Resolução

Geral

1. Questão sobre processamento paralelo e distribuído:
    I. (V) A computação paralela realmente usa vários processadores para resolver problemas rapidamente, dividindo-os em tarefas menores. É como se cada processador ficasse com uma parte do problema para resolver, o que acelera sua resolução.
    II. (F) A memória não é compartilhada em ambientes de processadores distribuídos com **acoplamento fraco**. Em sistemas distribuídos, **cada processador tem sua própria memória** e se comunica por mensagens.
    III. (F) Conhecer a arquitetura de comunicação é essencial na programação paralela. Saber como os processadores se comunicam ajuda a evitar gargalos e permite usar os recursos de forma eficiente.
    IV. (F) Grids computacionais podem estar distribuídos via Web. Um grid pode incluir computadores conectados pela Internet.
    V. (F) Um sistema distribuído **fortemente acoplado** usa **memória compartilhada** e **não é formado por computadores independentes**. Em sistemas fortemente acoplados, os processadores geralmente compartilham memória física.

2. Questão sobre a evolução da arquitetura de software:
    I. (V) A separação dos sistemas em 3 camadas lógicas (banco de dados, regras de negócio e interface), em contraste com a única camada utilizada para tudo até então, realmente torna o sistema mais complexo, o que requer profissionais especializados.
    II. (V) Essa separação em 3 camadas torna os sistemas mais flexíveis, permitindo modificar cada camada independentemente. Isto é, a mudança em uma camada não afeta outra mais.
    III. (F) A separação em 3 camadas na verdade reduziu o acoplamento já que agora cada camada é independente, podendo ser atualizada sem impactar diretamente as outras, o que facilita e muito a manutenção de código.

3. Questão sobre sistema de informação distribuído:
    I. (F) A distribuição de tarefas em um sistema distribuído não depende que o usuário indique qual servidor deve atender sua requisição. É o sistema quem vai determinar para onde enviar as tarefas/requisições. Lembrete: um dos princípios de sistemas distribuídos é a transparência, portanto, o usuário não tem ideia qual servidor será utilizado.
    II. (F) Depende da arquitetura. Em uma arquitetura cliente-servidor, de fato, o servidor é dedicado a atender pedidos dos clientes e o cliente tem

função exclusivamente requisitante. Porém, ao considerar uma arquitetura P2P, um dispositivo funciona como cliente e servidor ao mesmo tempo. Concluindo, não é uma regra que pode se afirmar sempre.

III. (F) Depende da arquitetura. Se for uma arquitetura P2P, está certo, todos os computadores executam tarefas de cliente e servidor. Entretanto, se for uma arquitetura cliente-servidor, um computador só pode ser um dos dois, cliente ou servidor. Concluindo, não é uma regra que pode se afirmar sempre.

IV. (F) O conceito apresentado trata-se da transparência de localização, e não de acesso. A transparência de acesso oculta diferenças na representação de dados e no modo de acesso a um recurso. Já a transparência de localização oculta o lugar em que um recurso está localizado.

V. (V) Isso é possível através de tecnologias como RMI (Remote Method Invocation).

4. Questão sobre serviços em nuvem:
   I. Os requisitos da plataforma do cliente mudam da seguinte forma: com a nuvem, o cliente passa a precisar apenas de um navegador e uma conexão estável com a Internet, reduzindo a dependência de um hardware potente.
   II. Esse tipo de serviço em nuvem apresenta melhor disponibilidade e menor risco de perda de dados porque a nuvem utiliza redundância, replicando os dados em vários servidores, assim garantindo um acesso contínuo e proteção contra falhas locais.
   III. Com a adoção de serviços em nuvem temos os seguintes benefícios: menor custo de infraestrutura, acessibilidade de qualquer lugar, escalabilidade fácil e backups automáticos.

RPC/RMI

1. Questão sobre RMI
   I. Primeiramente, a IDL (Interface Definition Language) é usada para descrever as interfaces remotas de forma padronizada e independente da linguagem de programação. Desta forma, podemos concluir que a utilização da IDL é para garantir que sistemas ou linguagens possam se comunicar corretamente, definindo claramente os métodos disponíveis, seus parâmetros e tipos de retorno, mesmo que sejam linguagens diferentes.
   II. A descrição de interface remota define quais métodos podem ser invocados remotamente. Já o stub é um objeto gerado com base nessa descrição de interface e age como um proxy do lado do cliente. O cliente chama o método no stub como se fosse local, o stub encapsula

essa chamada e envia pela rede ao servidor. No servidor, o método real é executado.

III. O stub tem sua implementação baseada no padrão de projeto proxy porque este padrão é utilizado para criar um objeto que representa outro. E é justamente isso que acontece no stub pois ele torna a invocação remota quase idêntica a uma chamada local, tanto é que o cliente chama o método no stub como se fosse local. Em seguida, o stub intercepta essa chamada, a encapsula e envia ao servidor, onde o método real é executado. O método criado no stub é uma representação do método presente no servidor.

IV. O serviço de nomes funciona como um catálogo de serviços em que são associados nomes à referências de objetos remotos, para que assim os clientes possam localizá-los e executar métodos remotamente. Como ele funciona:
    - O servidor registra o objeto remoto no serviço de nomes, associando-o a um identificador (nome)
    - O cliente consulta o serviço de nomes para obter a localização exata do objeto remoto (referência remota do objeto) a partir do seu nome
    - Com a localização exata em mãos, o cliente consegue invocar métodos no servidor

    Dessa forma, o seu uso é justificado pelo fato do cliente não precisar saber o endereço IP nem a porta do servidor, apenas o nome do objeto.

2. Questão sobre aplicação com objetos distribuídos
    I. O stub é o representante do objeto remoto do lado do cliente responsável por encapsular detalhes da comunicação remota, enviar as chamadas de método do cliente para o servidor e tornar a chamada ao método transparente para o cliente, parecendo uma chamada local na sua visão. Já o skeleton é o representante do objeto remoto do lado do servidor responsável por receber as chamadas do stub e delegá-las ao método real do objeto remoto. Ambos são considerados proxy porque atuam como representantes do objeto remoto.

    II. O Módulo de Referência Remota é responsável por gerenciar todas as referências aos objetos remotos em um sistema RMI. Age como uma ponte entre o cliente e o servidor, garantindo que as chamadas sejam corretamente roteadas. Garante que o stub use a conexão correta para alcançar o skeleton no servidor.

    III. A referência ao objeto é um identificador único que permite localizar e acessar um objeto remoto no servidor (tem o papel de localizar exatamente onde está aquele objeto remoto no servidor), permitindo que o cliente invoque métodos como se o objeto estivesse local. Ela é composta pelo endereço do servidor, identificador do objeto, protocolo de comunicação e estado da conexão.

3. As 3 semânticas de invocação do RPC são:
    a) **Maybe**
        - Descrição → O método pode ser executado ou não no servidor. Não há garantia de que a chamada ao método foi recebida e processada.
        - Como funciona → O cliente envia a requisição, mas não espera por uma confirmação ou resposta.
        - Modelo de falhas → Não lida com falhas. Se a mensagem se perder ou o servidor falhar, não há retransmissão.
        - Uso típico → Cenários em que a confirmação do servidor não é crítica, como notificações informais.

    b) **At-least-once**
        - Descrição → O método será executado pelo menos uma vez no servidor. Porém, pode ser executado mais de uma vez se houver retransmissão.
        - Como funciona → O cliente retransmite a requisição até obter uma confirmação ou resposta do servidor.
        - Modelo de falhas → Tolera falhas de comunicação, como mensagens perdidas, mas pode causar duplicação de execução do método no servidor.
        - Uso típico → Aplicações onde é preferível garantir a execução do método, mesmo que isso resulte em duplicações, como atualização de um banco de dados.

    c) **Exactly-one**
        - Descrição → O método será executado exatamente uma vez no servidor, garantindo idempotência (não muda o estado do programa mesmo se repetir a operação várias vezes). Mesmo se o cliente retransmitir a mensagem várias vezes, o servidor processará apenas uma vez.
        - Como funciona → O cliente retransmite a requisição várias vezes até obter uma confirmação ou resposta do servidor. A diferença é que aqui o servidor possui mecanismos para evitar duplicações, como identificadores únicos para cada chamada.
        - Modelo de falhas → Mais robusto, tolera falhas de comunicação e evita duplicações.
        - Uso típico → Sistemas financeiros e transacionais, onde duplicações podem causar inconsistências graves.

4. Sim, uma aplicação distribuída desenvolvida com base na distribuição de processos é diferente de uma construída com base na distribuição de objetos.

Distribuição de processos: A lógica da aplicação é distribuída entre diferentes processos (programas independentes) que se comunicam entre si. A comunicação neste caso é via mensagens (sockets, filas etc) e cada processo é independente, podendo estar em máquinas diferentes.

- Vantagens → Flexibilidade para usar diferentes linguagens em processos distintos e alta escalabilidade visto que novos processos podem ser adicionados facilmente.
- Desvantagens → Maior esforço para coordenar os processos e comunicação explícita, precisando que os desenvolvedores gerenciem a troca de mensagens.

Distribuição de objetos: Objetos remotos encapsulam dados e métodos, permitindo que os clientes chamem esses métodos como se fossem locais (RMI). A comunicação aqui ocorre por invocação remota de métodos e faz o uso de stubs e skeletons para abstrair a complexidade.

- Vantagens → Transparência (os métodos remotos parecem locais para o cliente), redução da complexidade para os desenvolvedores e reutilização de objetos remotos por múltiplos clientes.
- Desvantagens → Maior overhead em comparação a mensagens diretas e menor flexibilidade para usar diferentes tecnologias e linguagens em objetos.

Resumidamente, a distribuição de processos é mais adequada para sistemas heterogêneos ou quando cada componente do sistema utiliza tecnologias/linguagens diferentes. Já a distribuição de objetos é ideal para aplicações monolíticas distribuídas, onde a transparência e reutilização são mais importantes.

5. Questão sobre RPC
   I. **Vantagens quando comparado ao sockets:**
      - Transparência → O RPC oculta detalhes de comunicação da rede, permitindo que o procedimento seja chamado como se fosse local.
      - Simplicidade → Não tem a complexidade de implementar e gerenciar sockets, como a questão da serialização e deserialização de dados.
      - Reusabilidade → O mesmo procedimento pode ser chamado por vários clientes diferentes.

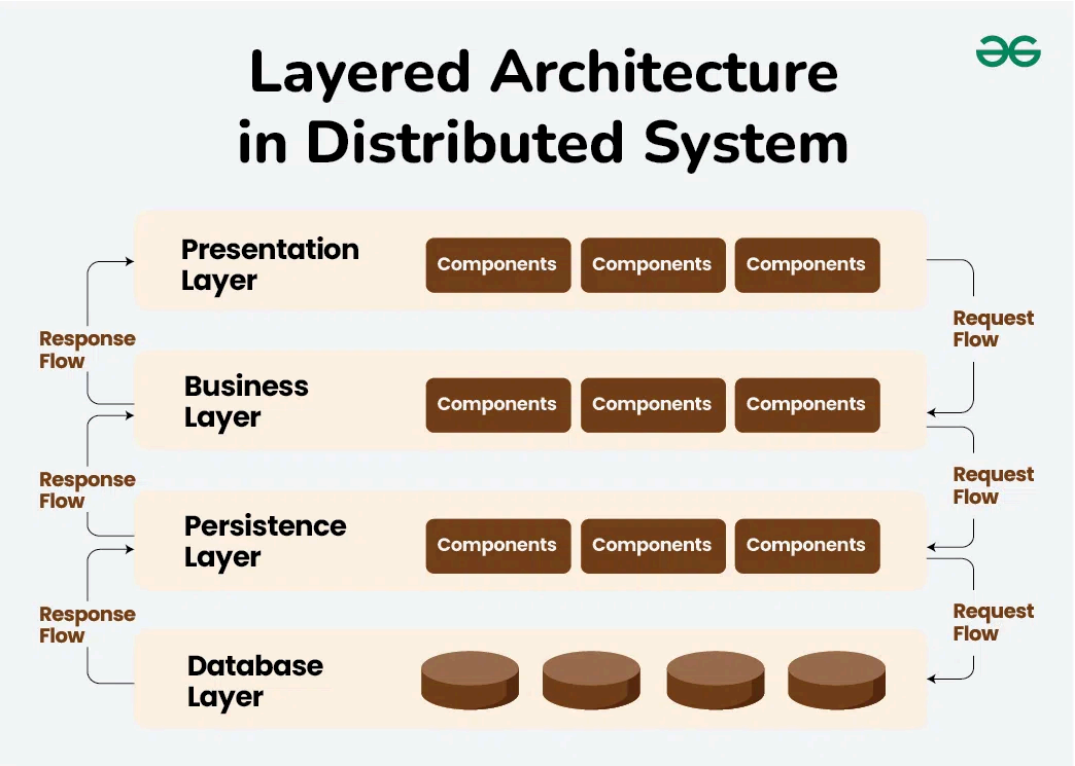
      Desvantagens:
      - Overhead → O RPC introduz um overhead devido à abstração de complexidade que faz, tornando-o mais lento que o uso direto de sockets.

- Menor flexibilidade → As opções de controle sobre a comunicação, como formatos de mensagens, são limitadas quando comparado ao uso de sockets.

II. **Vantagens quando comparado ao sockets:**
    - Transparência → O RMI permite que métodos sejam invocados remotamente em objetos como se fossem locais.
    - Reusabilidade → O mesmo método pode ser invocado por diversos clientes diferentes.
    - Suporte ao modelo de OO → É projetado para trabalhar com objetos e toda a questão da orientação a objetos.
    - Simplicidade → Reduz o esforço do desenvolvedor porque já fornece uma estrutura pronta para criar, registrar e invocar métodos.
    - O cliente não precisa saber o endereço IP nem a porta do servidor para executar o método, basta ter o nome do objeto remoto (quem guarda essas informações de localização é o serviço de nomes).

    Vantagens quando comparado ao RPC:
    - O cliente não precisa saber o endereço IP nem a porta do servidor para executar o método, basta ter o nome do objeto remoto (quem guarda essas informações de localização é o serviço de nomes).
    - O RMI permite usar objetos remotos completamente, incluindo estado e comportamento, além de aproveitar todas as questões relacionadas a OO, como herança e polimorfismo. Enquanto isso, o RPC se limita a procedimentos.
    - Oferece uma maior aderência ao paradigma orientado a objetos, diferente do RPC que segue um modelo procedural.

6. Padrão arquitetural “**Layers**”:
    - O sistema é dividido em camadas hierárquicas, onde cada camada tem uma responsabilidade específica e interage apenas com as camadas adjacentes. Exemplo: camada de apresentação, lógica de negócios e dados.



Pseudocódigo:

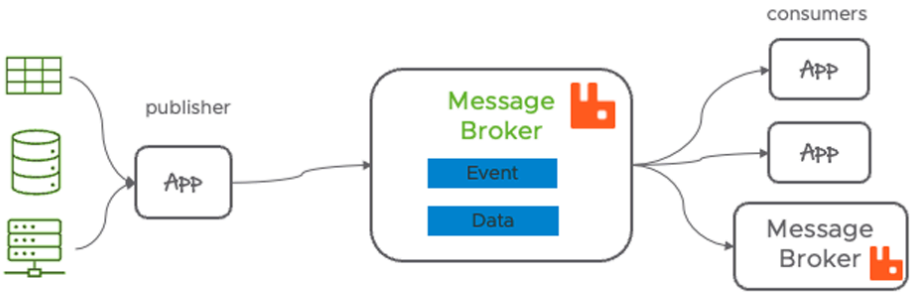
```
class Apresentacao:
    def executar():
        resultado = Logica.processar()
        return resultado

class Logica:
    def processar():
        dados = Dados.obter()
        return dados * 2

class Dados:
    def obter():
        return 10
```

Padrão arquitetural “**Broker**”:

- Um intermediário (broker) coordena a comunicação entre clientes e servidores, desacoplando as partes e fazendo com que elas se comuniquem de forma indireta. Exemplo: middleware com RabbitMQ ou Kafka.



Pseudocódigo:

```
# Broker atua como intermediário
class Broker:
    def __init__(self):
        self.fila = []

    def publicar(self, mensagem):
        # Producer envia uma mensagem ao broker
        print(f"Broker: Mensagem recebida - {mensagem}")
        self.fila.append(mensagem)

    def consumir(self):
        # Consumer consome a mensagem do broker
        if self.fila:
            mensagem = self.fila.pop(0)
            print(f"Broker: Mensagem entregue ao consumer - {mensagem}")
            return mensagem
        else:
            print("Broker: Nenhuma mensagem na fila.")
            return None

# Producer cria mensagens e as publica no broker
class Producer:
    def produzir(self, broker, mensagem):
        print(f"Producer: Enviando mensagem - {mensagem}")
        broker.publicar(mensagem)

# Consumer consome mensagens processadas pelo broker
class Consumer:
    def consumir(self, broker):
        mensagem = broker.consumir()
        if mensagem:
```

```
            print(f"Consumer: Processando mensagem - {mensagem}")
        else:
            print("Consumer: Nada para processar.")

# Exemplo de execução
if __name__ == "__main__":
    broker = Broker()
    producer = Producer()
    consumer = Consumer()

    # Producer publica mensagens no broker
    producer.produzir(broker, "Mensagem 1")
    producer.produzir(broker, "Mensagem 2")

    # Consumer consome mensagens do broker
    consumer.consumir(broker)
    consumer.consumir(broker)
    consumer.consumir(broker)  # Nenhuma mensagem para consumir
```

7. As classes de falha que podem surgir com essa invocação são:
    - **Falha de transmissão →** A mensagem de requisição enviada de A para B pode ser perdida na rede ou corrompida antes de chegar no servidor.
    - **Falha de processamento →** O servidor B pode falhar durante o processamento da requisição.
    - **Falha na resposta →** A mensagem de resposta de B para A pode ser perdida na rede ou corrompida antes de chegar no cliente.
    - **Falha na rede →** Pode ser que a rede esteja congestionada ou com falhas, atrasando a comunicação entre A e B, e consequentemente causando um timeout no cliente.

8. **Localidade de objetos:** Em sistemas distribuídos, os objetos podem estar em diferentes máquinas, enquanto no modelo básico os os objetos estão sempre locais. Dessa forma, no modelo distribuído é possível que o cliente invoque métodos de um objeto remoto como se ele estivesse local, ocultando detalhes da localização. Resumidamente, um cliente em uma máquina A pode invocar métodos de um objeto em uma máquina B sem precisar saber onde ele está ou como alcançá-lo.

   Chamada de métodos: No modelo distribuído, os métodos são invocados remotamente usando mecanismos como RMI (no modelo básico os métodos são chamados diretamente e executados na mesma máquina). A invocação ao método parece local para o cliente, mas envolve comunicação em rede até chegar no servidor que contém de fato o objeto remoto.

Referência: No modelo básico, uma referência se trata de um ponteiro para uma região da memória local. Contudo, em um sistema distribuído, uma referência encapsula as informações necessárias para localizar e acessar o objeto. Isto é, a referência é o que permite localizar exatamente onde está o objeto remoto no servidor.

Interface: Em ambos os modelos, a interface vai definir quais métodos estão disponíveis para os clientes. Só que no modelo distribuído essa interface deve ser projetada para uso remoto e especificada em uma IDL a fim de garantir interoperabilidade entre sistemas diferentes (IDL é usada para descrever as interfaces remotas de forma padronizada e independente da linguagem de programação).

Pub/Sub

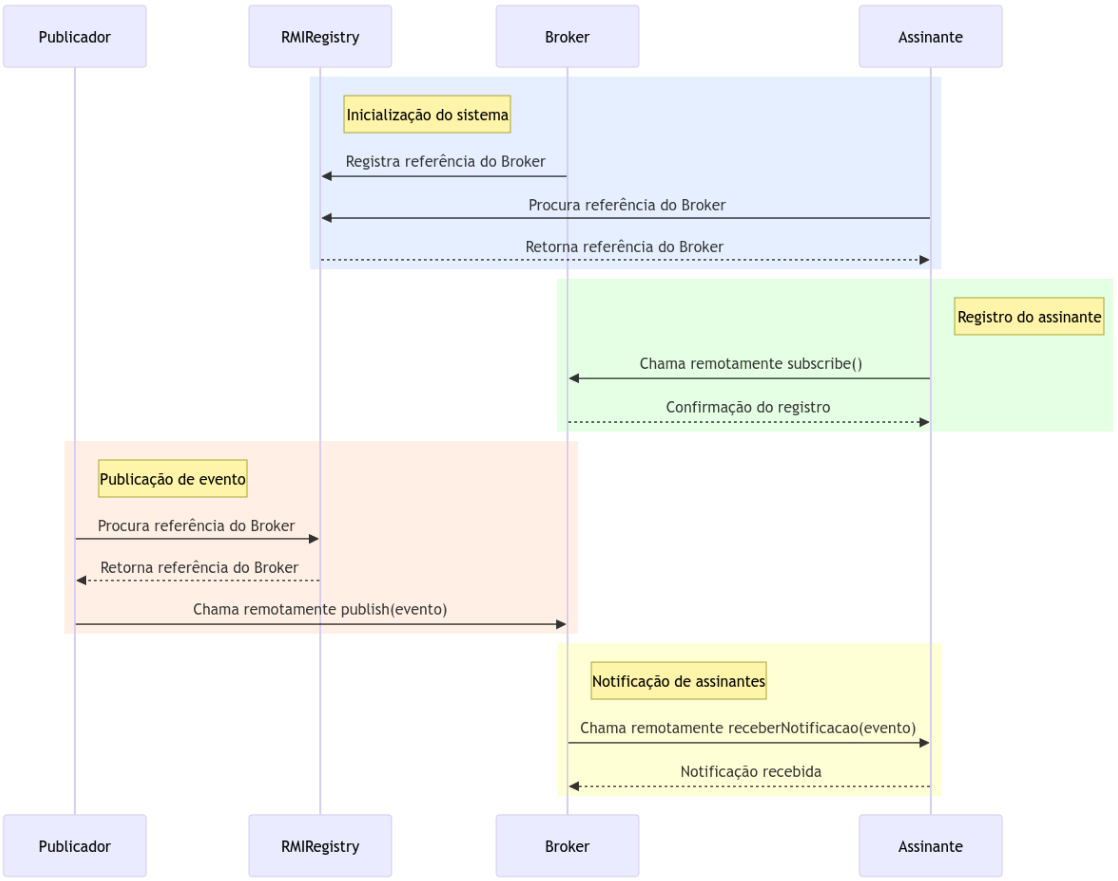
1. Questão sobre arquitetura para notificação de eventos distribuídos:

   Pub/Sub: Temos publicadores (quem publica os eventos) e assinantes (quem consome os eventos). Os publicadores enviam notificações de eventos para um elemento intermediário, o broker, que é responsável por distribuir essas notificações aos assinantes interessados.
   ECA (Event-Condition-Action): Trata-se de uma regra que descreve que, quando um evento ocorre, uma condição é avaliada. Se a condição for verdadeira, uma ação é executada. Por exemplo, imagine que o evento é o envio de um novo arquivo ao sistema e a condição é o tamanho do arquivo ser maior que 10MB. Se o tamanho do arquivo enviado atender à condição, então o move para uma pasta específica (ação).
   RMI (Remote Method Invocation): O RMI permite a invocação remota de métodos de objetos que estão localizados em outros computadores. Esses métodos são chamados como se os objetos estivessem na mesma máquina (do usuário, como se fosse uma chamada local).

   Como funcionaria os três juntos?

   - O broker registra sua referência no RMIRegistry quando iniciado para que os publicadores e assinantes possam encontrá-lo
   - O assinante busca a referência do broker no RMIRegistry
   - O assinante se registra no broker chamando o método remoto *subscribe()*
   - O publicador busca a referência do broker no RMIRegistry
   - O publicador invoca remotamente o método *publish(evento)* no broker, enviando os dados do evento

- O broker verifica as condições (ECA) e, caso o evento seja relevante para os assinantes registrados, invoca remotamente o método *receberNotificacao(evento)* para cada assinante



2. **Lembrete:** Indireção é quando tem um elemento intermediário entre o emissor (publicador) e receptor (assinante).

   A indireção pode ser implementada em sistemas distribuídos utilizando:
   - **Broker no modelo pub/sub →** O broker atua como um intermediário responsável por distribuir os eventos para os assinantes.
   - **Mensageria (fila de mensagens) →** Armazena as mensagens enviadas pelos publicadores em filas e as distribui para os assinantes, como RabbitMQ ou Kafka.

   As duas propriedades decorrentes do uso da indireção são:
   - **Desacoplamento no espaço →** O remetente não precisa conhecer a identidade do destinatário. Isto é, os publicadores e assinantes não precisam saber da existência uns dos outros, precisam conhecer apenas o elemento intermediário.

- **Desacoplamento no tempo →** O remetente e o destinatário podem ter tempos de vida independentes, isto é, os dois não precisam estar ativos ao mesmo tempo. O publicador pode enviar eventos mesmo que o assinante não esteja ativo na hora. O elemento intermediário garante a entrega posterior ao destinatário.

3. Um modelo ser assíncrono não implica necessariamente que ele seja desacoplado no tempo. Trata-se de dois conceitos diferentes.
    - **Exemplo de um caso assíncrono, mas acoplado no tempo:** Um servidor envia mensagens ao cliente por meio de sockets não-bloqueantes. Embora a comunicação seja assíncrona, o cliente precisa estar ativo para receber as mensagens.
    - **Exemplo de um caso assíncrono e desacoplado no tempo:** No modelo pub/sub com um broker, o publicador envia os eventos para o broker. O assinante pode receber os eventos muito tempo depois, mesmo que estivesse offline no momento da publicação, pois o broker é capaz de armazenar esses eventos e entregá-los posteriormente (quando o assinante estiver ativo).

4. ⚠ **Para mim essa questão é igual à questão 2 de pub/sub, só está escrita com outras palavras.**

Algoritmos

1. O algoritmo de exclusão mútua distribuída multicast utiliza relógios lógicos, como o relógio de Lamport, para garantir a ordenação dos pedidos de entrada na região crítica, assim evitando conflitos entre processos. Primeiramente, todo processo tem um relógio lógico e, quando um evento de envio ocorre, esse relógio (timestamp) é enviado junto à mensagem. O processo receptor, ao receber a mensagem, vai ajustar seu relógio lógico para o maior valor entre seu próprio relógio e o relógio recebido acrescido de 1 unidade. Dessa forma, garante-se uma ordem lógica global. Nesse algoritmo que utiliza multicast e relógios lógicos, quando um processo quer acessar a região crítica, ele vai enviar uma requisição multicast para todos os processos avisando seu desejo. Juntamente com a requisição é enviado o seu timestamp (relógio lógico). Os outros processos vão responder ao pedido e também utilizarão os timestamps para decidir qual será a ordem das requisições/acesso à região crítica. Aquele processo requisitante só vai conseguir acessar a seção crítica se obter uma resposta de aprovação de todos os outros processos. Caso contrário, seja porque tem alguém que pediu antes ou já tenha alguém na seção, a requisição será enfileirada. Desse modo, a exclusão mútua sempre vai atender o processo com menor timestamp entre as requisições, assegurando que esses eventos sejam processados de uma forma causal (cronológica). Ou seja, pedidos de

entrada na região crítica devem ser atendidos seguindo o timestamp das requisições.

2. O algoritmo de subscription flooding, como o próprio nome sugere, a rede é inundada com uma grande quantidade de assinaturas. Ou seja, é estressada com as assinaturas. Nessa abordagem todos os brokers da rede possuem uma tabela de assinaturas cujo objetivo é ajudar e otimizar a propagação das mensagens publicadas, indo direto para os assinantes interessados naquela mensagem. Esse algoritmo funciona da seguinte forma:
    - Quando um cliente (assinante) demonstra interesse em um determinado tópico (tipo de mensagem), ele envia sua assinatura para o broker local (mais próximo dele, de borda) e este atualiza sua tabela
    - O broker local irá propagar essa assinatura para os brokers adjacentes e assim por diante, formando uma inundação de mensagens até que todos os brokers da rede estejam cientes sobre aquela assinatura (atualizem suas tabelas)
    - Depois de todas as tabelas de assinaturas serem atualizadas, as mensagens publicadas são roteadas diretamente apenas para os brokers de borda que possuem assinantes interessados naquele tipo de mensagem

    Essas tabelas vão ser estruturadas associando um determinado tópico com os clientes que o assinaram, quem têm interesse nele. Além disso, elas podem ser otimizadas usando estruturas de dados como listas invertidas ou árvores de disseminação.

5. O algoritmo de exclusão mútua de Maekawa e o algoritmo baseado em multicast e relógios lógicos se diferem em como eles controlam o acesso à região crítica e quanto à quantidade de mensagens trocadas. No algoritmo de Maekawa cada processo pertence a um grupo de votação chamado quorum e, para acessar o recurso na região crítica, basta pedir e obter a aprovação de todos os processos participantes **do seu grupo**. Assim, o processo que deseja acessar o recurso não precisa pedir permissão para todos os outros processos, e sim apenas para o grupo dele, que é menor. A exclusão mútua é garantida por conta da ideia de grupos sobrepostos, ou seja, com pelo menos um processo em comum, impedindo assim acesso simultâneo ao recurso porque sempre vai ter um processo que pode bloquear o pedido se necessário. Já na abordagem com multicast e relógios lógicos, toda vez que um processo quiser acessar o recurso na região crítica, ele vai enviar uma requisição multicast para **todos** os outros processos a fim de obter sua autorização. Neste caso, o acesso à região crítica sempre vai ser dado para o processo com menor timestamp entre todas as requisições, garantindo que o acesso seja concedido de forma causal/cronológica.

7. **Estado consistente** significa respeitar a relação **Happened Before** e **Relógio Lógico de Lamport**

Happened Before: Símbolo "→"

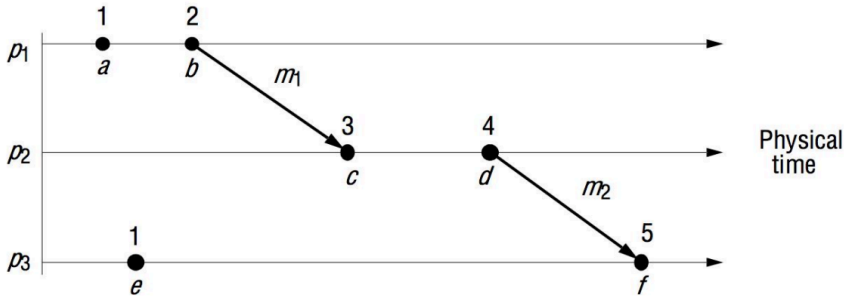
I. Se **a** e **b** são eventos do mesmo processo, e **a** vem antes de **b**, então **a → b** (o evento **a** aconteceu antes do evento **b**)
II. Se **a** é o remetente de uma mensagem em um processo p1 e **b** é o receptor dessa mensagem em um processo p2, então **a → b** (o evento de envio deve acontecer antes do evento de recebimento)
III. Transitividade: Se **a → b** e **b → c**, então **a → c**

Relógio Lógico de Lamport: Todo processo tem seu próprio relógio lógico (contador) e deve seguir as seguintes regras

I. Se ocorreu um evento **local** em um processo p1, o contador de p1 deve ser incrementado em 1
II. Se ocorreu um evento de **envio** em um processo p1, o contador de p1 deve ser incrementado em 1
III. Se ocorreu um evento de **recebimento** em um processo p1, o contador de p1 deve atualizar seu valor para o valor max(seu contador, contador recebido), incrementado em 1

Exemplo:
a → b em p1 e c → d em p2
b → c por causa de m1
d → f por causa de m2
a // e são eventos concorrentes, não dá para afirmar que **a** ocorreu antes ou depois de **e**, por exemplo

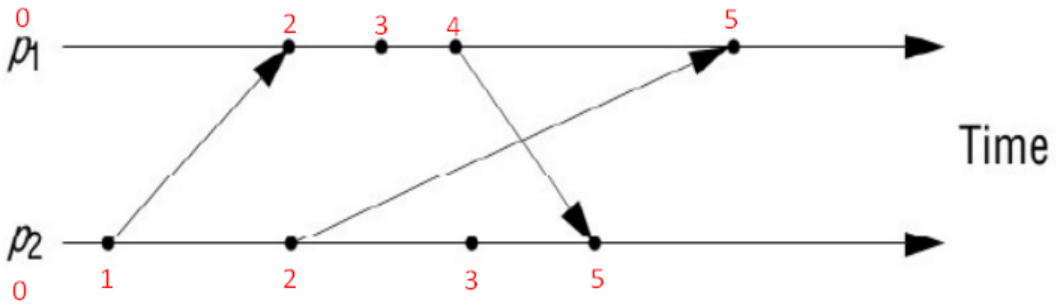
- São eventos em processos diferentes, não vale a regra I
- Não é uma relação causal, de envio e recebimento, não vale a regra II
- Não há transitividade, não vale a regra III
- Como cada processo tem seu próprio contador, pode ser que eles estejam em "taxas de crescimento" diferentes



Malha de estados consistentes para a figura da questão

Legenda: (contador de p1, contador de p2)
(0, 0) → não ocorreu evento em nenhum dos dois
(0, 1) → evento de envio em p2
(2, 1) → evento de recebimento em p1
(2, 2) → evento de envio em p2
(3, 2) → evento local em p1
(4, 2) → evento de envio em p1
(4, 3) → evento local em p2
(4, 5) → evento de recebimento em p2
(5, 5) → evento de recebimento em p1

Cada evento local/de envio incrementa o contador do respectivo processo em 1 unidade e os eventos de envio antecedem os envios de recebimento (garante a causalidade, regra II do Happened Before).



10. Primeiramente, o algoritmo de servidor central para exclusão mútua, como o próprio nome sugere, possui um único servidor que irá controlar toda a questão de exclusão mútua. Neste caso, o controle de acesso à região crítica vai ser feito através de um token e o servidor possui uma fila onde as requisições vão sendo armazenadas conforme chegam nele. Com isto já é possível notar que essa solução não garante o requisito “ordering” da exclusão mútua distribuída. Ou seja, a ordem cronológica dos pedidos de entrada na região crítica não será respeitada, até porque ele utiliza token para controlar o acesso, e não relógios lógicos. Assim, uma situação em que os pedidos não são processados na ordem happened-before é:
    - Imagine que temos dois processos, p1 e p2, querendo acessar a região crítica
    - O processo p1 envia um pedido para o servidor central às 10:00 (timestamp)
    - O processo p2 envia um pedido logo depois, às 10:01, para o servidor central
    - Por causa de atrasos na rede, o pedido de p2 chega **antes** do pedido de p1

- Assim que o pedido de p2 chega no servidor, ele já concede ao p2 o token de acesso para a região crítica
- Quando o pedido de p1 chegar, a região crítica estará ocupada por p2, então ele é colocado na fila de espera do servidor e deve aguardar que p2 libere a região, para assim conseguir o token de acesso

Como o servidor central garante o acesso à região crítica através de um token, e não por relógios lógicos, ele processa os pedidos (concede o token) na ordem em que eles vão chegando no servidor, independente do timestamp. Assim, ele processa o pedido p2 antes do p1, mesmo que, segundo a relação happened-before, o pedido p1 tenha ocorrido primeiro (e portanto, devendo ser processado primeiro).