

Sistemas Distribuídos

Comunicação em Grupo

abril de 2019

um nível a mais de indireção...

- desacoplamento espacial
- desacoplamento temporal

- comunicação de um para ? (um, vários, muitos)

grupos fortemente acoplados:

- replicação de serviços
 - confiabilidade
 - tempo de resposta
- clientes com estado compartilhado
- computação científica
- ...

Primitiva de comunicação *em grupo*

- um processo envia uma mensagem para um grupo de processos e *todos* os destinatários recebem
- primitivas de envio de mensagem em broadcast

grupos fracamente acoplados:

- observadores “melhor esforço”
 - monitoramento de ambientes de execução
 - consumo de dados: valores de mercados, etc
 - colaboração em grupos voláteis
- publish/subscribe e serviços de eventos
- diferentes políticas de distribuição de mensagens

- primitiva de envio para todo o grupo
 - possibilidade de implementar garantias e otimizações

exemplos

- MPI
- JGroups

Comunicação em Grupo: Problemas Novos

- alguns recebem e outros não
- processos recebem mensagens em ordens diferentes

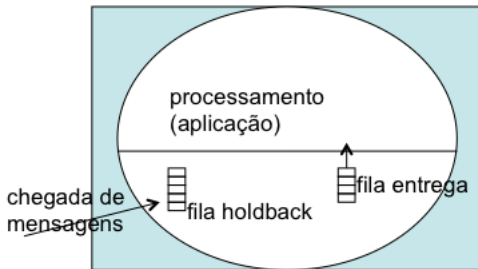
- confiabilidade
- ordenação

Implementação

- chegada X entrega

Garantia de entrega: Implementação

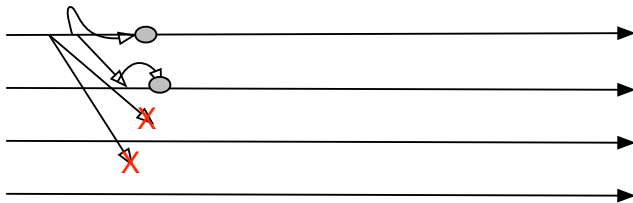
- uso de *holdback queue*
- em geral, mensagens com *identificadores únicos*
 - o que já existe no SO agora em bibliotecas ou middlewares
 - custos de espaço e processamento!



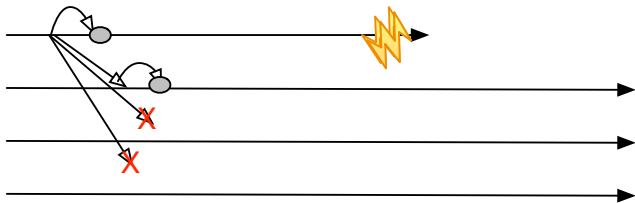
Exemplo: envio de msg para um grupo de processos



Exemplo: envio de msg para um grupo de processos



Exemplo: envio de msg para um grupo de processos



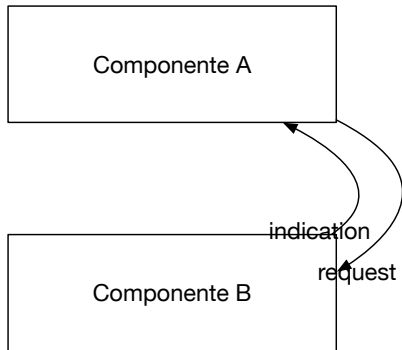
Níveis de garantia de entrega

- 1 *Melhor-esforço (best-effort)*: garantia de entrega entre processos corretos
- 2 *Confiável (reliable)*: garantia *all-or-nothing* mesmo se o emissor falhar
- 3 *Ordem xyz*: garantia de entrega na ordem *xyz*

Garantias de entrega: Implementação por algoritmos distribuídos

- notação orientada a eventos

Componentes orientados a eventos



- notação baseada em eventos assíncronos.
- cada componente é identificado por um nome, provê uma interface na forma de eventos que ele trata (*requests*) ou produz (*indications*). e garante um conjunto de propriedades.

- R. Guerraoui, L. Rodrigues. Reliable Distributed Programming. Springer. 2006.

Exemplo de código nos componentes

```
upon event <Event1 | att1, att2, ...> do
  ...faz algo...
  trigger <Event2 | att1, att2, ...>;// envia evento

upon event <Event3 | att1, att2, ...> do
  ...faz algo...
  trigger <Event4 | att1, att2, ...>; //envia evento
```

- Um processo envia uma msg em um passo de comunicação para todos os processos do sistema, incluindo ele mesmo
- Custo para garantir confiabilidade é apenas do lado do emissor (se ele falhar, nenhuma garantia de entrega é oferecida)

Interface e propriedades do “Bcast melhor-esforço”

Module:

Name: BestEffortBroadcast (beb).

Events:

Request: <bebBroadcast | m>: bcast m

Indication: <bebDeliver | src, m>: entrega m

Properties:

BEB1: Validade melhor-esforço: se P_i e P_j
são corretos, toda msg de P_i é entregue em P_j

BEB2: Não duplicação: nenhuma msg é entregue
mais de uma vez

BEB3: Não-criação: se a msg é entregue em P_j
então ela foi difundida por P_i

Algoritmo Bcast básico

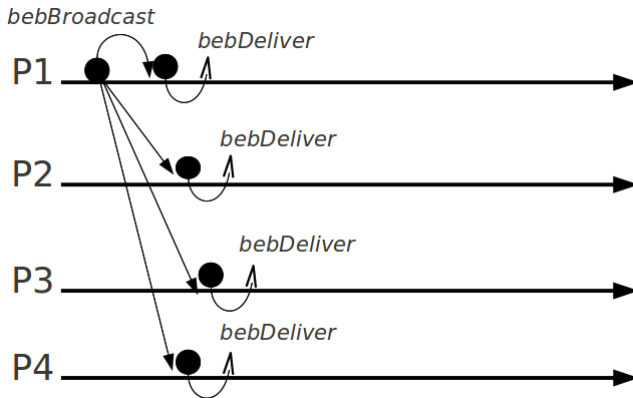
```
Implements: BestEffortBroadcast (beb)
```

```
Uses: PerfectPointToPointLinks (pp2p)
```

```
upon event <bebBroadcast | m> do  
  forall pi do  
    trigger <pp2pSend | pi, m>;
```

```
upon event <pp2pDeliver> | pi, m> do  
  trigger <bebDeliver | pi, m>;
```

Exemplo de execução de Bcast básico



Desempenho: o algoritmo requer um único passo de comunicação e troca N mensagens

Interface do enlace

Module:

Name: PerfectP2PLink

Events:

Request: < pp2pSend | dest, msg >

Indication: < pp2pDeliver | src, msg >

- entrega confiável: se p_i manda para p_j e nenhum deles falha, p_j em algum momento recebe
- ausência de duplicação de mensagens
- ausência de criação de mensagens

Propriedades do enlace

propriedades caras em alguns ambientes!

- 1 retransmite eternamente (didático mas sem bom desempenho...)
- 2 elimina duplicatas
- 3 entrega mesmo com falhas intermitentes...

- 1 se transmissor não falhar, garantia de entrega a todos
- 2 se transmissor falhar:
 - processos podem “discordar” sobre entrega de mensagem
 - broadcast de melhor esforço pode não ter transmitido para todos ou falha pode ter ocorrido antes de terem sido feitas as retransmissões necessárias

- todos os processos devem receber (tratar) o mesmo conjunto de mensagens
 - noção de *acordo*
- solução para modelo *fail-stop*

Interface e propriedades do *Bcast* confiável

Molule:

Name: Reliable Broadcast (rb).

Events:

Request: <rbBroadcast | m>: bcast m

Indication: <rbDeliver | src, m>: entrega m

Properties:

RB1: Validade: se P_i e P_j

são corretos, toda msg de P_i é entregue em P_j

RB2: Não duplicação: nenhuma msg é entregue
mais de uma vez

RB3: Não-criação: se a msg é entregue em P_j
então ela foi difundida por P_i

RB4: Acordo: Se msg é entregue em processo correto p_i ,
então m é entregue em algum momento em qualquer
processo correto p_j

Algoritmo Bcast confiável regular

Implements: ReliableBroadcast (rb)

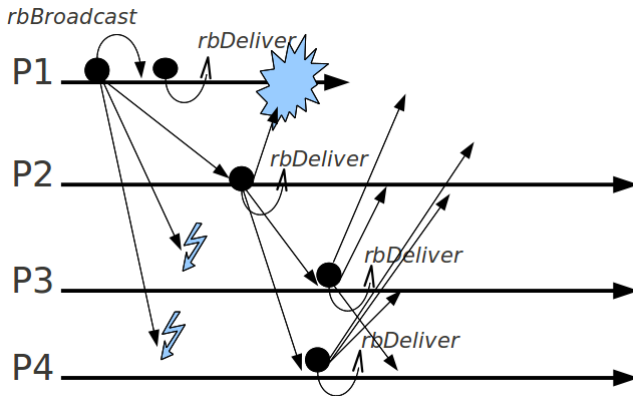
Uses: BestEffortBroadcast (beb)

```
upon event <Init> do
  delivered := {};
```

```
upon event <rbBroadcast | m> do
  delivered := delivered U {m};
  trigger <rbDeliver | self, m>;
  trigger <bebBroadcast | [DATA,self,m]>;
```

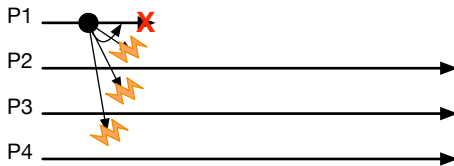
```
upon event <bebDeliver> | pi, [DATA,self,m] > do
  if(m not in delivered) do
    delivered := delivered U {m};
    trigger <rbDeliver | source_m, m>;
    trigger <bebBroadcast | [DATA,source_m,m]>;
```

Exemplo de execução de Bcast confiável regular



Broadcast confiável regular – limitação

- mensagem pode ser tratada no próprio processo emissor antes de chegar a outros, e em seguida emissor falhar



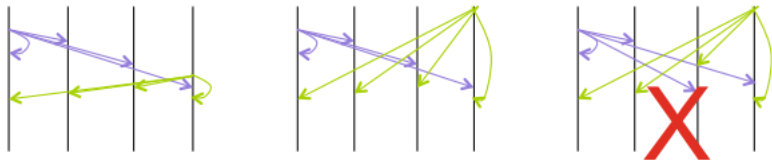
- pode causar problemas se reflexos no “mundo externo”

Broadcast confiável uniforme

- mensagem só é entregue a aplicação depois que todos os processos no grupo confirmam recebimento
- necessidade de detector de falhas perfeito
 - gera alertas de falhas de processos
- acoplamento fortíssimo entre processos

- As soluções anteriores não garantem ordem de entrega das mensagens enviadas por processos diferentes (**pp2p pode garantir ordem de entrega de msgs de 1 mesmo processo**)
- Algumas aplicações precisam de garantias de ordem de entrega.
 - ordem total
 - ordem causal

Ordem Total



- Não importa a ordem de entrega, mas deve ser a mesma em todos os processos do grupo.

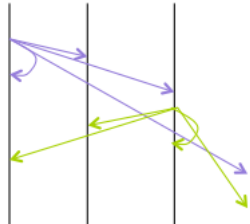
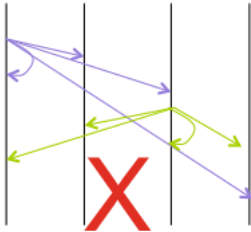
Ordem total com sequenciador

Ordem total com votação de ordem

Ordem Causal

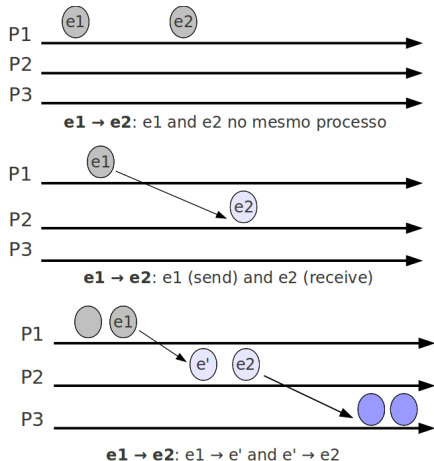
possível relação de causalidade

- uma mensagem pode ter sido consequencia de outra ainda não vista



Causalidade em eventos distribuídos...

- $a \prec b$: o evento a ocorre antes do evento b se
 - a ocorre antes de b em um mesmo processo P
 - $a = \text{send}(m)$ no processo P e $b = \text{receive}(m)$ no processo Q
 - existe c tal que a precede c e c precede b
- $a \parallel b$: os eventos a e b são concorrentes



Interface e propriedades do “Bcast ordem causal”

Molule:

Name: ReliableCausalOrder (rco)

Events:

Request: <rcoBroadcast | m>: bcast m

Indication: <rcoDeliver | src, m>: entrega m

Properties:

CB: Entrega causal: m2 só é entregue em Pi

se toda msg mi tal que mi precede m2 foram entregues

RB1: Validade

RB2: Não-duplicação

RB3: não-criação

RB4: Acordo

Algoritmo Bcast ordem causal

```
Uses: ReliableBroadcast (rb)
upon event <Init> do
  delivered := 0; past := 0;
upon event <rcoBroadcast | m> do
  trigger <rbBroadcast | [DATA,past,m]>;
  past := past U {(self,m)};
upon event <rbDeliver> | pi, [DATA,past_m,m] > do
  if(m not in delivered) then
    forall (s_n, n) in past_m do
      if(n not in delivered) then
        trigger <rcoDeliver | s_n, n>;
        delivered := delivered U {n};
        past := past U {(s_n,n)};
      trigger <rcoDeliver | pi, m>;
      delivered := delivered U {m};
      past := past U {(pi,m)};
```

custos proibitivos... cada mensagem carrega todas as que a precedem causalmente

Relógio Lógico (Lamport)

- conceito de relógio lógico modela ordenação (parcial) de eventos
- cada evento associado a um valor $r(e_i)$ de forma que $e_i \prec e_j \Rightarrow r(e_i) < r(e_j)$

- cada processo P mantém um contador, inicialmente $RL_P = 0$
- a cada evento e , P associa um valor $RL_P(e)$ e incrementa seu contador
- se evento e é envio de mensagem, P acrescenta campo *timestamp* com valor $RL_P(e)$
- ao receber mensagem M com timestamp TS , processo Q faz $RL_Q = \max(RL_Q, TS) + 1$

exemplo

- algoritmo de exclusão mútua distribuída

Broadcast causal com espera

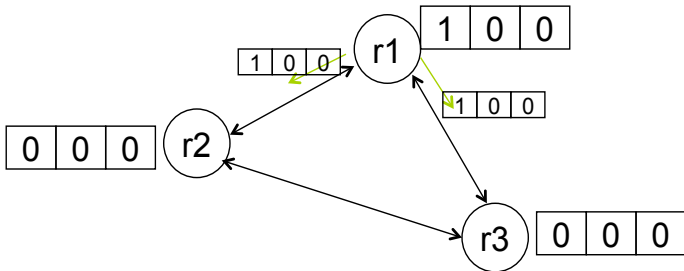
- uso de relógio lógico e timestamps: funciona se tempo de entrega de mensagens tem limite máximo

- uso de mensagens “posteriores” para validar mensagens com timestamp TS
- tempo de quarentena (dependente de latência da rede) em fila de *holdback*

Vetor de timestamps: idéia básica

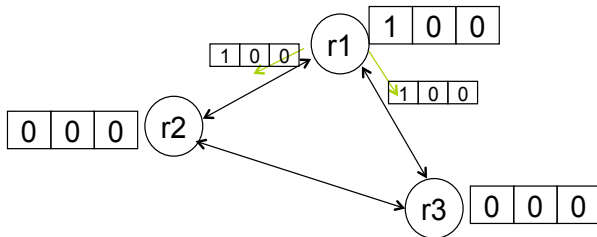
- Dados N processos, usa-se um vetor de N elementos (ao invés de um valor escalar)
- A cada evento “e”, associa-se um **vetor de timestamps** cujo i -ésimo elemento indica quantos mensagens do processo i já foram vistas

Exemplo de uso de VT para ordenação de eventos

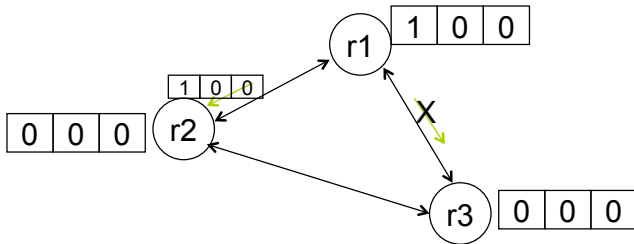


- é necessário não apenas estabelecer ordem mas garantir que foram vistas todas as mensagens que precedem causalmente determinada mensagem!

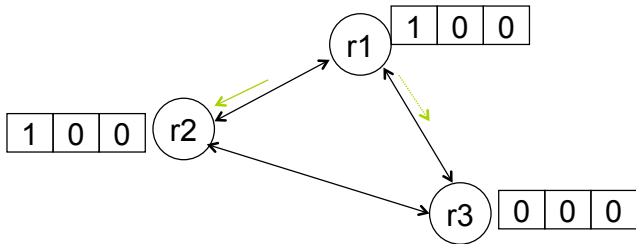
Exemplo de entrega em ordem causal



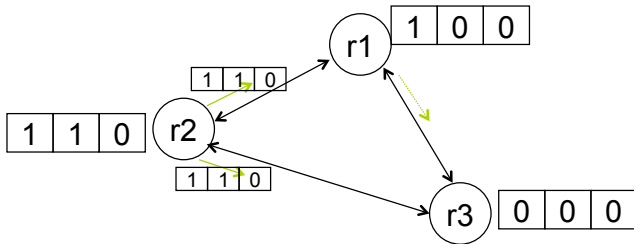
Exemplo de entrega em ordem causal



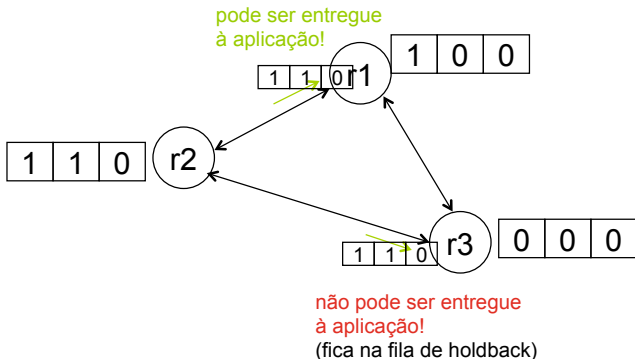
Exemplo de entrega em ordem causal



Exemplo de entrega em ordem causal



Exemplo de entrega em ordem causal



- a entrega deve ser postergada até as mensagens anteriores serem vistas
- ack's ou reenvios sob demanda

- algoritmos de ordenação supõem que cada participante conhece os membros do grupo
- grupos estáticos x dinâmicos (falhas, adesões, etc)

serviço de *membership*

- alerta participantes sobre saídas e entradas

- R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming* Springer, 2006
- A. Tanenbaum e M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2007. (seções 6.2 e 8.4)