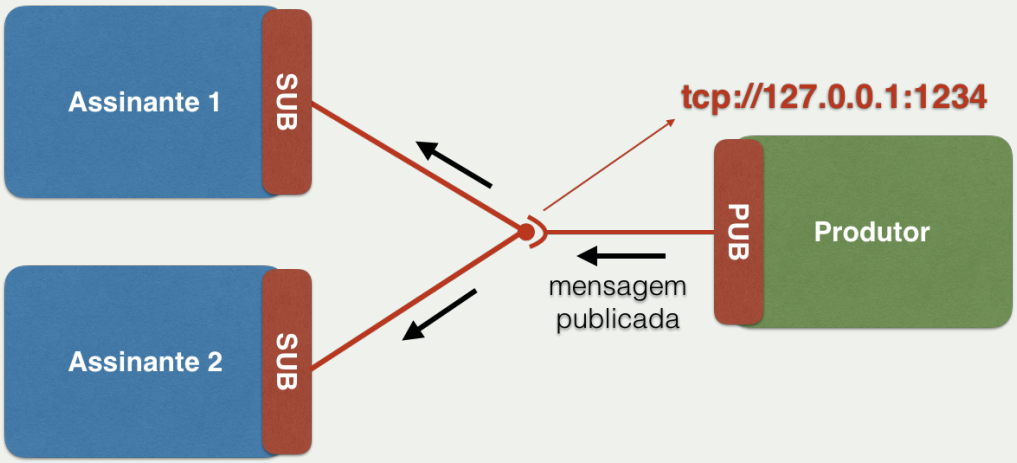
- Servidor usa um **socket REP** e cliente usa um **socket REQ**
- Filas persistem pedidos (no cliente) e respostas (no servidor)
  - Cliente precisa receber resposta de um pedido antes de fazer um novo pedido para o mesmo servidor
  - Servidor precisa enviar resposta antes de receber um novo pedido do mesmo cliente
- Cliente pode conectar em um ou mais servidores

ZeroMQ: Request-reply

```
import org.zeromq.ZMQ;
public class Servidor{
  public static void main(String args[]){
    ZMQ.Context ctx = ZMQ.context(1);
    ZMQ.Socket socket = ctx.socket(ZMQ.REP);
    socket.bind("tcp://*:1234");
    while(true){
      byte[] req = socket.recv(0);
      socket.send("World", 0);
    }
  }
}
```

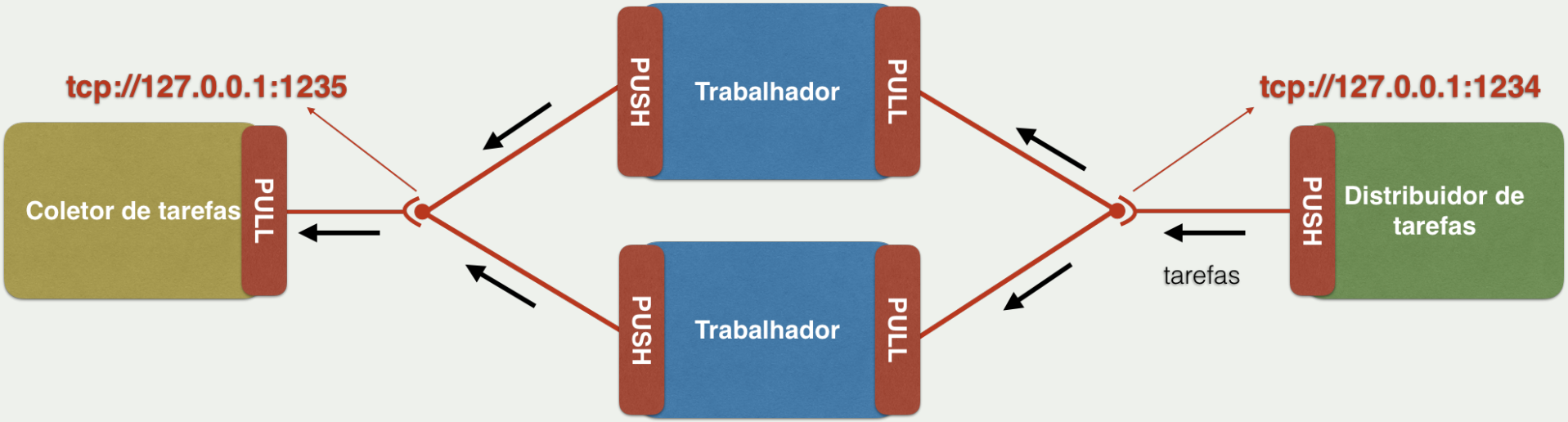
```
import org.zeromq.ZMQ;
public class Cliente{
  public static void main(String args[]){
    ZMQ.Context ctx = ZMQ.context(1);
    ZMQ.Socket socket = ctx.socket(ZMQ.REQ);
    socket.connect("tcp://localhost:1234");
    socket.send("Hello", 0);
    System.out.println(socket.recv(0));
  }
}
```


ZeroMQ: Publish & Subscribe



- Produtor usa **socket PUB** e consumidor usa **socket SUB**
  - **PUB** envia mensagem **para todos assinantes**
- Pode assinar todas mensagens do produtor (*SubscribeALL*) ou somente mensagens específicas
- Consumidor pode ser assinantes de múltiplos produtores
- Se produtor enviar uma mensagem e não houver assinantes, essa será perdida

ZeroMQ: Pipeline



- Usado em cenários que se deseja processamento de dados em paralelo
- Tarefas são distribuídas para processos trabalhadores usando varredura cíclica (*round-robin*)
  - **PUSH** envia mensagem **para um dos trabalhadores**

Exercício 01

- Quero desenvolver um aplicativo de mensagens instantâneas para Android
- Apresente o modelo conceitual ilustrando como seria a troca de mensagens entre dois usuários
- Apresente as bibliotecas, serviços ou frameworks que precisariam ser usados para desenvolver tal solução, bem como os requisitos de rede para esses serviços funcionarem (caso existam)

Comunicação confiável: modelo cliente e servidor

Comunicação confiável: cliente-servidor

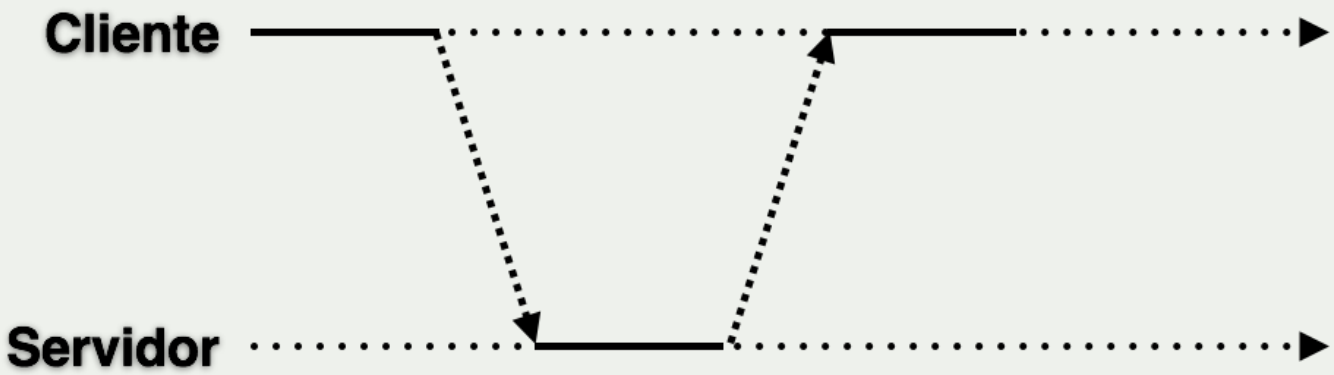
- Um **processo** é considerado **falho** se durante sua execução apresentar um **comportamento diferente daquilo que foi especificado**
  - Omissão de recepção – não recebe mensagens enviadas a ele
  - Omissão de envio – não envia mensagens que se espere que ele envie
  - Parada (ou queda) – deixa de funcionar
  - Arbitrária – continua a funcionar, porém produz saídas incorretas

Canais de comunicação podem exibir falhas por queda, omissão, expiração de tempo (*timeout*) e arbitrárias

- **Canais confiáveis** mascaram falhas por queda e omissão

Comunicação confiável: cliente-servidor

Comunicação ponto-a-ponto



Cliente

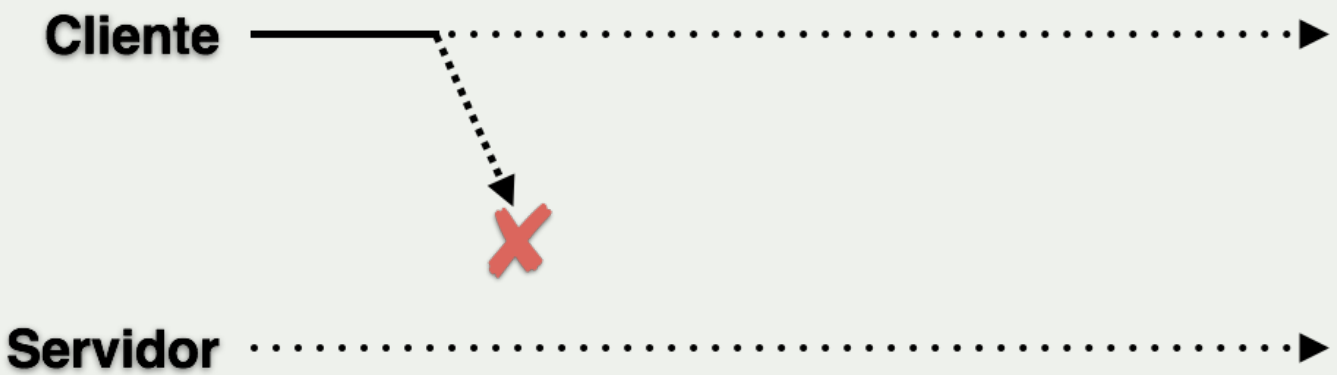
1. Encontra servidor
2. Codifica mensagem
3. Envia mensagem
4. Recebe resposta

Servidor

1. Recebe mensagem
2. Decodifica mensagem
3. Processa
4. Envia resposta

Comunicação confiável: cliente-servidor

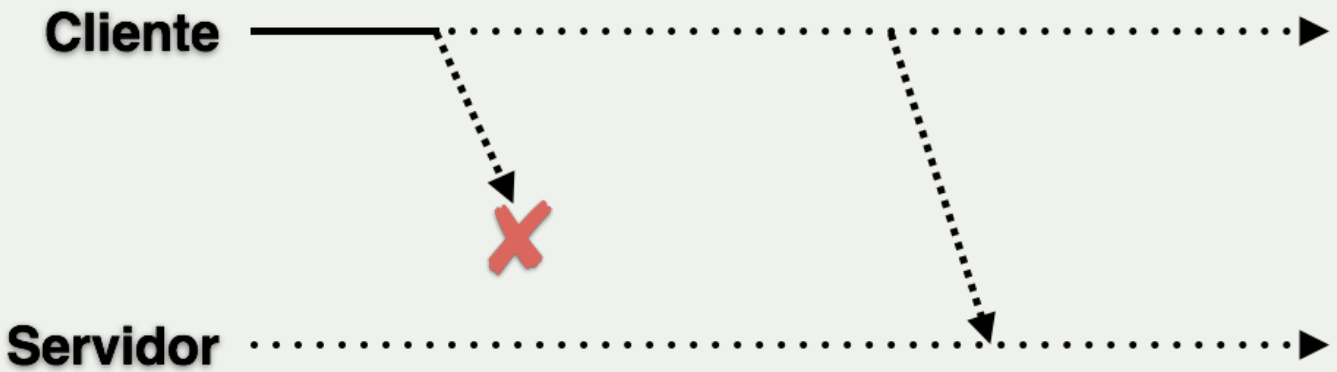
Comunicação ponto-a-ponto



- O que fazer se o pedido for perdido?

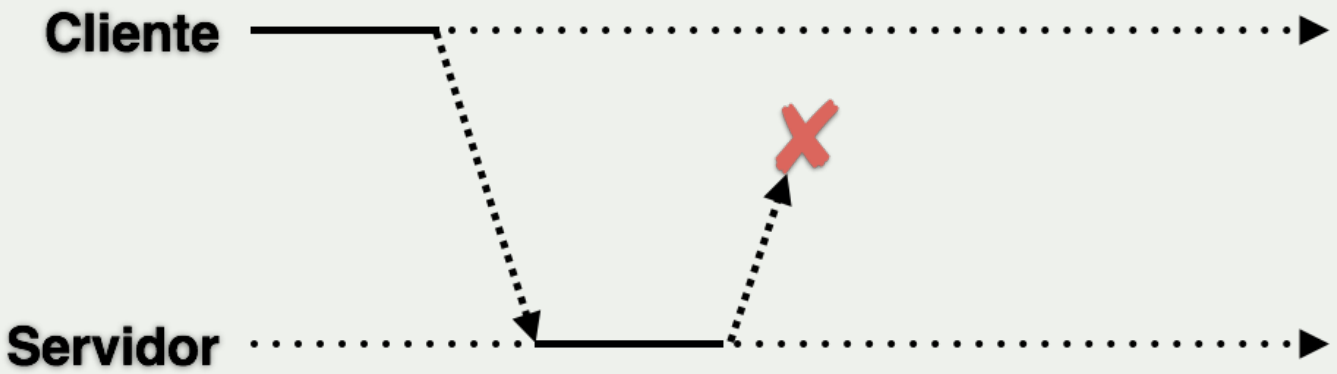
Comunicação confiável: cliente-servidor

Comunicação ponto-a-ponto



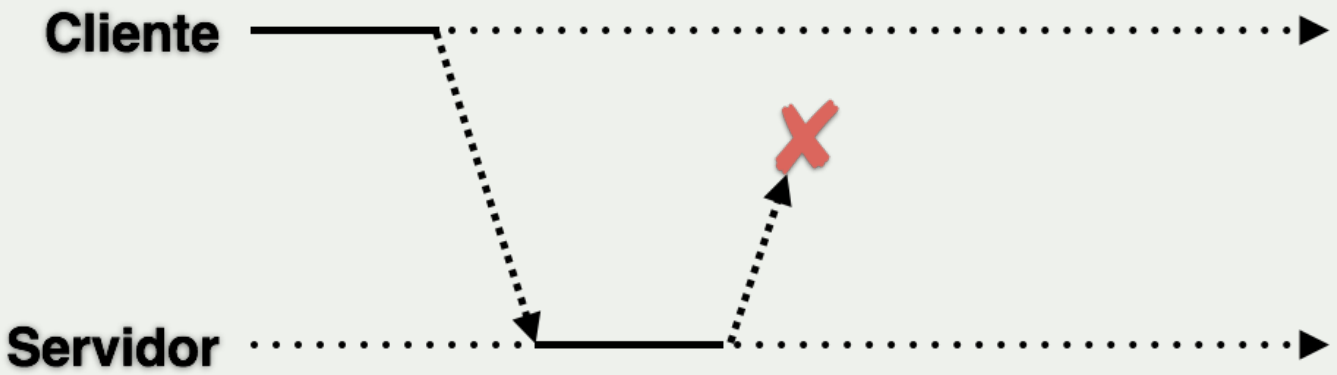
- O que fazer se o pedido for perdido?
  - Detectar que a mensagem foi perdida
  - Aguardar pelo tempo de expiração (*timeout*)
  - Enviar um novo pedido

Comunicação confiável ponto-a-ponto: cliente-servidor



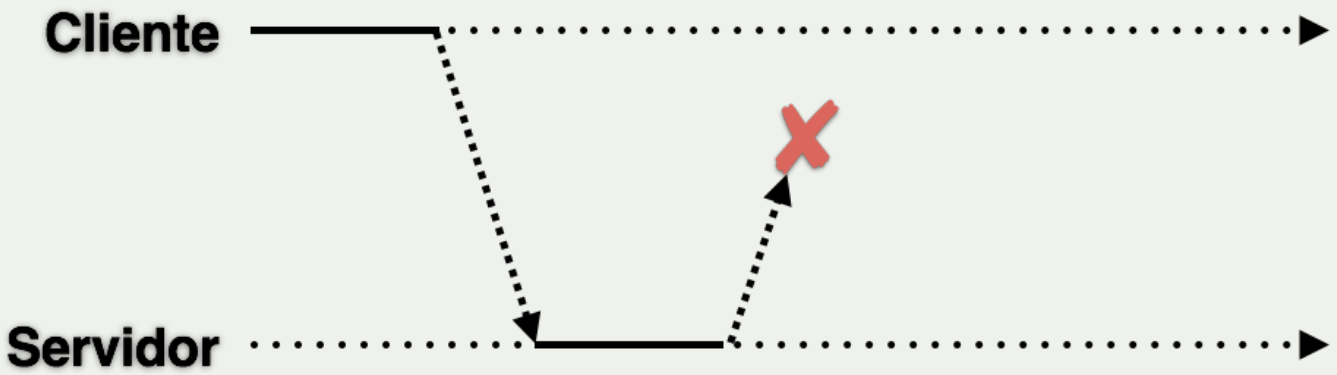
- O que fazer se a resposta for perdida?
  - Cliente aguardará pelo tempo de expiração e enviará um novo pedido

Comunicação confiável ponto-a-ponto: cliente-servidor



Uso do TCP seria suficiente para garantir comunicação confiável?

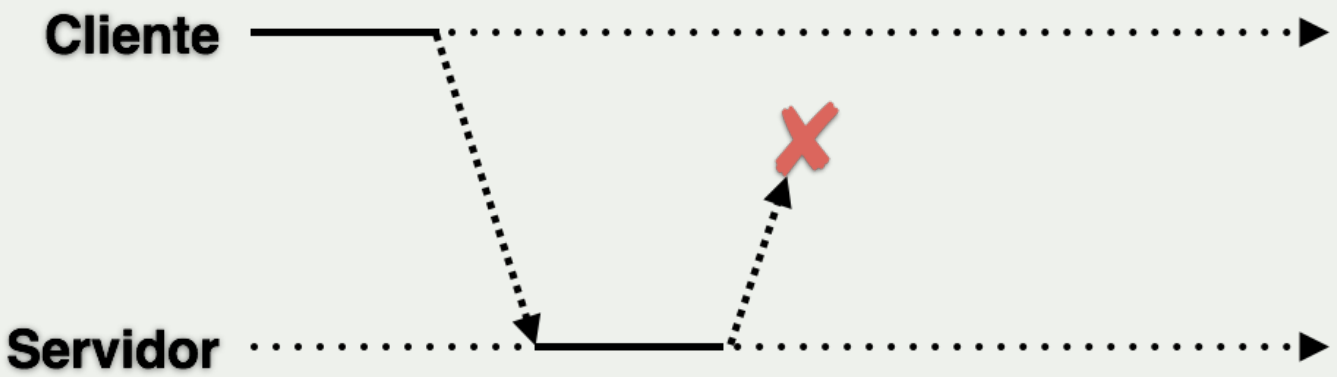
Comunicação confiável ponto-a-ponto: cliente-servidor



Uso do TCP seria suficiente para garantir comunicação confiável?

- TCP mascara falhas por omissão (uso de *ack* e retransmissões)
- TCP não mascara falhas por queda, quando a conexão é interrompida abruptamente

Comunicação confiável ponto-a-ponto: cliente-servidor

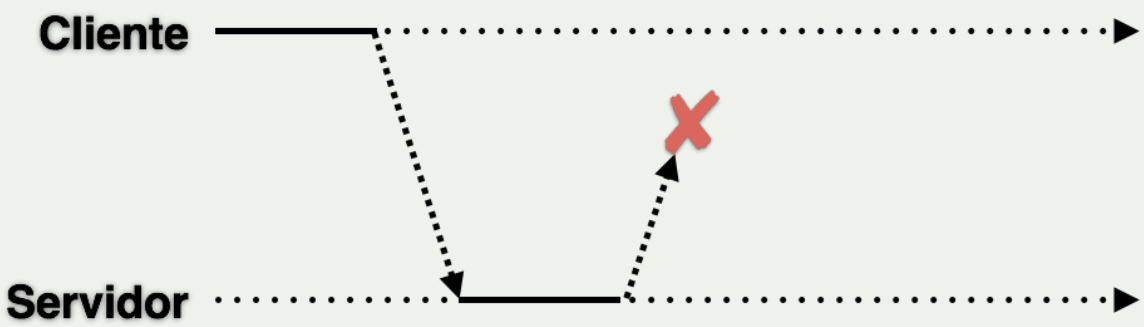


Uso do TCP seria suficiente para garantir comunicação confiável?

- TCP mascara falhas por omissão (uso de *ack* e retransmissões)
- TCP não mascara falhas por queda, quando a conexão é interrompida abruptamente

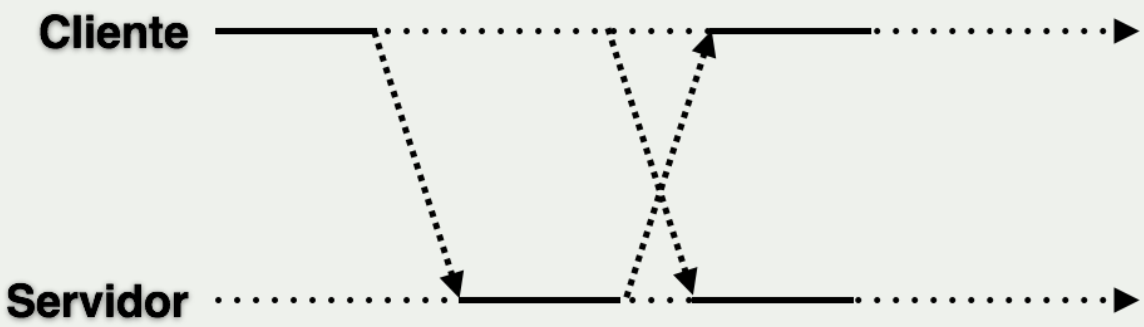
Um sistema distribuído pode mascarar falhas de queda tentando estabelecer uma nova conexão

Comunicação confiável ponto-a-ponto: cliente-servidor



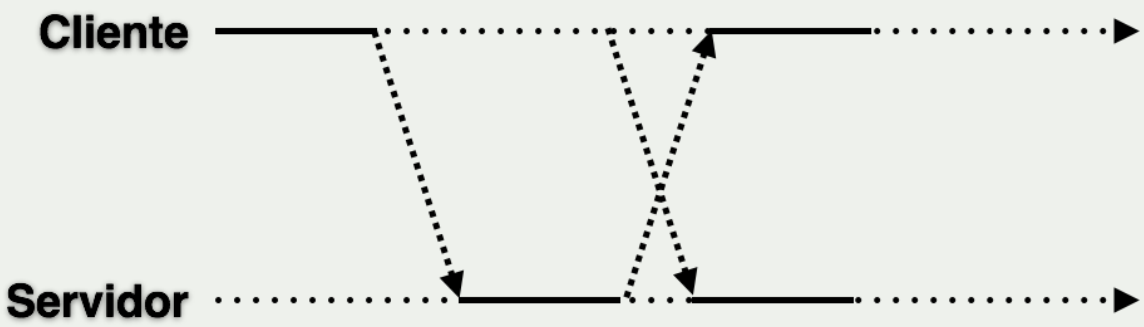
- Servidor demorou muito para enviar resposta e cliente reenviou o pedido

Comunicação confiável ponto-a-ponto: cliente-servidor



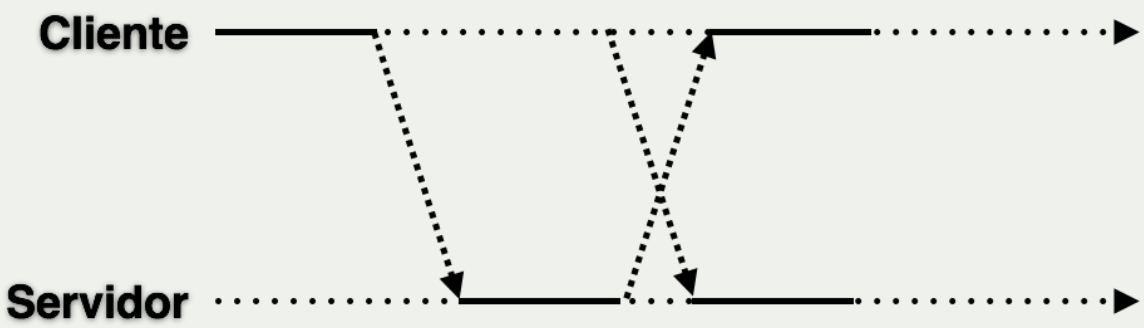
- Servidor demorou muito para enviar resposta e cliente reenviou o pedido

Comunicação confiável ponto-a-ponto: cliente-servidor



- Servidor demorou muito para enviar resposta e cliente reenviou o pedido
  - Comportamento apropriado somente para **requisições idempotentes**

Comunicação confiável ponto-a-ponto: cliente-servidor

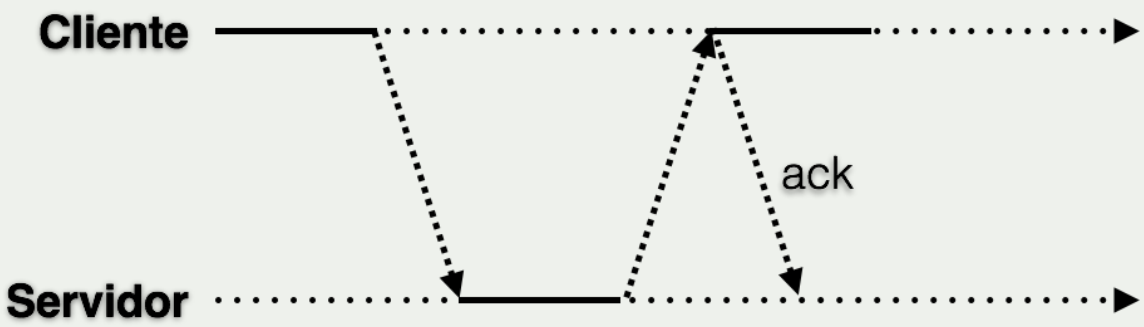


- Servidor demorou muito para enviar resposta e cliente reenviou o pedido
  - Comportamento apropriado somente para **requisições idempotentes**

Requisição idempotente sempre gera o mesmo resultado, mesmo se executada uma ou mais vezes

- Deixa o servidor no mesmo estado e produzindo o mesmo resultado
- Ex: HTTP GET, leitura em um servidor de arquivos, etc

Comunicação confiável ponto-a-ponto: cliente-servidor



- Servidor poderia manter um histórico com todos os pedidos anteriores ou mesmo exigir confirmação de entrega (ACK)
  - Adequado para **requisições não idempotentes**

Comunicação confiável: Semânticas para requisições remotas

Se uma operação ocorreu com sucesso...

No máximo uma vez (at-most-once)

- Se o cliente recebeu resposta, então o **pedido** foi **processado uma única vez**
- Pode ser implementado por meio de histórico de mensagens no servidor (filtrar mensagens repetidas) ou simplesmente não reenviar os pedidos

Comunicação confiável: Semânticas para requisições remotas

Se uma operação ocorreu com sucesso...

Ao menos uma vez (at-least-once)

- Se o cliente recebeu resposta, então o **pedido** foi **processado no mínimo uma vez**, mas pode ser sido processado mais vezes
- Não é necessário histórico no servidor, bastaria reenviar pedidos até receber uma resposta

Comunicação confiável: Semânticas para requisições remotas

- **No máximo uma vez** (at-most-once) – com reenvio de mensagens
  - Simples de implementar, porém não é tolerante a faltas
- **No máximo uma vez** (at-most-once) – com histórico
  - Um pouco mais complexo para implementar, porém tolerante a faltas
- **Ao menos uma vez** (at-least-once)
  - Simples para implementar e tolerante a faltas

Comunicação confiável: Semânticas para requisições remotas

- Uma chamada de procedimento local segue a semântica **exactly once**
- Maioria das implementações de RPC oferecem somente uma semântica
  - **at-least-once** ou **at-most-once**
- É desejado projetar uma aplicação para idempotente e sem manutenção de estado (*stateless*)

Alguns pontos sobre a comunicação cliente/servidor

- **Confiável x não confiável**
  - Canal não confiável: exige confirmação de todas mensagens (pedido e resposta) – aplicação fica responsável
  - Canal confiável: a resposta pode atuar como uma confirmação do pedido
  - Comunicação confiável pode delegar para os mecanismos de transporte lidarem com mensagens perdidas

Receber (pull) ou enviar (push)

- **Cliente: modelo pull (receber)**
  - Cliente responsável por obter dados do servidor (envia pedido)
    - Ex: HTTP
  - Vantagem: servidor não precisa manter estado
  - Dificuldade: escalabilidade limitada
- **Servidor: modelo push (enviar)**
  - Servidor envia dados para o cliente
    - Ex: Streaming de vídeo
  - Vantagem: mais escalável
  - Dificuldade: servidor precisa manter estado

Comunicação em grupo: um para muitos

- **Características de um grupo**
  - estáticos ou dinâmicos
  - abertos ou fechados
- **Endereçamento de um grupo**
  - Multicast, broadcast
  - Multicast na camada da aplicação (unicast)
- **Atomicidade, ordenação de mensagens e escalabilidade**

Exercício 02

1. Servidores que implementam a semântica *at-least-once* podem ser considerados como *stateless*? Justifique sua resposta
2. Defina a semântica **maybe**
3. Faça um pseudo-código de uma comunicação com a semântica at-least-once (cliente e servidor)
4. Faça um pseudo-código de uma comunicação com a semântica at-most-once (cliente e servidor)
5. Java RMI adota qual semântica para requisições: at-most-once, at-least-once ou ambas?