

SISTEMAS DISTRIBUÍDOS

Capítulo 9

Sistemas de comunicação indireta

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore, T. Kindberg, G. Blair
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2011

Para saber mais:

- secção 6.1-6.4, 15.4 – apenas parte relativa aos tipos de multicast (não entrar na parte de implementação)

COMUNICAÇÃO INDIRETA

1. Na comunicação indireta, as mensagens não são endereçadas a processos de forma direta ou explícita.
2. O envio e a receção das mensagens está desacoplado, no espaço e no tempo
 - O emissor e o(s) recetor(es) podem não se conhecer;
 - O emissor e o(s) recetor(es) podem não executar em simultâneo;
3. A comunicação/interação é suportada por recurso a outras entidades
 - comunicação em grupo
 - message queues, etc.

FORMAS DE COMUNICAÇÃO : FACETAS

ponto-a-ponto (ou **unicast**) [**1x1**]

– entre um emissor e um único receptor

multi-ponto

- **broadcast** ou **difusão total** [$1 \times n$] – envio de uma mensagem de um emissor para todos os receptores
- **multicast** ou **comunicação em grupo** ou **difusão parcial** [$1 \times k \leq n$] – um emissor para um grupo de receptores
-
- **funcional** ou **anycasting** [1×1 de n] – envio de uma mensagem de um emissor para um de múltiplos possíveis receptores
- **geocast** [$1 \times k$ de n] – envio de uma mensagem de um emissor para os receptores de uma dada área geográfica

COMUNICAÇÃO MULTI-PONTO: TCP/IP

A comunicação em difusão total, parcial e difusão funcional podem ser suportados diretamente ao nível da rede TCP/IP.

ex:

broadcast: ping 255.255.255.255 ; 192.168.1.255

multicast: 224.0.0.0-239.255.255.255

anycast: endereços unicast tratados de forma especial no encaminhamento

Na Internet, anycast é implementado anunciando o mesmo endereço IP a partir de múltiplos locais.

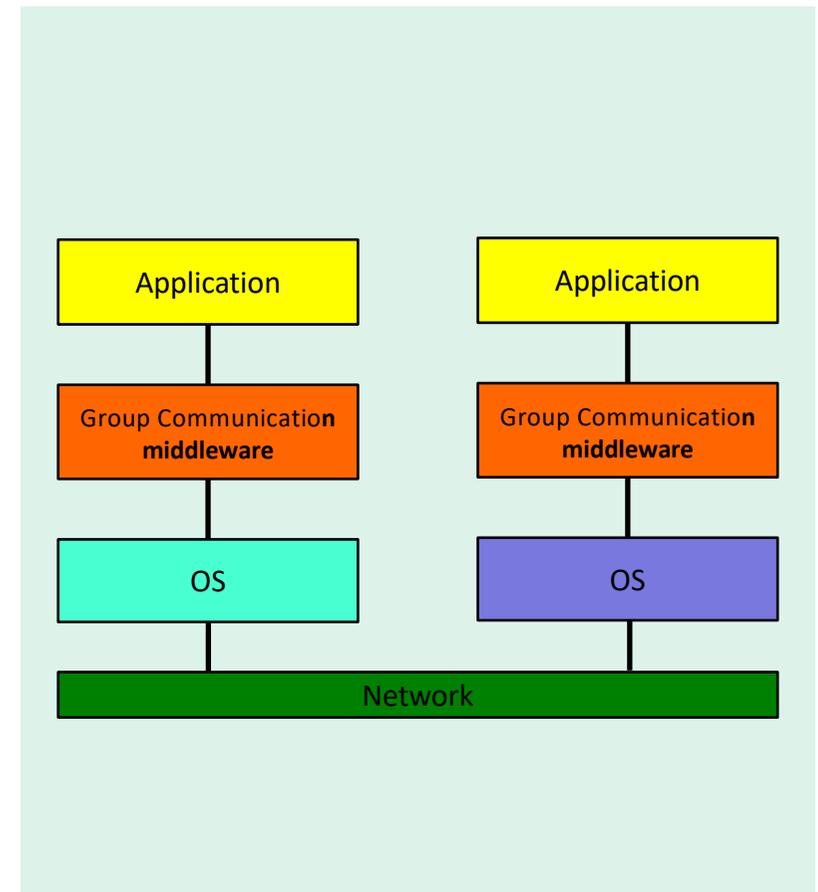
Como o encaminhamento escolhe a rota mais curta, o caminho seguido dependerá da origem

COMUNICAÇÃO EM GRUPO : IMPLEMENTAÇÃO EM MIDDLEWARE

A comunicação em grupo exposta diretamente pelo OS oferece funcionalidades limitadas:

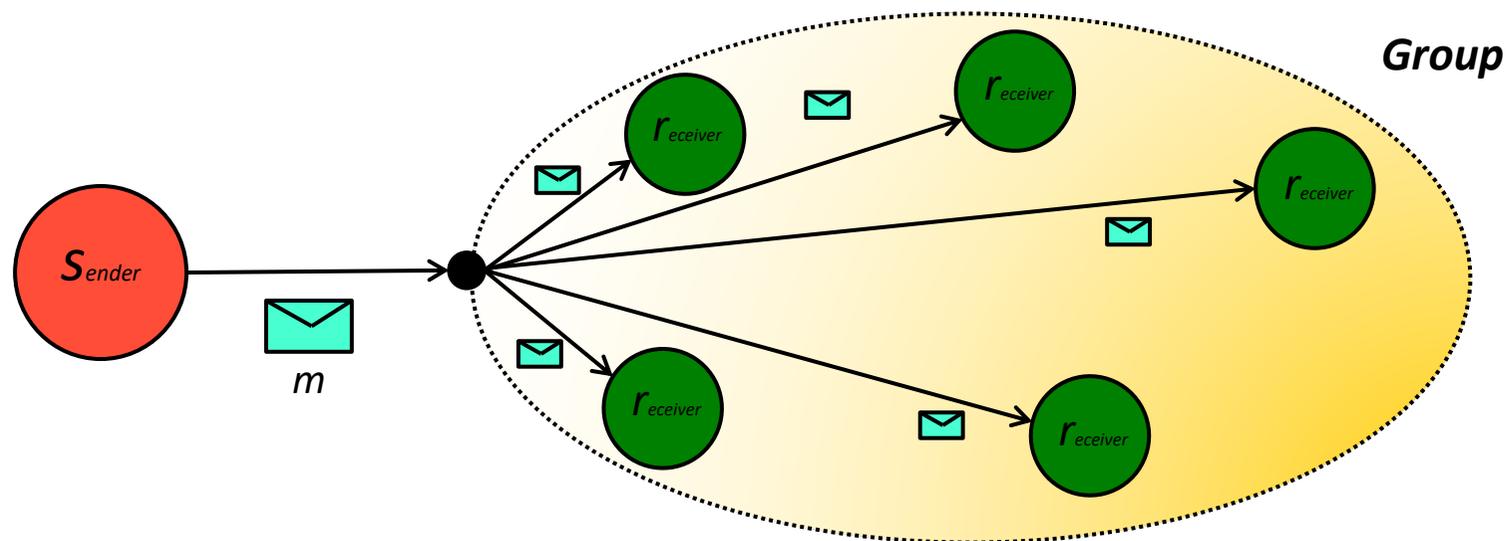
- garantias de entrega pobres;
não fiável, sem ordem, ...
- âmbito local (devido a restrições administrativas ao nível da rede)

A implementação em *middleware* dedicado é a forma usual de endereçar estas limitações.



COMUNICAÇÃO EM GRUPO : MODELO (1)

Na comunicação em grupo, o **grupo** é um conjunto **dinâmico** de *communication end-points* (portas, processos, servidores, etc.) que é tratado pelo sistema como uma entidade singular do ponto vista da comunicação.



COMUNICAÇÃO EM GRUPO : MODELO (2)

Características transversais:

- O alvo das mensagens é o grupo, designado através de um endereço/identificador **único**;
- O emissor **pode** ou **não pertencer** ao grupo;
- A composição do grupo é **dinâmica**, não necessariamente visível
- A filiação evolui de forma independente dos emissores

API

- **Típica:** send, receive, joinGroup, leaveGroup, createGroup, destroyGroup
- **Especializada:** PUB_{lish}/SUB_{scribe} (modelo editor/assinante)

AGENDA PARA HOJE

Sistemas de disseminação de eventos: pub/sub

SISTEMAS DE DISSEMINAÇÃO DE EVENTOS: PUB/SUB

Pode ser visto como um sistema comunicação em grupo, com uma interface especializada.

Modelo Genérico:

- o(s) **editor**(es)/publisher(s) **publica**(m) mensagens/eventos, usualmente organizadas por **tópicos** ou **canais** temáticos;
- o(s) **assinante**(s)/subscriber(s) manifestam interesse em receber certas mensagens, indicando o(s) **tópicos**, **canal** ou **padrão do conteúdo** das mensagens a receber
- Os sistemas pub/sub que suportam filtragem baseada no conteúdo implementam o conceito de **content-based routing**

SISTEMAS DE DISSEMINAÇÃO DE EVENTOS: PUB/SUB

Os editores não necessitam de conhecer quais ou quantos subscritores existem ou se receberam as suas mensagens

- Tal permite desacoplar o envio da recepção das mensagens/eventos;

As interações são assíncronas e podem ocorrer mesmo quando os editores e os assinantes não coexistem no tempo

PUBLISH/SUBSCRIBE: ARQUITECTURAS

Centralizadas

Existem servidores, denominados **broker(s)**

Tanto os editores como os assinantes operam como clientes do broker.

Permite implementar funcionalidades adicionais com relativa facilidade—
e.g.: persistência, filtragem baseada no conteúdo, replicação,
particionamento.

Descentralizadas (P2P)

Os editores e os subscritores organizam-se entre si para encaminhar os eventos.

PUBLISH/SUBSCRIBE: EXEMPLO APACHE KAFKA

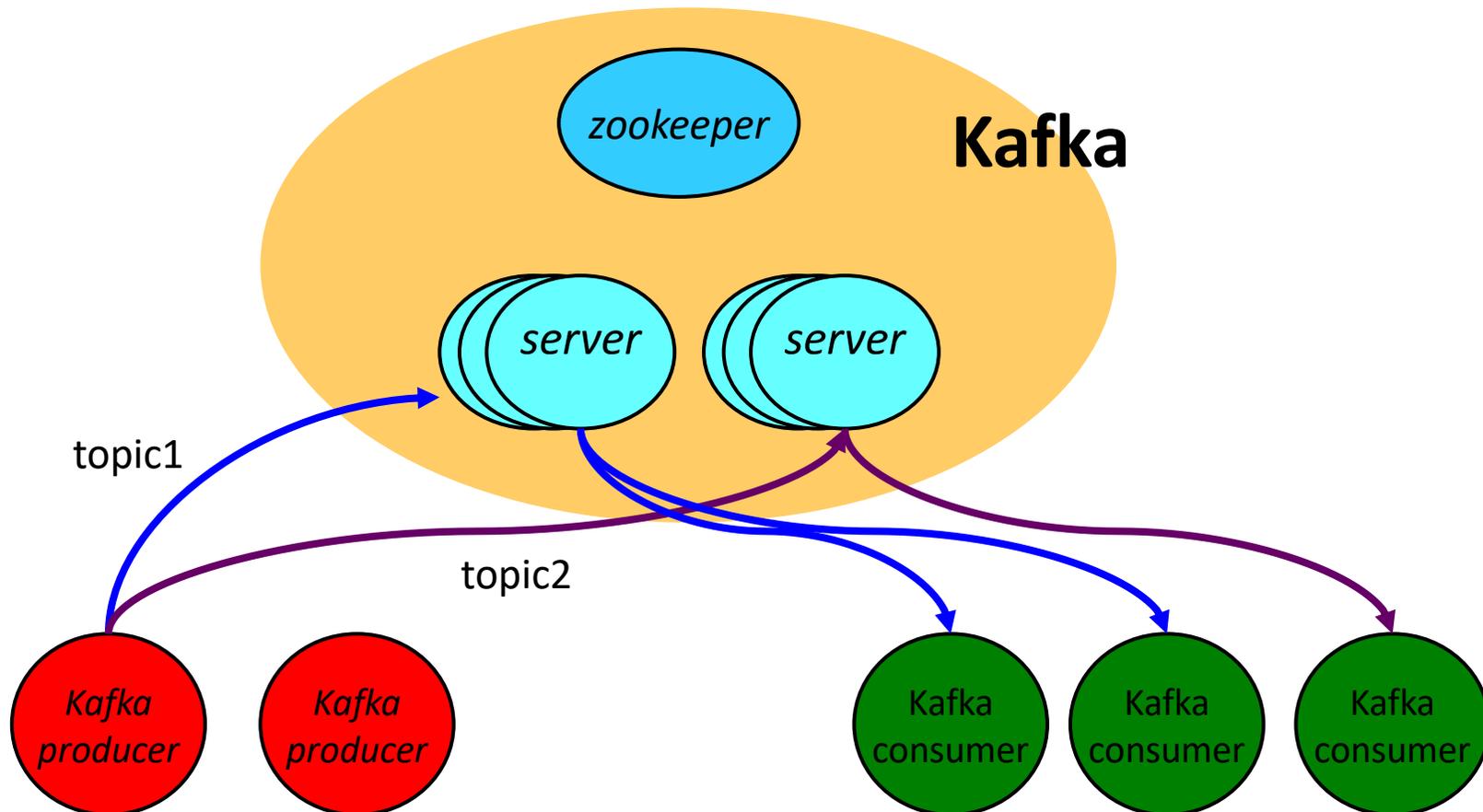
Apache Kafka é um sistema de comunicação em grupo (bastante sofisticado) que oferece uma API publish/subscribe, baseada em tópicos.

Oferece diversas garantias, incluindo persistência e múltiplas semânticas de entrega

Suporta particionamento e replicação dos servidores (brokers).

Depende do Zookeeper para tolerar falhas e resolver o *rendezvous* entre produtores e consumidores

APACHE KAFKA: ARQUITETURA



Editores publicam eventos num tópicos – eventos ficam totalmente ordenados (numa partição).

Assinantes consomem eventos ordenadamente a partir de tópicos.

KAFKA: PUBLISHER (PRODUCER)

```
Properties props = new Properties();

props.put("bootstrap.servers", "kafka:9092");

props.put("key.serializer", StringSerializer.class.getName());
props.put("value.serializer", StringSerializer.class.getName());

props.put("log.retention.minutes", 10);

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

String topic = ...;
String event = ...;

ProducerRecord<String, String> data = new ProducerRecord<>(topic ,
    event);
producer.send(data);
...
producer.close();
```

KAFKA: PUBLISHER (PRODUCER)

Endereço de servidores
para ligação ao Sistema.

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.serializer", StringSerializer.class.getName());  
props.put("value.serializer", StringSerializer.class.getName());
```

```
props.put("log.retention.minutes", 10);
```

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

```
String topic = ...;  
String event = ...;
```

```
ProducerRecord<String, String> data = new ProducerRecord<>(topic ,  
    event);  
producer.send(data);  
...  
producer.close();
```

KAFKA: PUBLISHER (PRODUCER)

```
Properties props = new Properties();  
props.put("bootstrap.servers", "kafka:9092")
```

Classe usada para
serializar/deserializar
mensagens.

```
props.put("key.serializer", StringSerializer.class.getName());  
props.put("value.serializer", StringSerializer.class.getName());
```

```
props.put("log.retention.minutes", 10);
```

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

```
String topic = ...;  
String event = ...;
```

```
ProducerRecord<String, String> data = new ProducerRecord<>(topic ,  
    event);  
producer.send(data);  
...  
producer.close();
```

KAFKA: PUBLISHER (PRODUCER)

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.serializer", StringSerializer.class);  
props.put("value.serializer", StringSerializer.class);
```

Tempo até remover
mensagens do sistema.

```
props.put("log.retention.minutes", 10);
```

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

```
String topic = ...;
```

```
String event = ...;
```

```
ProducerRecord<String, String> data = new ProducerRecord<>(topic ,  
    event);
```

```
producer.send(data);
```

```
...
```

```
producer.close();
```

KAFKA: PUBLISHER (PRODUCER)

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.serializer", StringSerializer.class.getName());  
props.put("value.serializer", StringSerializer.class.getName());
```

```
props.put("log.retention.minutes", 10);
```

Permite aceder ao Kafka para publicar eventos.

```
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

```
String topic = ...;
```

```
String event = ...;
```

```
ProducerRecord<String, String> data = new ProducerRecord<>(topic ,  
    event);
```

```
producer.send(data);
```

```
...  
producer.close();
```

KAFKA: PUBLISHER (PRODUCER)

```
Properties props = new Properties();  
  
props.put("bootstrap.servers", "kafka:9092");  
  
props.put("key.serializer", StringSerializer.class.getName());  
props.put("value.serializer", StringSerializer.class.getName());  
  
props.put("log.retention.minutes", 10);  
  
KafkaProducer<String, String> producer = new
```

Cria evento, indicando o tópic
e o conteúdo do evento, e
publica-o. Podem-se publicar
múltiplos eventos.

```
String topic = ...;  
String event = ...;  
  
ProducerRecord<String, String> data = new ProducerRecord<>(topic ,  
    event);  
producer.send(data);  
...  
producer.close();
```

KAFKA: SUBSCRIBER (CONSUMER)

```
Properties props = new Properties();

props.put("bootstrap.servers", "kafka:9092");

props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());

props.put("group.id", "gid" + System.nanoTime());

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("topic0", "topic1", ... ));

ConsumerRecords<String, String> records = consumer.poll(1000);
records.forEach( r -> {
    System.err.println( r.topic() + "/" + r.value());
});
consumer.commitSync();

consumer.close();
```

KAFKA: SUBSCRIBER (CONSUMER)

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.deserializer", StringDeserial  
props.put("value.deserializer", StringDeseri
```

Usado para identificar grupo de assinantes – usado para paralelizar processamento de eventos.

```
props.put("group.id", "gid" + System.nanoTime());
```

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

```
consumer.subscribe(Arrays.asList("topic0", "topic1", ... ));
```

```
ConsumerRecords<String, String> records = consumer.poll(1000);
```

```
records.forEach( r -> {
```

```
    System.err.println( r.topic() + "/" + r.value());
```

```
});
```

```
consumer.commitSync();
```

```
consumer.close();
```

KAFKA: SUBSCRIBER (CONSUMER)

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.deserializer", StringDeserializer.class.getName());  
props.put("value.deserializer", StringDeserializer.class.getName());
```

```
props.put("group.id", "gid" + System.nanoTime());
```

Permite aceder ao Kafka para consumir eventos.

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

```
consumer.subscribe(Arrays.asList("topic0", "topic1", ... ));
```

```
ConsumerRecords<String, String> records = consumer.poll(1000);
```

```
records.forEach( r -> {  
    System.err.println( r.topic() + "/" + r.value());
```

```
});
```

```
consumer.commitSync();
```

```
consumer.close();
```

KAFKA: SUBSCRIBER (CONSUMER)

```
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "kafka:9092");
```

```
props.put("key.deserializer", StringDeserializer.class.getName());  
props.put("value.deserializer", StringDeserializer.class.getName());
```

```
props.put("group.id", "gid" + S
```

```
KafkaConsumer<String, String> c
```

Subscreve os eventos dos tópicos indicados, consome primeiro conjunto de eventos e indica que eventos foram consumidos.

```
consumer.subscribe(Arrays.asList("topic0", "topic1", ... ));
```

```
ConsumerRecords<String, String> records = consumer.poll(1000);
```

```
records.forEach( r -> {  
    System.err.println( r.topic() + "/" + r.value());
```

```
});
```

```
consumer.commitSync();
```

```
consumer.close();
```

PUBLISH/SUBSCRIBE: PROPRIEDADES

Desacoplamento entre componentes

- Vantagens:
 - Emissor não necessita de conhecer receptor
 - Emissor e receptor não necessitam de estar a funcionar ao mesmo tempo
 - Emissor não necessita de esperar pelo receptor - assincronismo
 - Melhor desempenho
- Desvantagens:
 - Mais complicado definir propriedades exactas do sistema

Modelo de programação *event-driven*

- Modelo reactivo - como nos GUIs

REPLICAÇÃO E SISTEMAS PUBLISH/SUBSCRIBE

Os sistemas *publish/subscribe*, como o Kafka, são usados como “building block” para construir sistemas replicados.

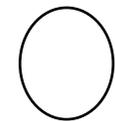
Porquê?

Sistema *publish/subscribe* garante a fiabilidade da propagação das operações entre as réplicas.

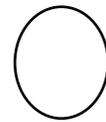
(Alguns) Sistemas *publish/subscribe* ordenam as operações, permitindo implementar facilmente uma solução de replicação de máquina de estados.

REPLICAÇÃO DE MÁQUINA DE ESTADOS

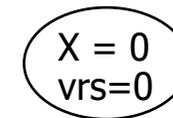
Com recurso a sistema pub/sub
com ordenação total das
mensagens



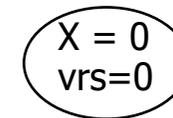
cliente



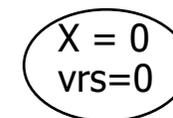
cliente



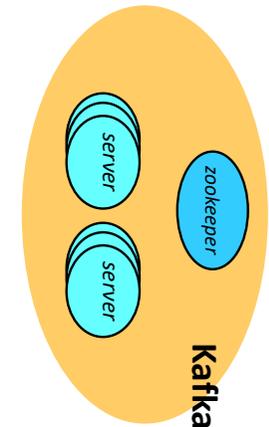
réplica



réplica



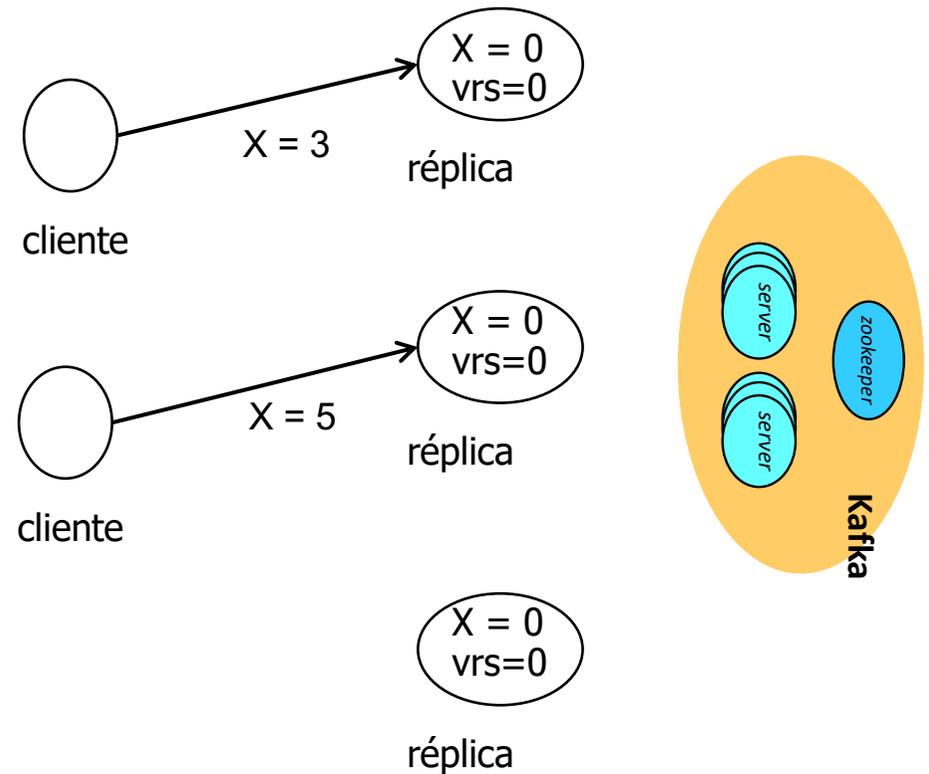
réplica



REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

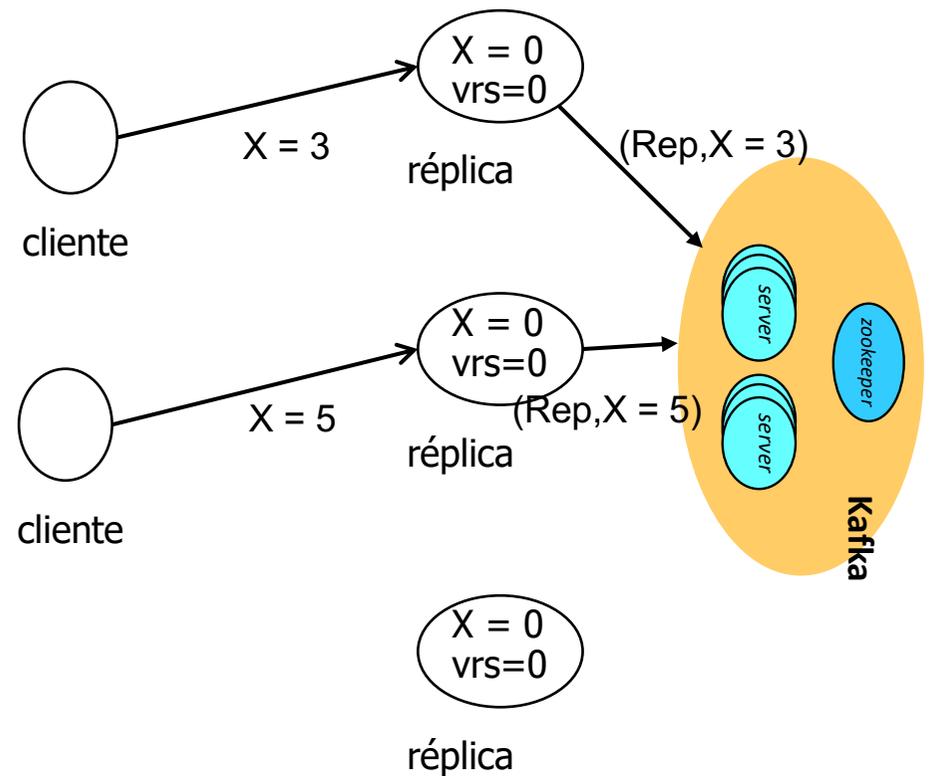
1. Cliente envia operação para qualquer réplica.



REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

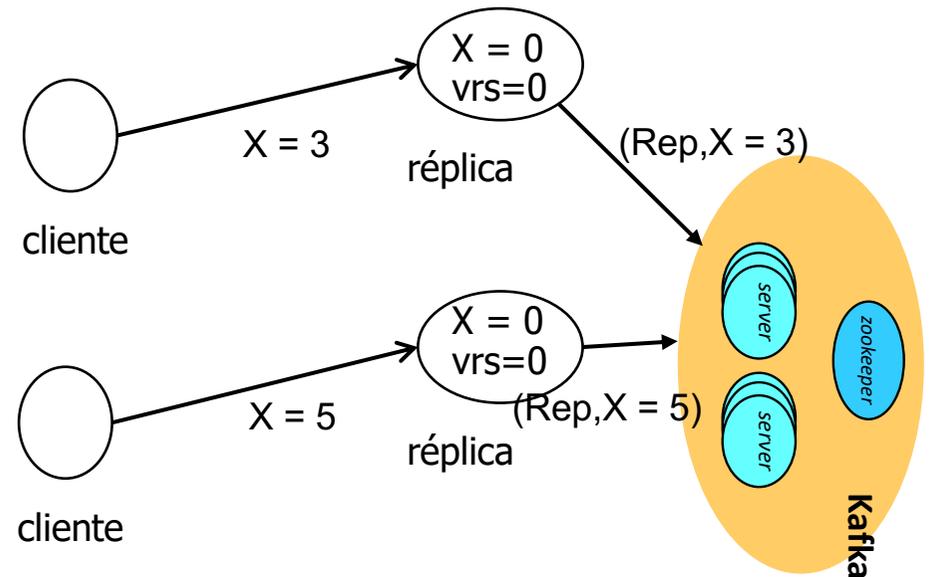
1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.



REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.

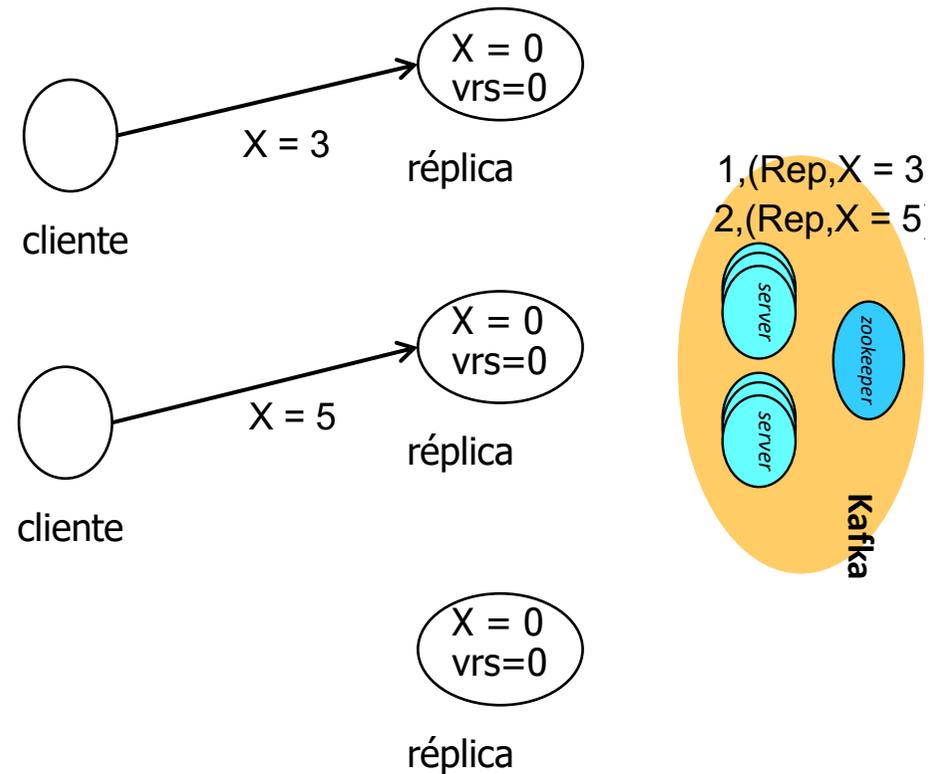


```
ProducerRecord<String, String> data = new ProducerRecord<>(topic, event);  
long offset = producer.send(data).get().offset();
```

REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.



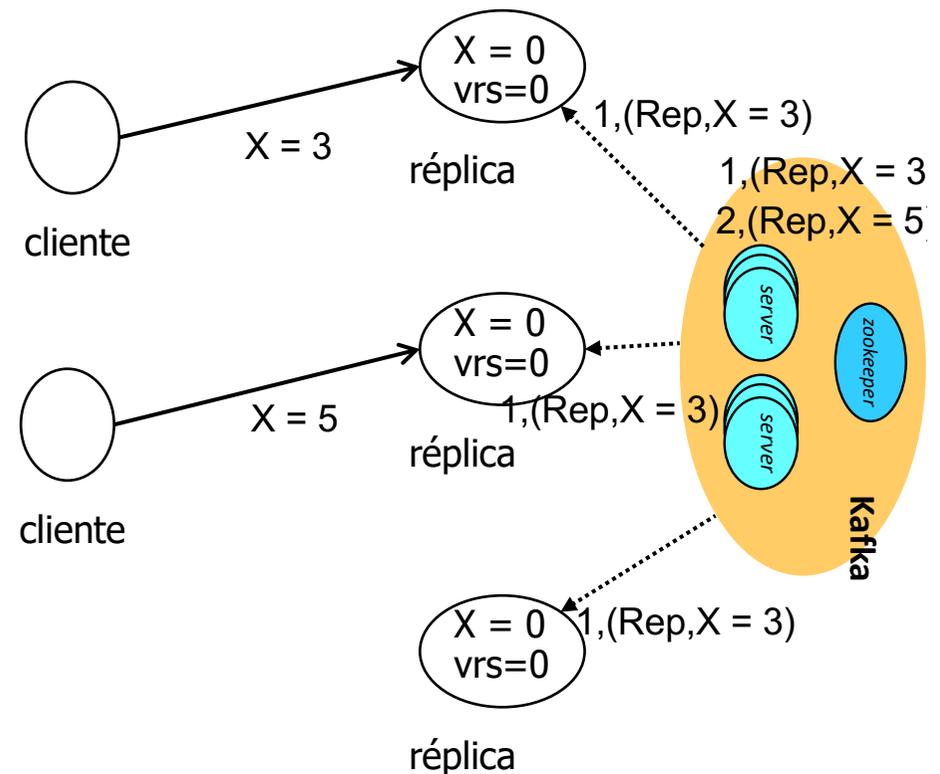
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.

Thread em background a executar no servidor.

1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.



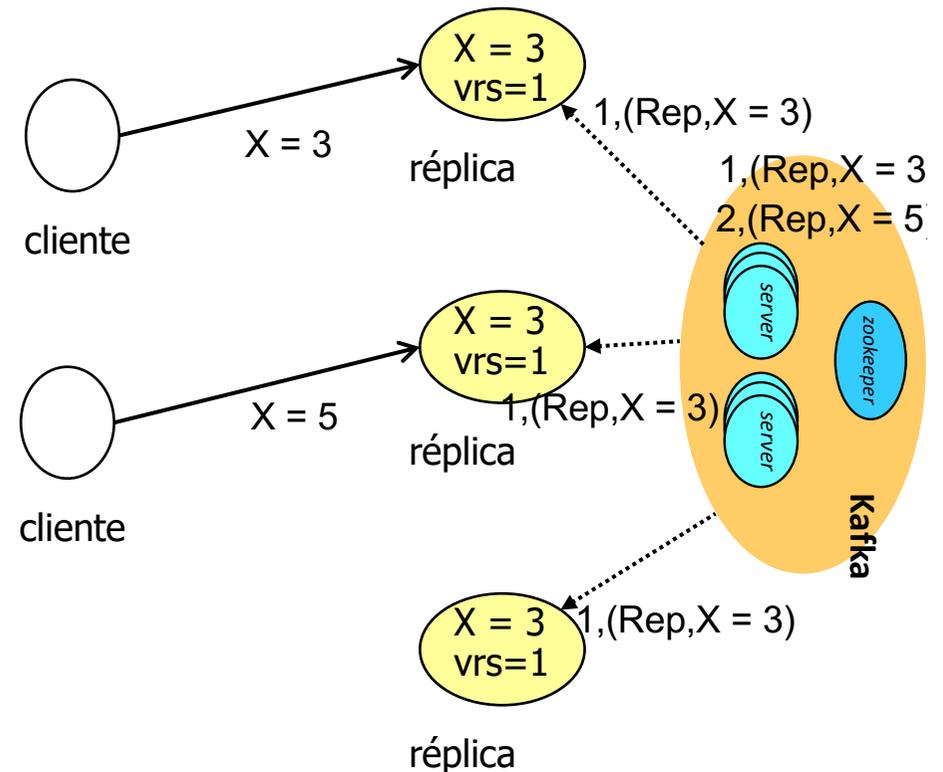
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.

Thread em background a executar no servidor.

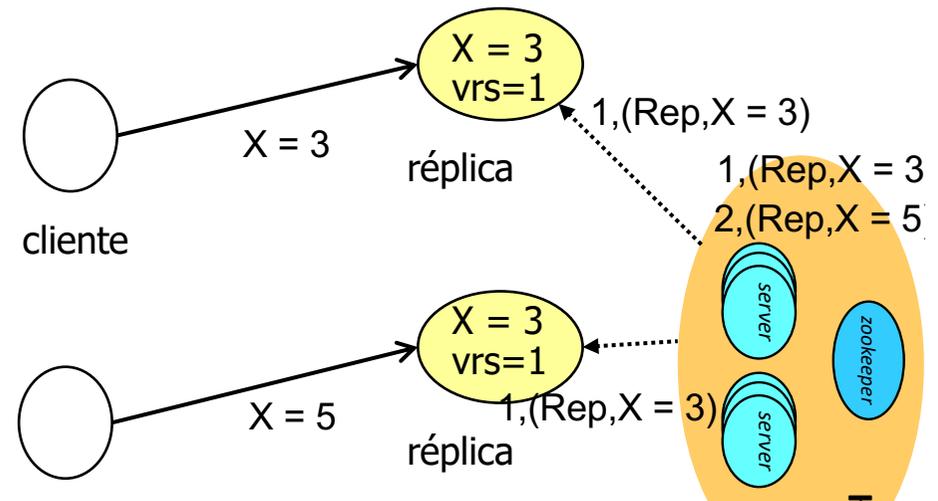
1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.



REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.



```
ConsumerRecords<String, String> records = consumer.poll(1000);
records.forEach( r -> {
    System.err.println(r.offset() + "->" + r.topic() + "/" + r.value());
});
```

1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.

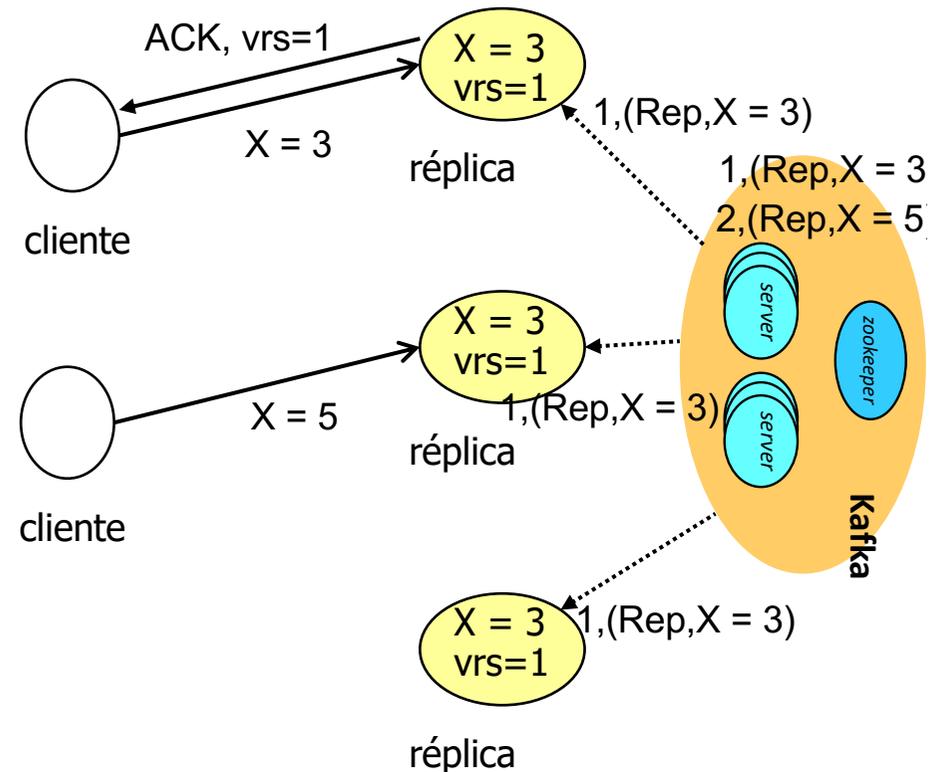
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.
3. Após execução da operação retorna resultado ao cliente

Thread em background a executar no servidor.

1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.



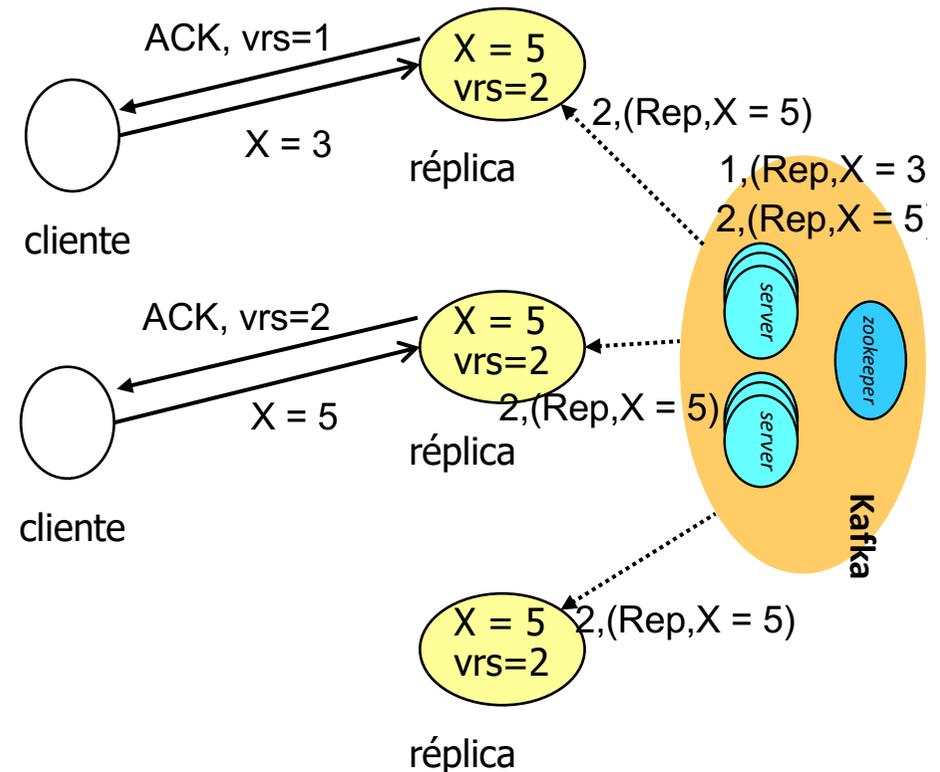
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.
3. Após execução da operação retorna resultado ao cliente

Thread em background a executar no servidor.

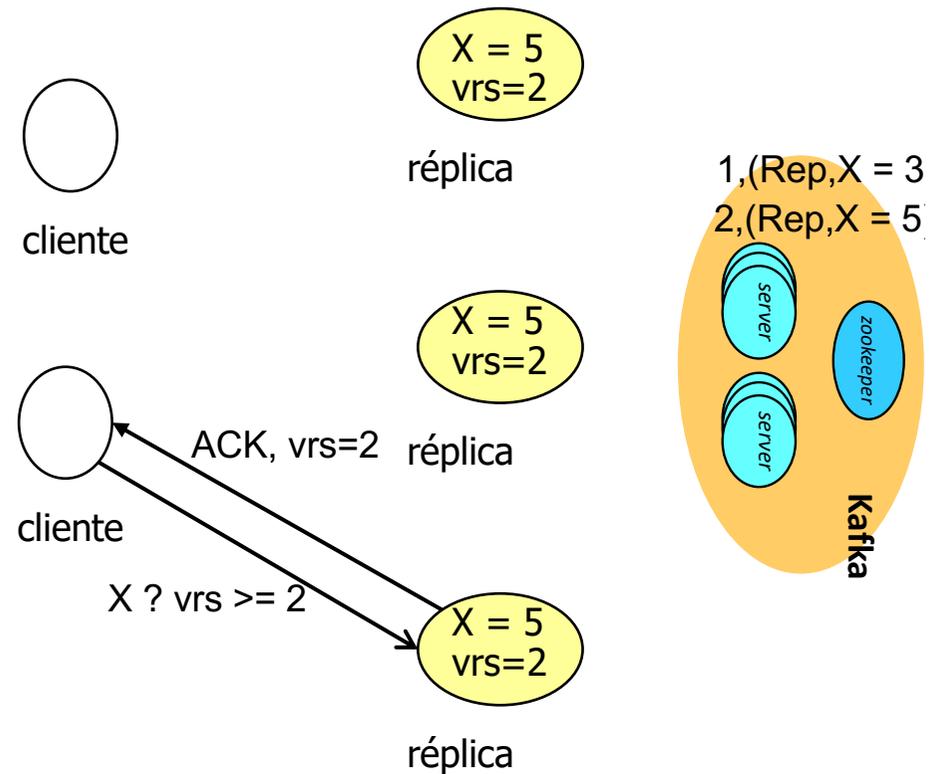
1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.



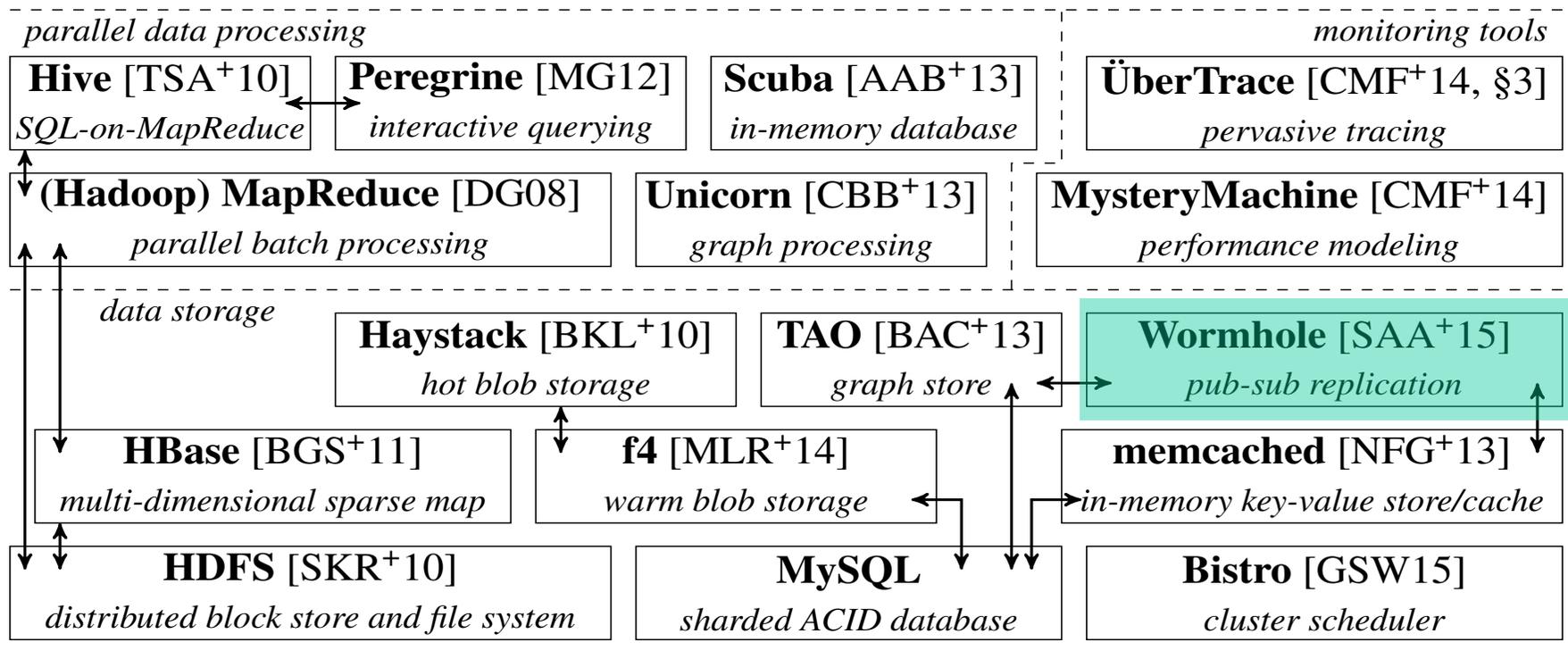
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de leitura

1. Cliente envia operação para qualquer réplica (enviando última versando vista).
2. Servidor responde caso versão local seja igual ou superior à versão pedida; caso contrário espera até executar operação correspondente.



EXEMPLO: FACEBOOK



NA ÚLTIMA AULA

Sistemas de disseminação de eventos: pub/sub

- o(s) **editor**(es)/publisher(s) **publica**(m) mensagens/eventos, usualmente organizadas por **tópicos** ou **canais** temáticos;
- o(s) **assinante**(s)/subscriber(s) manifestam interesse em receber certas mensagens, indicando o(s) **tópicos**, **canal** ou **padrão do conteúdo** das mensagens a receber

REPLICAÇÃO E SISTEMAS PUBLISH/SUBSCRIBE

Os sistemas *publish/subscribe*, como o Kafka, são usados como “building block” para construir sistemas replicados.

Porquê?

Sistema *publish/subscribe* garante a fiabilidade da propagação das operações entre as réplicas.

(Alguns) Sistemas *publish/subscribe* ordenam as operações, permitindo implementar facilmente uma solução de replicação de máquina de estados.

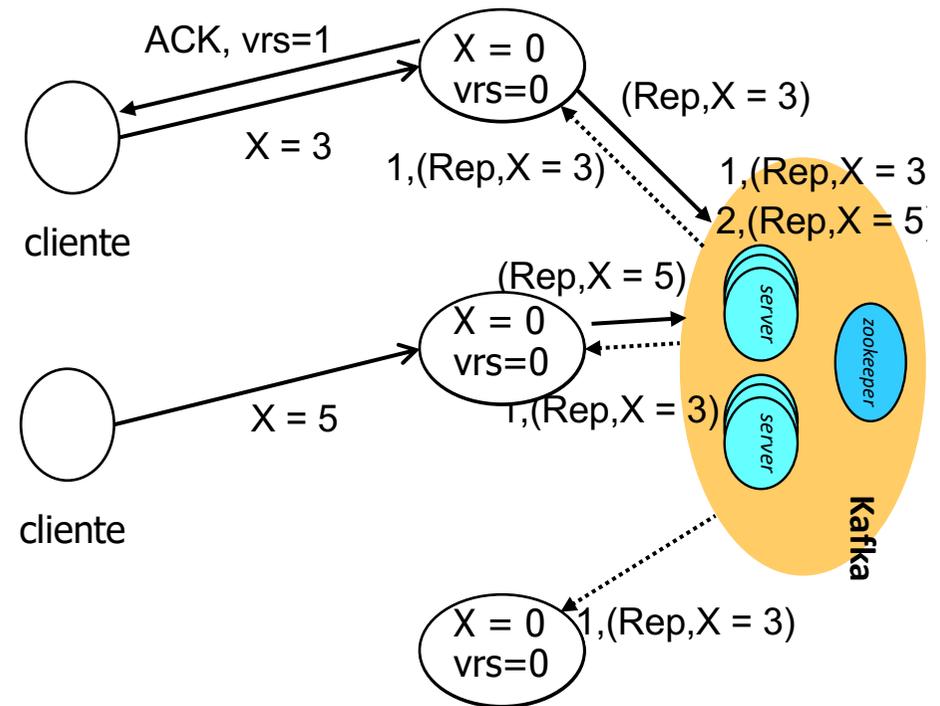
REPLICAÇÃO USANDO PUB/SUB

Execução de uma operação de escrita

1. Cliente envia operação para qualquer réplica.
2. Servidor publica operação no sistema pub/sub; resultado indica número de ordem da operação.
3. Após execução da operação retorna resultado ao cliente

Thread em background a executar no servidor.

1. Quando recebe operação do sistema pub/sub, executa-a e atualiza o número da última operação processada.



AGENDA PARA HOJE

Sistemas de disseminação de eventos: pub/sub

Sistemas de message queues

Comunicação em grupo

MESSAGE-ORIENTED MIDDLEWARE (MOM) OU MESSAGE-QUEUING SYSTEMS

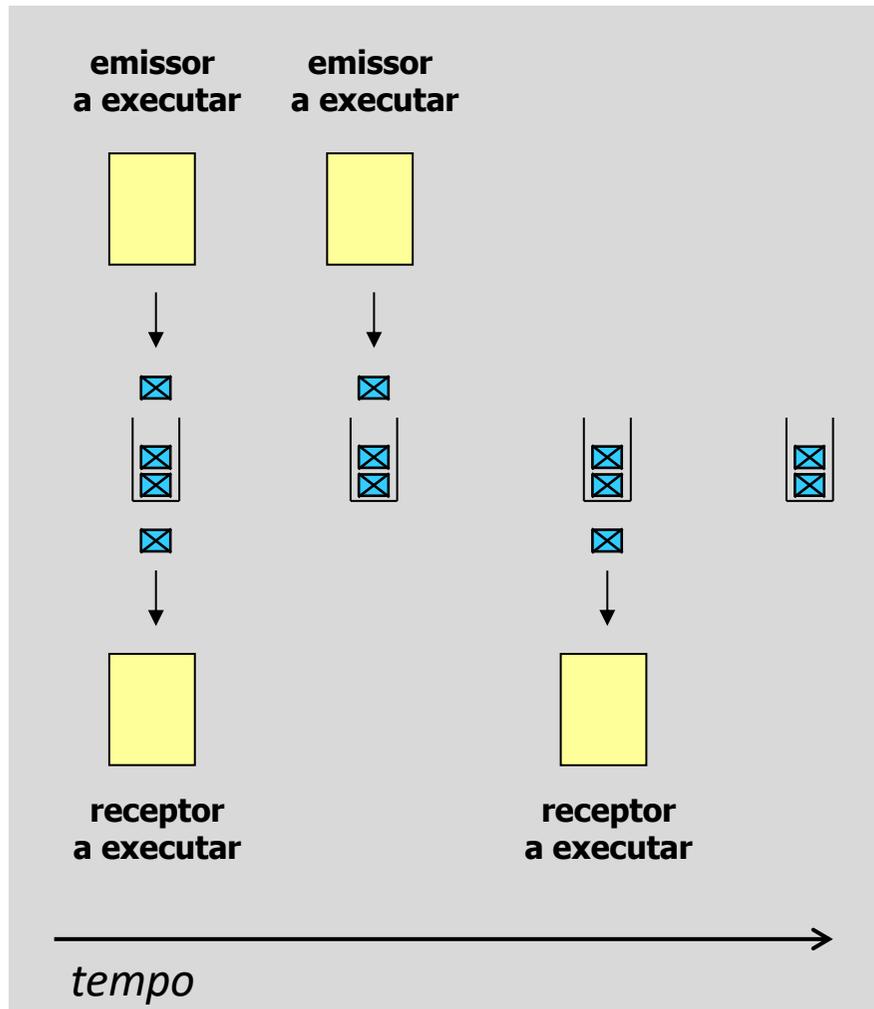
Os processos comunicam pela troca de mensagens usando um subsistema intermédio que assegura a persistência através de filas de mensagens (queues)

- Comunicação assíncrona
- Suporta transferência de mensagens que duram vários minutos

Uma mensagem é sempre endereçada por um processo para uma queue e só pode ser consumida da queue por um outro processo / aplicação (uma queue pode, no entanto, ser partilhada por vários processos / aplicações)

- O emissor apenas tem garantias que a mensagem foi entregue na queue

INTERACÇÃO EMISSOR / RECEPTOR



Primitivas:

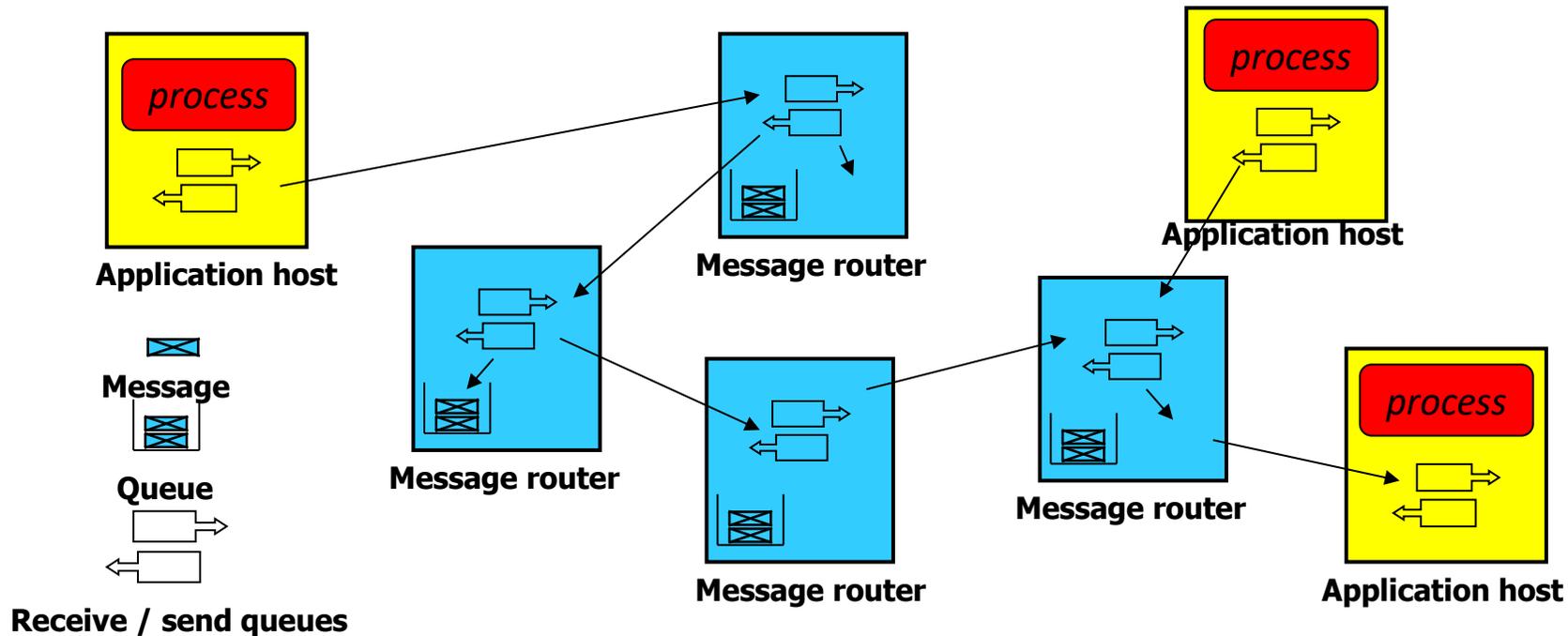
Put – adiciona uma mensagem a uma *queue*

Get – bloqueia até que a *queue* não esteja vazia e remove a primeira mensagem

Poll – verificar se existem mensagens na *queue*

Notify – Instalar um *handler* para ser notificado se uma mensagem chegar

ESTRUTURA GERAL DE UM MOM



Mensagens podem ser propagadas para o destino através de filas intermédias

Algumas máquinas podem ter filas e atuar apenas como *routers*

Permite implementar funcionalidade adicional – ex.: *log* para segurança ou tolerância a falhas, conversão de formato de mensagens/gateway (message brokers)

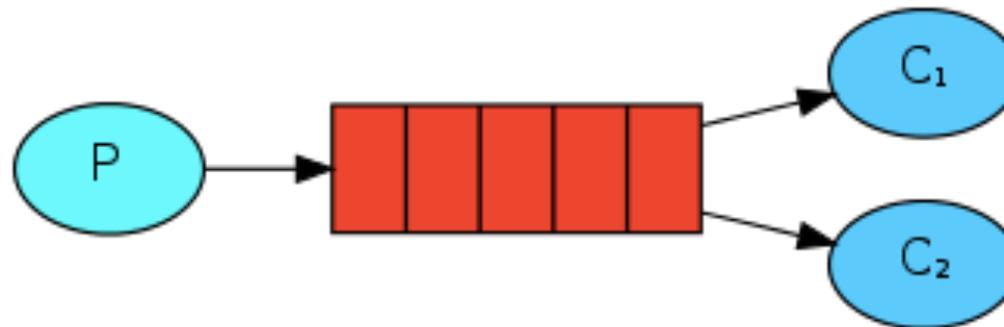
Produtos: RabbitMQ, IBM MQ, Amazon MQ, Azure Service Bus, Google Cloud Pub/Sub

ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Distribuir tarefas por múltiplos processos

Produtor escreve tarefa numa fila. Consumidores retiram as tarefas da fila.

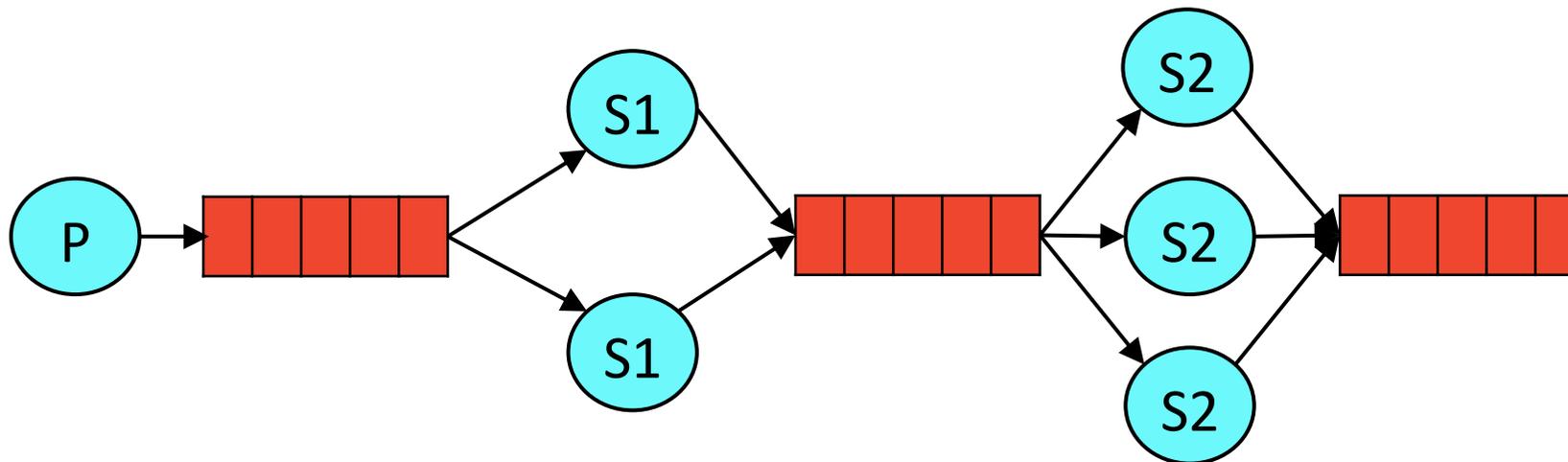


ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Executar fluxo de ações (e.g. ao processar um pedido)

Para cada ação existe uma fila. Cada ação pode ser executada por um número variável de processos dependendo da complexidade.



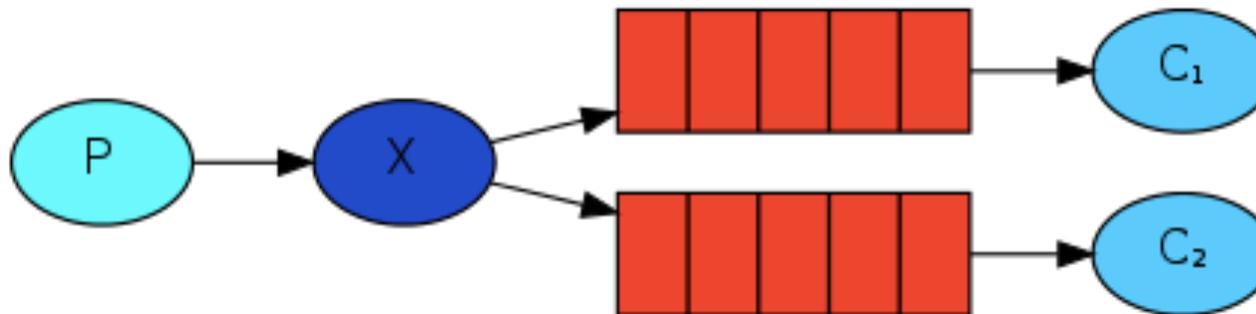
ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Disseminar tarefas para múltiplos processos

Broadcast, Publish/Subscribe (filtragem no receptor)

Cada consumidor tem uma fila. A mensagem enviada pelo produtor é colocada na fila de todos os consumidores.



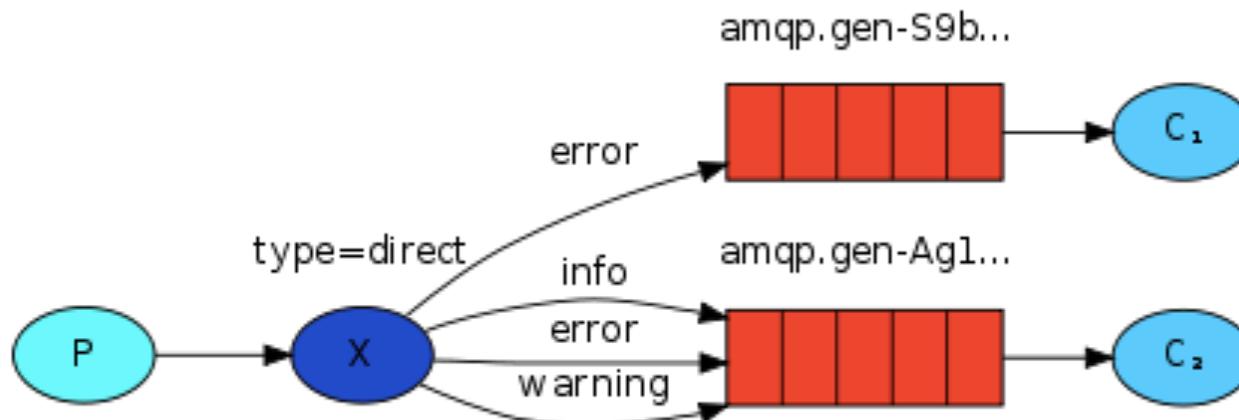
ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Receber mensagens seletivamente

publish/subscribe (baseada em tópicos)

As mensagens são colocadas nas filas consoante o seu tópico.



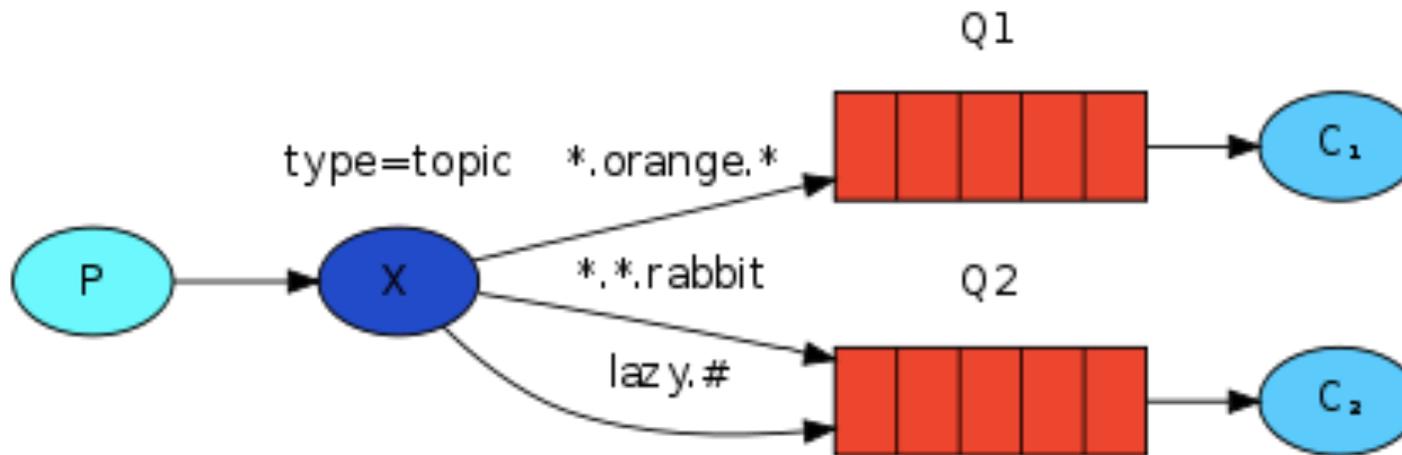
ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Receber mensagens baseado num padrão

Content-based routing

As mensagens são colocadas nas filas dependendo do conteúdo da mensagem.

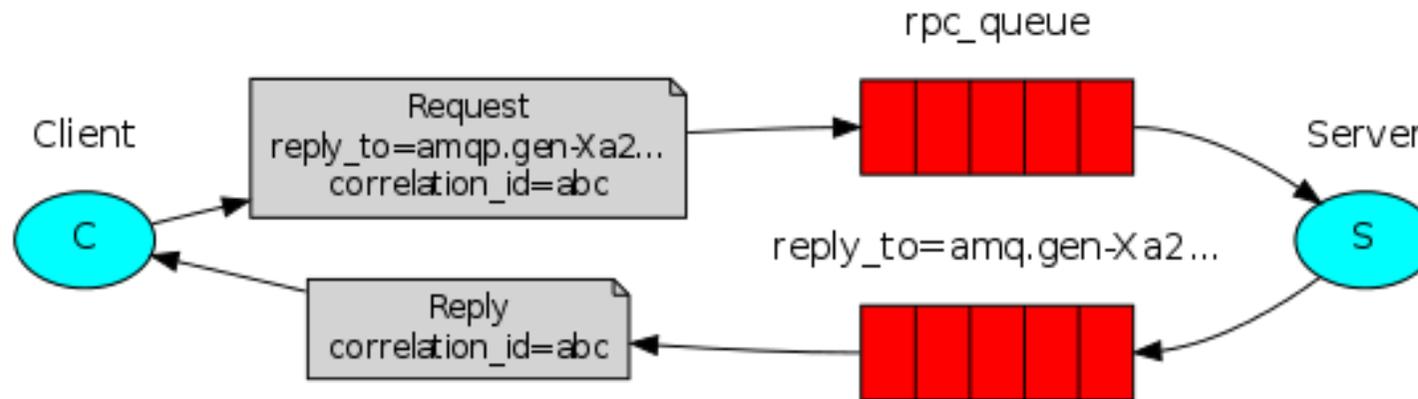


ALGUNS USOS DE SISTEMAS DE MESSAGE-QUEUE

(Figures from: <https://www.rabbitmq.com/tutorials/tutorial-three-java.html>)

Invocação remota (c/ duas queues)

Implementa-se um sistema de RPC assíncrono, usando uma fila para colocar os pedidos, que são depois processado pelo servidor, e outra para cada cliente, na qual o servidor coloca o resultado.



RABBITMQ

MOM que implementa o protocolo AMPQ 1.0

AMPQ (Advanced Message Queuing Protocol) é um protocolo aberto e standardizado;

A arquitetura é cliente/servidor, com múltiplos servidores interligados;

Clientes para diversas linguagens: Java, Python, PHP, Go, etc.

O transporte usa um formato binário, sobre o qual se pode usar, por exemplo JSON.

RABBITMQ

Set-up queues

```
ConnectionFactory factory = new ConnectionFactory();  
Connection conn = factory.newConnection("localhost");  
Channel channel = conn.createChannel();  
  
channel.queueDeclare("my_queue", true);
```



durable

RABBITMQ : CONSUMER/RECEIVER

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE_NAME, false, false, false, null);

String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());

channel.close();
connection.close();
```

RABBITMQ : CONSUMER/RECEIVER

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE_NAME, false, false, false, null);

Consumer consumer = new DefaultConsumer(channel) {
    public void handleDelivery(String cTag, Envelope env, AMQP.BasicProperties props, byte[] body) throws IOException {
        ...
    }
};

for(;;)
    channel.basicConsume(QUEUE_NAME, true, consumer);

// channel.close();
// connection.close();
```

AGENDA PARA HOJE

Sistemas de disseminação de eventos: pub/sub

Sistemas de message queues

Comunicação em grupo

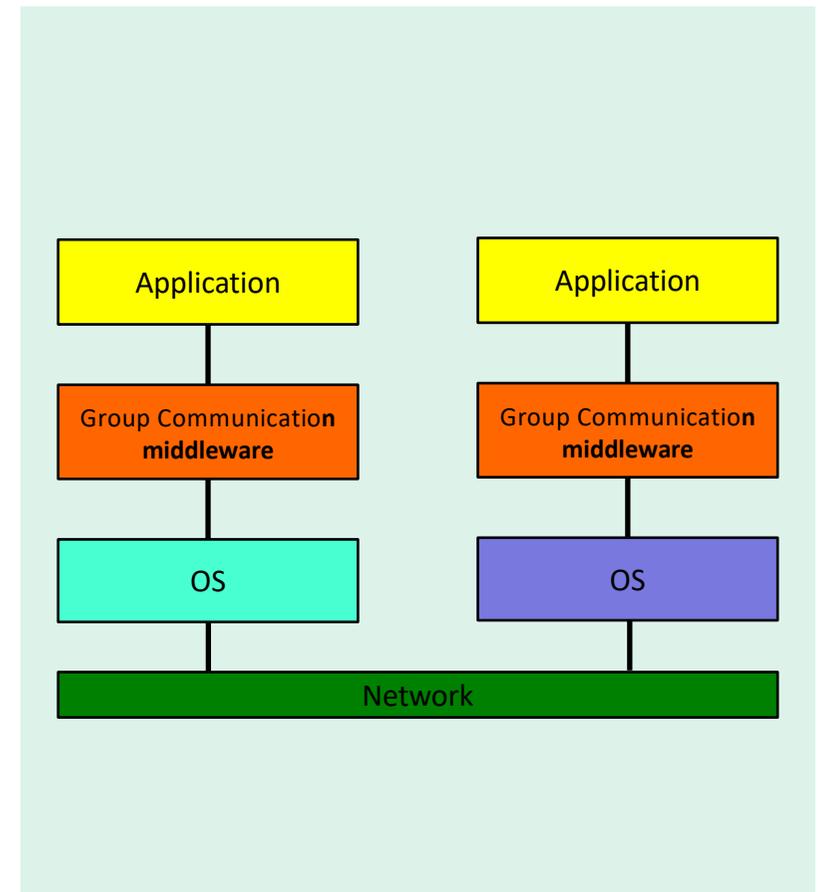
COMUNICAÇÃO EM GRUPO : IMPLEMENTAÇÃO EM MIDDLEWARE

O middleware de comunicação em grupo fica responsável por questões como:

Gestão da filiação

group_id -> {*receiver1*, *receiver2*, ...}

Gestão da comunicação: envio, encaminhamento, buferização das mensagens, ACKs, NACKs, etc.

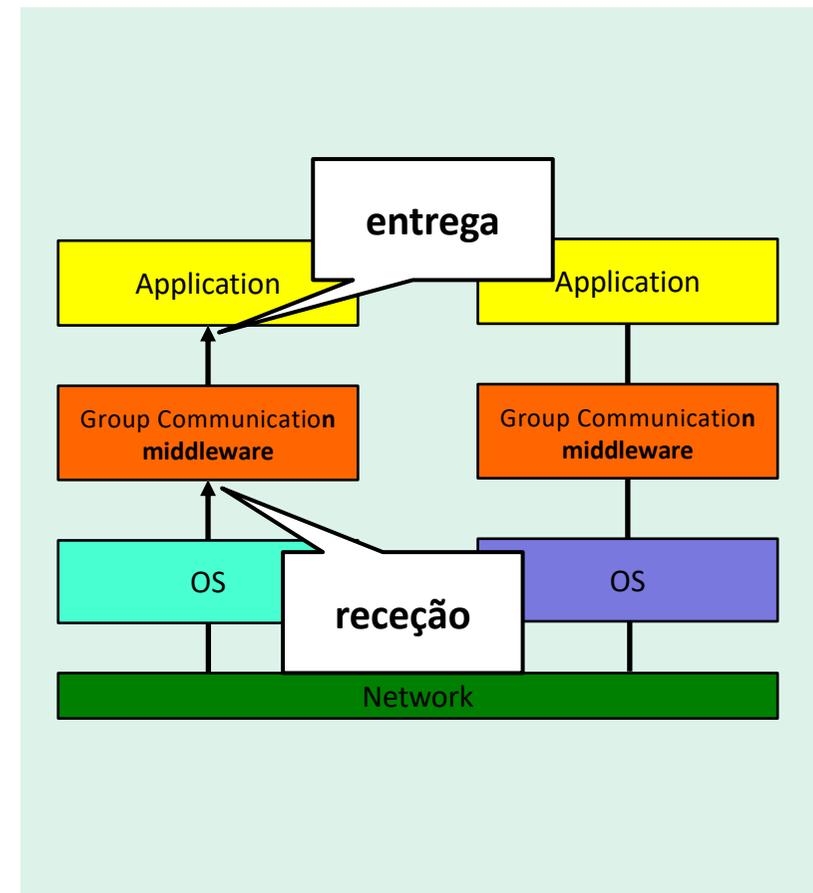


COMUNICAÇÃO EM GRUPO : ENTREGA DAS MENSAGENS

Diz-se que o sistema de comunicação **entrega** uma mensagem quando ela é entregue à aplicação para ser consumida

O middleware pode **atrasar a entrega** das mensagens, **reordená-las**, a fim de garantir as propriedades desejadas

Em cada instância do *middleware*, haverá para esse efeito, uma fila local de mensagens recebidas, ainda por entregar



CARACTERIZAÇÃO DO MULTICAST: FIABILIDADE

Multicast não fiável (*unreliable multicast*)

Multicast fiável (*reliable multicast*)

CARACTERIZAÇÃO DO MULTICAST: FIABILIDADE

Multicast não fiável (*unreliable multicast*) – em caso de falha, não existem garantias sobre a entrega das mensagens aos vários elementos do grupo.

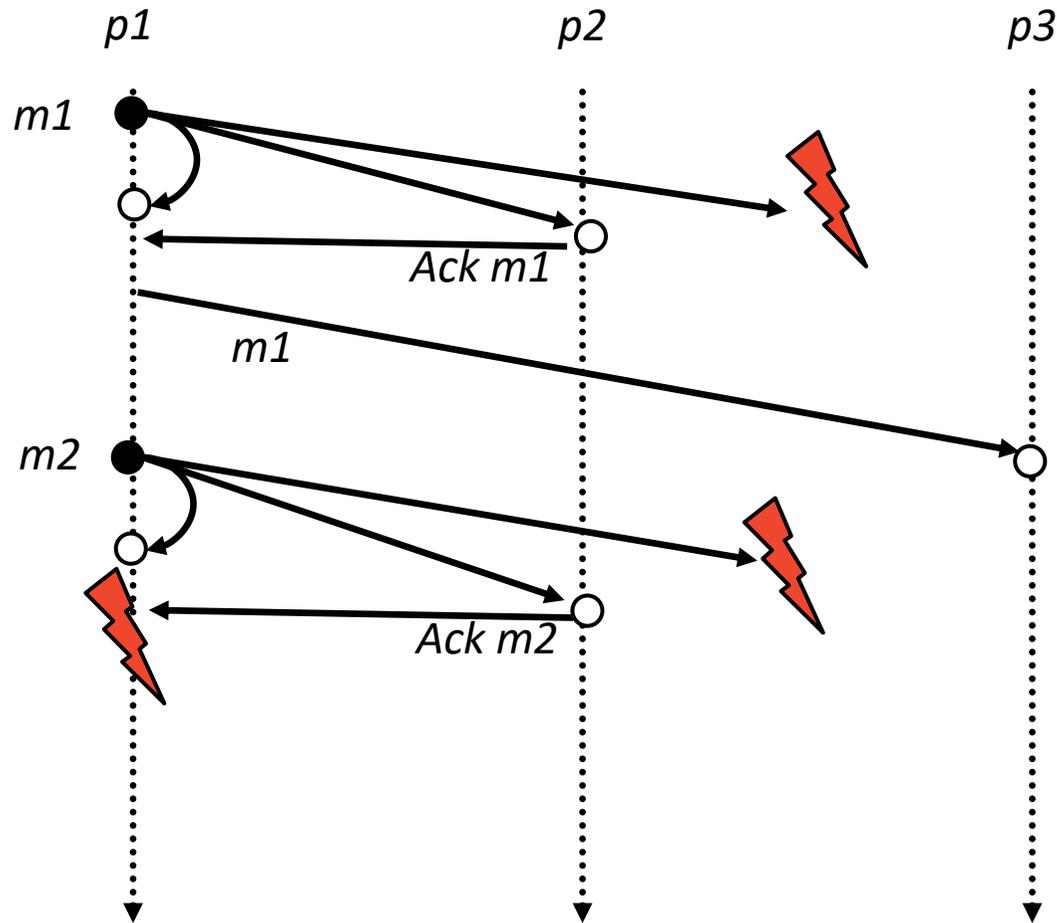
- i.e., a mensagem pode ser não entregue a nenhum ou a alguns dos membros do grupo.
- e.g. IP multicast

CARACTERIZAÇÃO DO MULTICAST: FIABILIDADE

Multicast fiável (*reliable multicast*) – uma mensagem enviada para um grupo ou é entregue a todos os membros corretos (que não falham) ou a nenhum

- Membros que falham podem ter entregue a mensagem ou não
- Implementação simples (e errada): o emissor emite, individualmente, para cada um dos elementos do grupo de forma fiável (acks+retransmissões)
 - **Porquê errada?**

EXEMPLO



Em caso de falhas do emissor, uma mensagem pode ser entregue apenas a alguns dos membros do grupo. Necessário propagar mensagens já entregues para todos os outros nós.

CARACTERIZAÇÃO DO MULTICAST: FIABILIDADE

Multicast fiável (reliable multicast) – uma mensagem enviada para um grupo ou é entregue por todos os membros corretos (que não falham) ou por nenhum

- Membros que falham podem ter entregue a mensagem ou não
- Implementação simples (e errada): o emissor emite para cada um dos elementos do grupo de forma fiável (acks+retransmissões)
 - Em caso de falha do emissor, é necessário que os elementos do grupo propaguem as mensagens recebidas para os elementos que ainda não as receberam
- **Uniform agreement**: se algum processo que falha entrega a mensagem, todos os processos corretos devem entregar a mensagem

CARACTERIZAÇÃO DO MULTICAST: ORDEM

Sem ordem – as mensagens podem ser entregues por diferentes ordens em diferentes processos

Ordem FIFO – as mensagens de um processo são entregues pela ordem de emissão em todos os processos

Na implementação recorre-se a números de sequência gerados em cada emissor...

CARACTERIZAÇÃO DO MULTICAST: ORDEM

Ordem causal – se **m1** pode causar **m2**, **m1** deve ser entregue sempre antes de **m2**

- Se duas mensagens forem **emitidas pelo mesmo processo**, considera-se que o **envio da primeira pode causar o envio da segunda**
 - A entrega da primeira deve acontecer antes da entrega da segunda
- Se duas mensagens forem **emitidas em processos diferentes**, a **primeira pode causar a segunda se for entregue antes do envio da segunda** (ou transitivamente incluindo outras mensagens)
- Se uma mensagem não pode causar a outra, qualquer ordem de entrega respeita a ordem causal

CARACTERIZAÇÃO DO MULTICAST: ORDEM

Ordem causal – se **m1** pode causar **m2**, **m1** deve ser entregue sempre antes de **m2**

- A implementação da ordem causal pode ser conseguida por recurso a relógios lógicos (+ ACKs) ou relógios vetoriais.

CARACTERIZAÇÃO DO MULTICAST: ORDEM

Ordem total – as mensagens m_1 , m_2 são entregues por ordem total, se forem entregues pela **mesma ordem em todos os processos**

- $\forall p_i, \text{receive}(m_1, p_i) < \text{receive}(m_2, p_i)$ **ou**
- $\forall p_i, \text{receive}(m_1, p_i) > \text{receive}(m_2, p_i)$
- NOTA: a ordem de entrega das mensagens de um mesmo emissor pode ser diferente da ordem de emissão
- Um sistema de multicast fiável no qual as mensagens são entregues por ordem total chama-se sistema de **multicast atómico**

Ordem total causal – ordem total + ordem causal

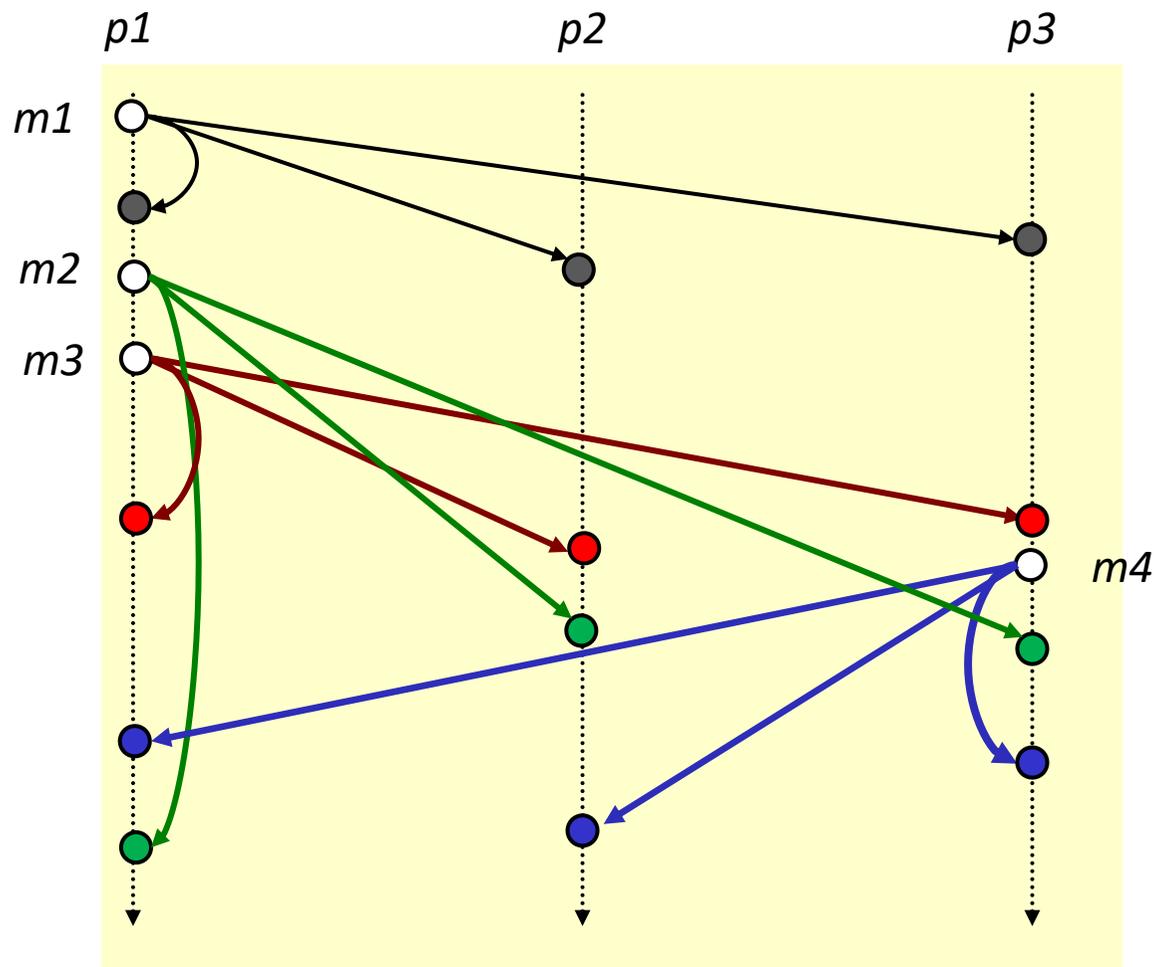
- As mensagens são entregues pela **mesma ordem** em **todos** os processos, segundo uma ordem que respeita a **causalidade**.

CARACTERIZAÇÃO DO MULTICAST: ORDEM

Ordem total – as mensagens m_1 , m_2 são entregues por ordem total, se forem entregues pela **mesma ordem em todos os processos**

- Implementação pode ser conseguida à custa de um **sequenciador**.
- solução centralizada – as mensagens são enviadas ao sequenciador e este impõe a ordem (transformando a ordem total em ordem FIFO).
- solução descentralizada – o sequenciador apenas emite uma “stream” adicional que indica a ordem de entrega a observar em todos os receptores.

ORDENS: EXEMPLO



$m1, m2$: FIFO, total, causal
 $m1, m3$: FIFO, total, causal
 $m1, m4$: total, causal

$m2, m3$: total, não FIFO, não causal

$m2, m4$: sem ordem

$m3, m4$: causal, total

FACETAS DA COMUNICAÇÃO MULTICAST: RESUMO

Fiabilidade

- Multicast fiável
- Multicast não-fiável

Ordenação das mensagens

- Sem ordem
- Ordem FIFO
- Ordem total

- Ordem casual
- Ordem total causal

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Pode-se usar um sistema de comunicação em grupo para propagar e executar operações num conjunto de réplicas.

Replicação de máquina de estado:

- Ordem total;
- Replicação fiável.

Pode ser usado para replicar qualquer serviço com operações deterministas.

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço de mensagens.

Duas operações

1. adicionar mensagem a mailbox (post message);
2. remover mensagem de mailbox (remove message from inbox + delete message).

Fiabilidade?

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço de mensagens.

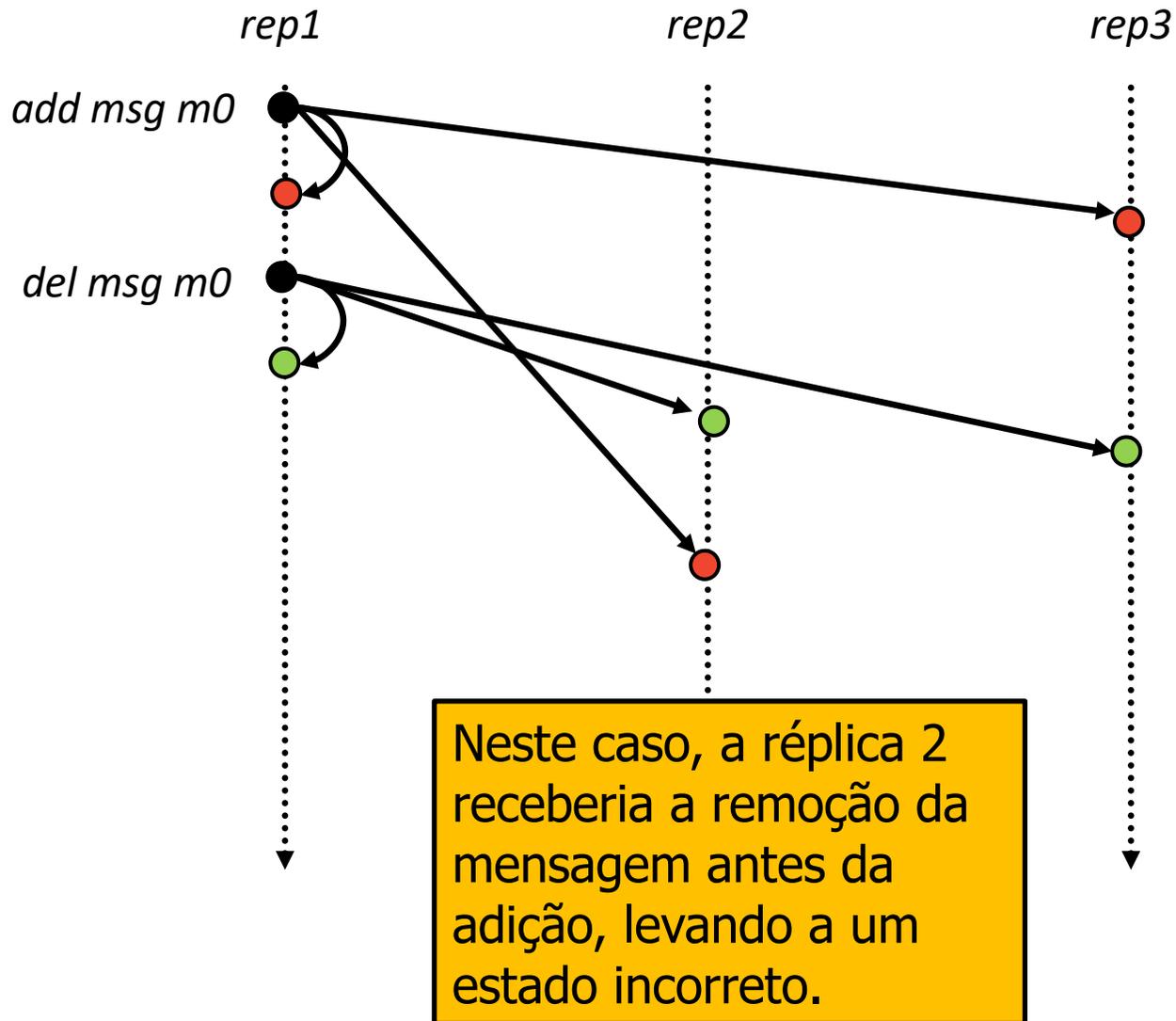
Duas operações

1. adicionar mensagem a mailbox (post message);
2. remover mensagem de mailbox (remove message from inbox + delete message).

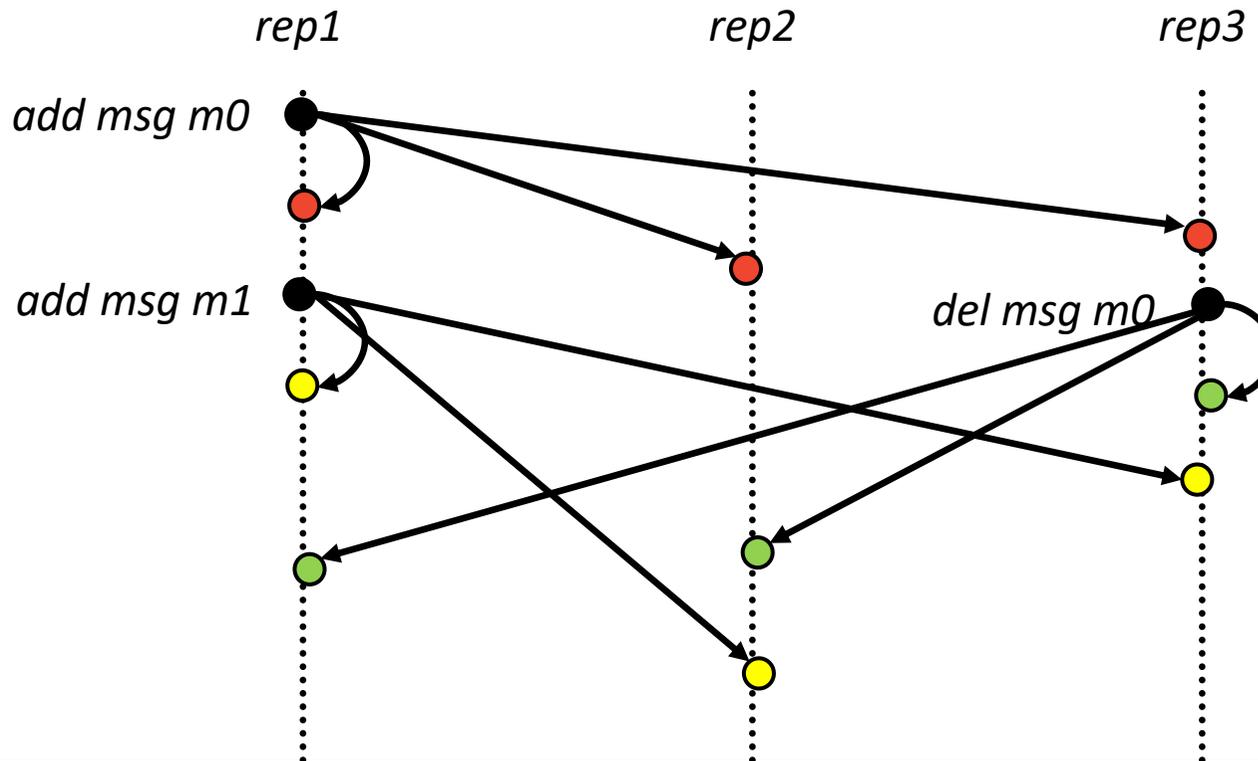
Fiabilidade? Multicast fiável.

Ordem das mensagens?

EXEMPLO : SEM ORDEM



EXEMPLO : ORDEM CAUSAL



Apesar de executarem as operações por ordem diferente, todas as réplicas convergem para o mesmo estado, porque a adição dum mensagem m tem de acontecer antes da remoção de m . Logo, por ordem causal, a remoção será entregue sempre depois da adição respetiva.

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço de mensagens.

Duas operações:

1. adicionar mensagem a mailbox (post message);
2. remover mensagem de mailbox (remove message from inbox + delete message).

Fiabilidade? Multicast fiável.

Ordem das mensagens? Ordem causal.

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Outro exemplo que necessita de ordem causal

Fórum de mensagens

- resposta a uma mensagem deve ser executada sempre depois da mensagem original;
- respostas concorrentes não necessitam de aparecer pela mesma ordem.

Fiabilidade? Multicast fiável.

Ordem das mensagens? Ordem causal.

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço de utilizadores.

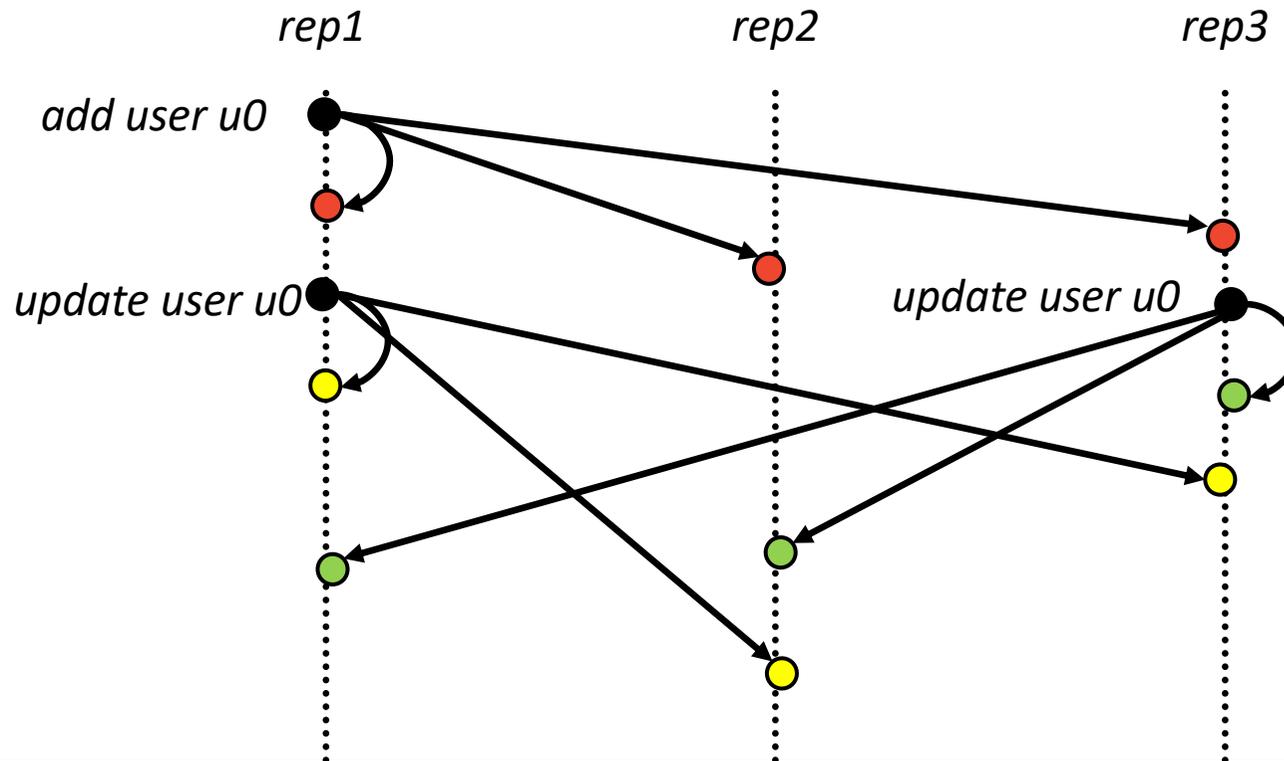
Três operações:

1. adicionar um utilizador (*post user*);
2. atualizar um utilizador (*update user*);
3. remover um utilizador (*remove user*).

Fiabilidade? Multicast fiável.

Ordem das mensagens?

EXEMPLO : ORDEM CAUSAL



Atualizações concorrentes podem ser processadas por ordens diferentes em diferentes réplicas, levando a um estado diferente. Ordem causal não é suficiente. É necessária ordem total (ou processamento adicional nas operações).

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço de utilizadores.

Três operações:

1. adicionar um utilizador (post user);
2. atualizar um utilizador (update user);
3. remover um utilizador (remove user).

Fiabilidade? Multicast fiável.

Ordem das mensagens? Ordem total.

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço que mantém medidas de sensores.

Uma operação:

1. Registrar leitura dum sensor – operação executada periodicamente – e.g. de 10 em 10 segundos.

Fiabilidade?

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço que mantém medidas de sensores.

Uma operação:

1. Registrar leitura dum sensor – operação executada periodicamente – e.g. de 10 em 10 segundos.

Fiabilidade? Multicast não fiável.

Se se perder uma operação não é importante, porque o próximo registo atualizará a informação.

Ordem das mensagens?

SISTEMAS DE COMUNICAÇÃO EM GRUPO E REPLICAÇÃO DE DADOS

Serviço que mantém medidas de sensores.

Uma operação:

1. Registrar leitura dum sensor – operação executada periodicamente – e.g. de 10 em 10 segundos.

Fiabilidade? Multicast não fiável.

Se se perder uma operação não é importante, porque o próximo registo atualizará a informação.

Ordem das mensagens? Sem ordem.

Se uma operação for entregue fora de ordem, o próximo registo atualizará a informação.

PARA SABER MAIS

G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, Distributed Systems –Concepts and Design, Addison-Wesley, 5th Edition, 2011

- capítulo 6.1-6.4, 15.4