Comunicação Confiável

Ricardo Anido

3 de Junho de 2014

Terminologia

Um componente provê *serviços* para seus clientes. Para prover o serviço um componente pode usar serviços de outro componente *Rightarrow* um componente pode *depender* de outro componente. Algumas propriedades:

- Availability – disponibilidade para uso

Terminologia

Um componente provê *serviços* para seus clientes. Para prover o serviço um componente pode usar serviços de outro componente *Rightarrow* um componente pode *depender* de outro componente. Algumas propriedades:

- Availability – disponibilidade para uso
- Reliability – confiabilidade no uso

Terminologia

Um componente provê *serviços* para seus clientes. Para prover o serviço um componente pode usar serviços de outro componente *Rightarrow* um componente pode *depender* de outro componente. Algumas propriedades:

- Availability – disponibilidade para uso
- Reliability – confiabilidade no uso
- Safety – segurança no uso, probabilidade de catástrofes ocorrerem

Terminologia

Um componente provê *serviços* para seus clientes. Para prover o serviço um componente pode usar serviços de outro componente *Rightarrow* um componente pode *depender* de outro componente. Algumas propriedades:

- Availability – disponibilidade para uso
- Reliability – confiabilidade no uso
- Safety – segurança no uso, probabilidade de catástrofes ocorrerem
- Maintanability – facilidade de correção do sistema

Terminologia

Diferenças sutis (em inglês):

- **Failure** – Componente falha se deixa de se comportar conforme sua *especificação*.

Terminologia

Diferenças sutis (em inglês):

- **Failure** – Componente falha se deixa de se comportar conforme sua *especificação*.
- **Error** – a parte do estado do componente que pode levar a uma falha.

Terminologia

Diferenças sutis (em inglês):

- **Failure** – Componente falha se deixa de se comportar conforme sua *especificação*.
- **Error** – a parte do estado do componente que pode levar a uma falha.
- **Fault** – a causa do erro.

Terminologia

O que se pode fazer quanto a falhas:

- **Fault prevention** – previnir contra falhar

Terminologia

O que se pode fazer quanto a falhas:

- **Fault prevention** – previnir contra falhar
- **Fault tolerance** – construir um componente de maneira a mascarar a ocorrência de falhas.

Terminologia

O que se pode fazer quanto a falhas:

- **Fault prevention** – previnir contra falhar
- **Fault tolerance** – construir um componente de maneira a mascarar a ocorrência de falhas.
- **Fault removal** – reduzir a presença de falhas graves

Terminologia

O que se pode fazer quanto a falhas:

- **Fault prevention** – previnir contra falhar
- **Fault tolerance** – construir um componente de maneira a mascarar a ocorrência de falhas.
- **Fault removal** – reduzir a presença de falhas graves
- **Fault forecasting** – prever o número, incidência e consequência de falhas.

Modelos de falhas

- **Crash** – Componente para, mas se comporta corretamente antes da parada.

Modelos de falhas

- **Crash** – Componente para, mas se comporta corretamente antes da parada.
- **Omission** – Componente não responde alguma requisição, ou não envia alguma mensagem.

Modelos de falhas

- **Crash** – Componente para, mas se comporta corretamente antes da parada.
- **Omission** – Componente não responde alguma requisição, ou não envia alguma mensagem.
- **Timing** – Componente produz saída é correta, mas em momento inesperado (antes ou depois do especificado)

Modelos de falhas

- **Crash** – Componente para, mas se comporta corretamente antes da parada.
- **Omission** – Componente não responde alguma requisição, ou não envia alguma mensagem.
- **Timing** – Componente produz saída é correta, mas em momento inesperado (antes ou depois do especificado)
- **Output** – Componente produz saída incorreta.

Modelos de falhas

- **Crash** – Componente para, mas se comporta corretamente antes da parada.
- **Omission** – Componente não responde alguma requisição, ou não envia alguma mensagem.
- **Timing** – Componente produz saída é correta, mas em momento inesperado (antes ou depois do especificado)
- **Output** – Componente produz saída incorreta.
- **Bizantine** – Componente pode fazer qualquer coisa.

Crash Failures

Problema: cliente não consegue distinguir entre um componente lento ou com falha de crash (processo ou canal). Podemos supor:

- **Fail-silent** – componente exibe falhas de omissão ou crash; clientes não conseguem distinguir esses dois modos de falha.

Crash Failures

Problema: cliente não consegue distinguir entre um componente lento ou com falha de crash (processo ou canal). Podemos supor:

- **Fail-silent** – componente exibe falhas de omissão ou crash; clientes não conseguem distinguir esses dois modos de falha.
- **Fail-stop** – componente pode falhar, mas falha pode ser detectada.

Crash Failures

Problema: cliente não consegue distinguir entre um componente lento ou com falha de crash (processo ou canal). Podemos supor:

- **Fail-silent** – componente exibe falhas de omissão ou crash; clientes não conseguem distinguir esses dois modos de falha.
- **Fail-stop** – componente pode falhar, mas falha pode ser detectada.
- **Fail-safe** – componente pode falhar, mas apenas falhas *benignas*.

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.
  - Modelo de *conversações distribuídas*, veremos adiante.

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.
  - Modelo de *conversações distribuídas*, veremos adiante.
- **Redundância de componentes** – por exemplo canais e processos replicados

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.
  - Modelo de *conversações distribuídas*, veremos adiante.
- **Redundância de componentes** – por exemplo canais e processos replicados
  - Mas tem que haver *consenso* entre os participantes.

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.
  - Modelo de *conversações distribuídas*, veremos adiante.
- **Redundância de componentes** – por exemplo canais e processos replicados
  - Mas tem que haver *consenso* entre os participantes.
  - Exemplo: *N-redundancy*, usado em sistemas críticos, como aviação.

Fault-tolerance

Qualquer tratamento de falhas exige redundância

- **Redundância no tempo** – por exemplo re-executar uma operação
  - Blocos `try... except` são exemplos de tratamento embutido na linguagem – mas tem que tratar o caso distribuído.
  - Modelo de *conversações distribuídas*, veremos adiante.
- **Redundância de componentes** – por exemplo canais e processos replicados
  - Mas tem que haver *consenso* entre os participantes.
  - Exemplo: *N-redundancy*, usado em sistemas críticos, como aviação.
  - Votação? Commit a cada passo? Caro e bloqueia! Veremos adiante.

Failure detection

Básicamente, falhas são detectadas por mecanismos de *timeout*.

- Acerto do valor do timeout é difícil e dependente da aplicação.

Failure detection

Básicamente, falhas são detectadas por mecanismos de *timeout*.

- Acerto do valor do timeout é difícil e dependente da aplicação.
- Não conseguimos distinguir falhas de processos de falhas de comunicação

Failure detection

Básicamente, falhas são detectadas por mecanismos de *timeout*.

- Acerto do valor do timeout é difícil e dependente da aplicação.
- Não conseguimos distinguir falhas de processos de falhas de comunicação
- Como propagar a notificação da falha?

Failure detection

Básicamente, falhas são detectadas por mecanismos de *timeout*.

- Acerto do valor do timeout é difícil e dependente da aplicação.
- Não conseguimos distinguir falhas de processos de falhas de comunicação
- Como propagar a notificação da falha?
  - Ao detectar a falha, para.

Failure detection

Básicamente, falhas são detectadas por mecanismos de *timeout*.

- Acerto do valor do timeout é difícil e dependente da aplicação.
- Não conseguimos distinguir falhas de processos de falhas de comunicação
- Como propagar a notificação da falha?
  - Ao detectar a falha, para.
  - Dissemina o conhecimento da falha (*gossiping*, *lazy-forwarding*).

Failure detection

No caso de canais de comunicação, outras técnicas de redundância são usadas adicionalmente;

- Deteção de erros

Failure detection

No caso de canais de comunicação, outras técnicas de redundância são usadas adicionalmente;

- Deteção de erros
  - bits de redundância nas mensagens para detectar erros.

Failure detection

No caso de canais de comunicação, outras técnicas de redundância são usadas adicionalmente;

- Deteção de erros
  - bits de redundância nas mensagens para detectar erros.
  - números nos pacotes para detectar perda de pacotes.

Failure detection

No caso de canais de comunicação, outras técnicas de redundância são usadas adicionalmente;

- Deteção de erros
  - bits de redundância nas mensagens para detectar erros.
  - números nos pacotes para detectar perda de pacotes.

Failure detection

No caso de canais de comunicação, outras técnicas de redundância são usadas adicionalmente;

- Deteção de erros
  - bits de redundância nas mensagens para detectar erros.
  - números nos pacotes para detectar perda de pacotes.
- Correção de erros
  - bits de redundância nas mensagens que permita correção de erros.
  - solicitar retransmissões de pacotes perdidos.

RPC Confiável

Comunicação RPC: o que pode dar errado?

1. Cliente não encontra o servidor

RPC Confiável

Comunicação RPC: o que pode dar errado?

1. Cliente não encontra o servidor
2. Requisição de cliente é perdida

RPC Confiável

Comunicação RPC: o que pode dar errado?

1. Cliente não encontra o servidor
2. Requisição de cliente é perdida
3. Crash do servidor

RPC Confiável

Comunicação RPC: o que pode dar errado?

1. Cliente não encontra o servidor
2. Requisição de cliente é perdida
3. Crash do servidor
4. Resposta do servidor é perdida

RPC Confiável

Comunicação RPC: o que pode dar errado?

1. Cliente não encontra o servidor
2. Requisição de cliente é perdida
3. Crash do servidor
4. Resposta do servidor é perdida
5. Crash do cliente

RPC Confiável

Comunicação RPC, soluções:

1. Cliente não encontra o servidor
2. Requisição de cliente é perdida
3. Crash do servidor
4. Resposta do servidor é perdida
5. Crash do cliente

RPC Confiável

Comunicação RPC, soluções:

1. Cliente não encontra o servidor
   - Avisa cliente
2. Requisição de cliente é perdida

3. Crash do servidor
4. Resposta do servidor é perdida
5. Crash do cliente

RPC Confiável

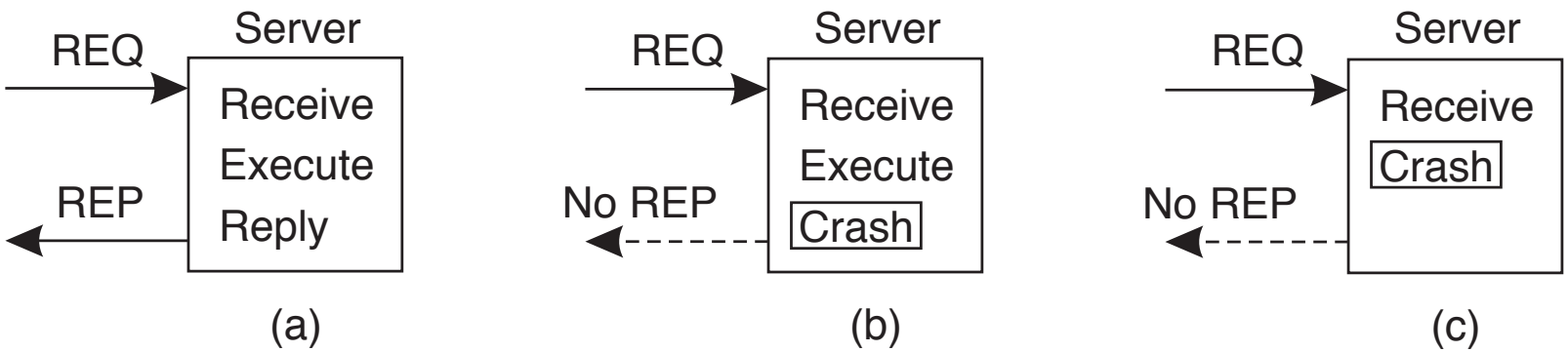
Comunicação RPC, soluções:

1. Cliente não encontra o servidor
   - Avisa cliente
2. Requisição de cliente é perdida
   - Re-transmite requisição
3. Crash do servidor
4. Resposta do servidor é perdida
5. Crash do cliente

RPC Confiável

Comunicação RPC, soluções:

- 3. Crash do servidor
  Mais difícil de tratar, não sabemos o que já foi executado:



RPC Confiável

Precisamos definir o que se espera do servidor:

- **At-least-once-semantics:** Servidor garante que vai executar requisição ao menos uma vez, independente de erros.

RPC Confiável

Precisamos definir o que se espera do servidor:

- **At-least-once-semantics:** Servidor garante que vai executar requisição ao menos uma vez, independente de erros.
- **At-most-once-semantics:** Servidor garante que vai executar requisição no máximo uma vez.

RPC Confiável

Comunicação RPC, soluções:

- 4. Resposta do servidor é perdida
  Detectar perda de resposta pode ser difícil, mas também tem que considerar se servidor falhou (antes ou depois de executar a requisição).

RPC Confiável

Comunicação RPC, soluções:

- 4. Resposta do servidor é perdida
  Detectar perda de resposta pode ser difícil, mas também tem que considerar se servidor falhou (antes ou depois de executar a requisição).
- **Solução**: Nenhuma exceto talvez fazer todas as operações **idempotentes** (operação que, se repetida não causa diferença no estado do sistema).

RPC Confiável

Comunicação RPC, soluções:

- 5. Crash do cliente
  Servidor está gastando recursos inutilmente (computação *órfã*).

RPC Confiável

Comunicação RPC, soluções:

- 5. Crash do cliente
  Servidor está gastando recursos inutilmente (computação *órfã*).
- **Soluções possíveis**:

RPC Confiável

Comunicação RPC, soluções:

- 5. Crash do cliente
  Servidor está gastando recursos inutilmente (computação *órfã*).
- **Soluções possíveis**:
  - Considerar que computações órfãs serão desconsideradas (ou re-executadas, ou voltam para estado de checkpoint) quando cliente se recuperar.

RPC Confiável

Comunicação RPC, soluções:

- 5. Crash do cliente
  Servidor está gastando recursos inutilmente (computação *órfã*).
- **Soluções possíveis**:
  - Considerar que computações órfãs serão desconsideradas (ou re-executadas, ou voltam para estado de checkpoint) quando cliente se recuperar.
  - Exigir que computações completem em T unidades de tempo; computações mais velhas são simplesmente removidas

Comunicação Confiável

- Temos um **canal multicast** *c* com dois grupos (possivelmente com overlap):

Comunicação Confiável

- Temos um **canal multicast** c com dois grupos (possivelmente com overlap):
  - O grupo **Sender** $SND(c)$ de processos que *submetem* mensagens para o canal c.

Comunicação Confiável

- Temos um **canal multicast** c com dois grupos (possivelmente com overlap):
  - O grupo **Sender** $SND(c)$ de processos que *submetem* mensagens para o canal c.
  - O grupo **Receiver** $RCV(c)$ de processos que podem receber mensagens do canal c

Comunicação Confiável

- **Confiabilidade simples:** Se processo $P \in RCV(c)$ no momento em que a mensagem m foi submetida a c, e P não deixa $RCV(c)$, então m deve ser entregue a P.

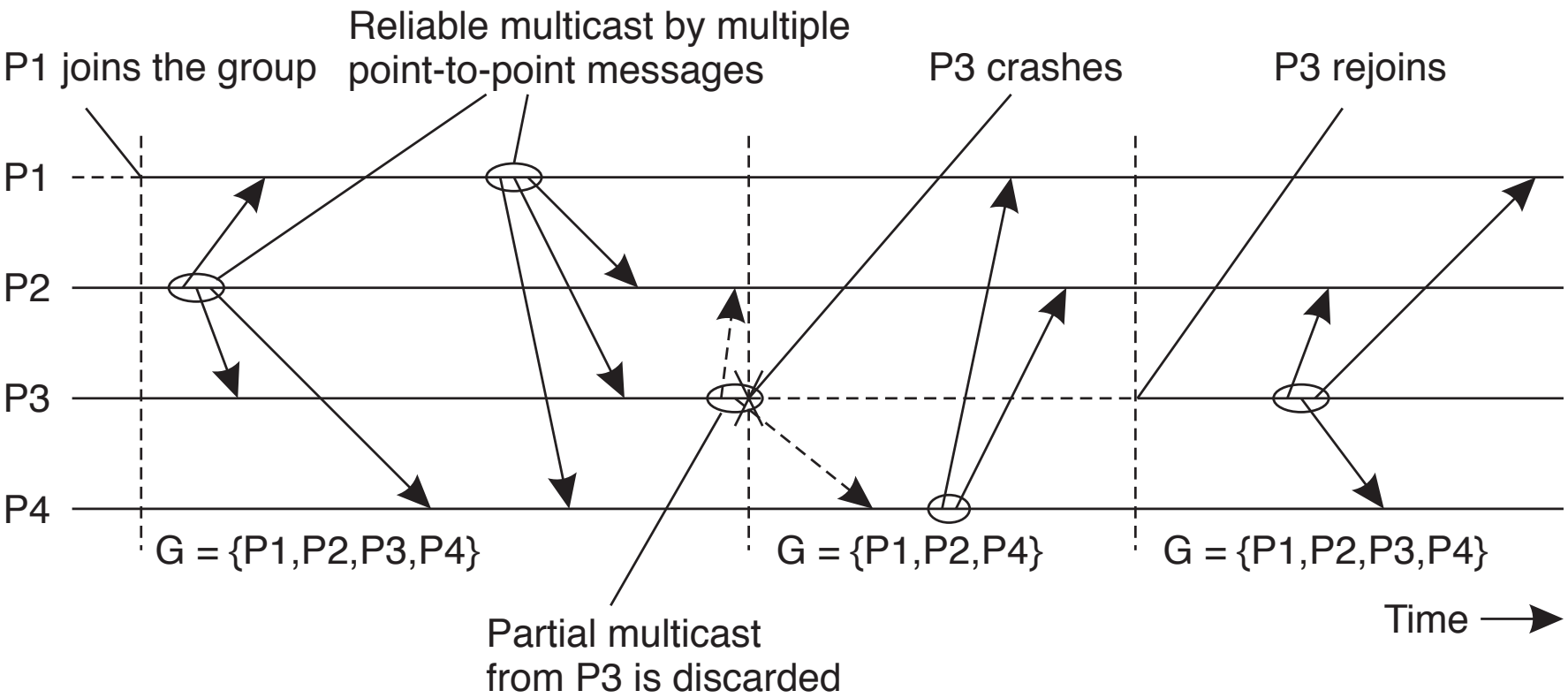
Comunicação Confiável

- **Confiabilidade simples:** Se processo $P \in RCV(c)$ no momento em que a mensagem m foi submetida a c, e P não deixa $RCV(c)$, então m deve ser entregue a P.
- **Multicast Atômico:** Mensagem m submetida ao canal c é entregue para o processo $P \in RCV(c)$ se e somente se m é entregue para *todos* os membros de $RCV(c)$.

Comunicação Confiável

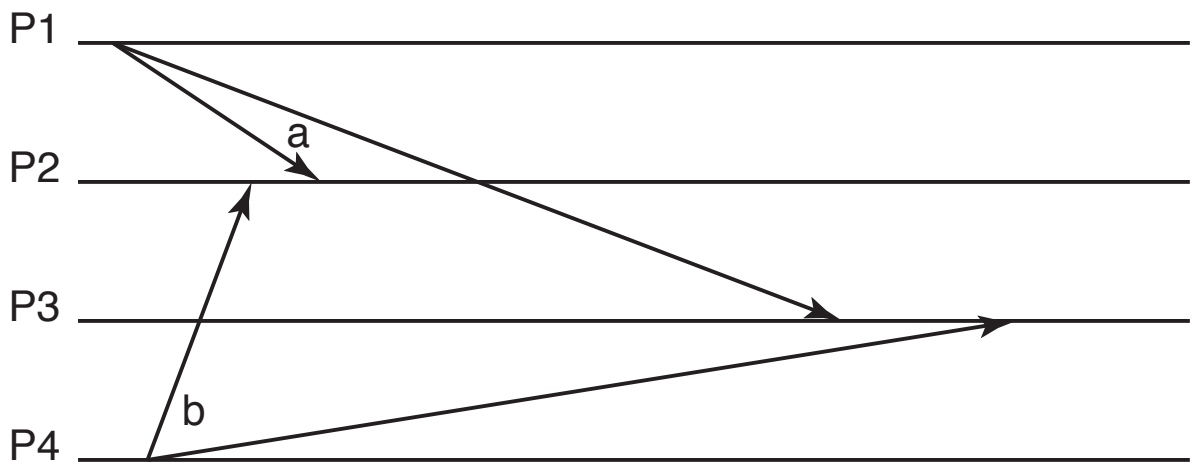
- **Confiabilidade simples:** Se processo $P \in RCV(c)$ no momento em que a mensagem m foi submetida a c, e P não deixa $RCV(c)$, então m deve ser entregue a P.
- **Multicast Atômico:** Mensagem m submetida ao canal c é entregue para o processo $P \in RCV(c)$ se e somente se m é entregue para *todos* os membros de $RCV(c)$.
- **Multicast Confiável:** Multicast atômico, e mensagens entregues na mesma ordem a todos os processos. Ou seja, se a mensagem m_1 é entregue antes da mensagem m_2 para algum processo $P \in RCV(c)$, então m_1 é entregue antes de m_2 para todo $P \in RCV(c)$.

Multicast Atômico



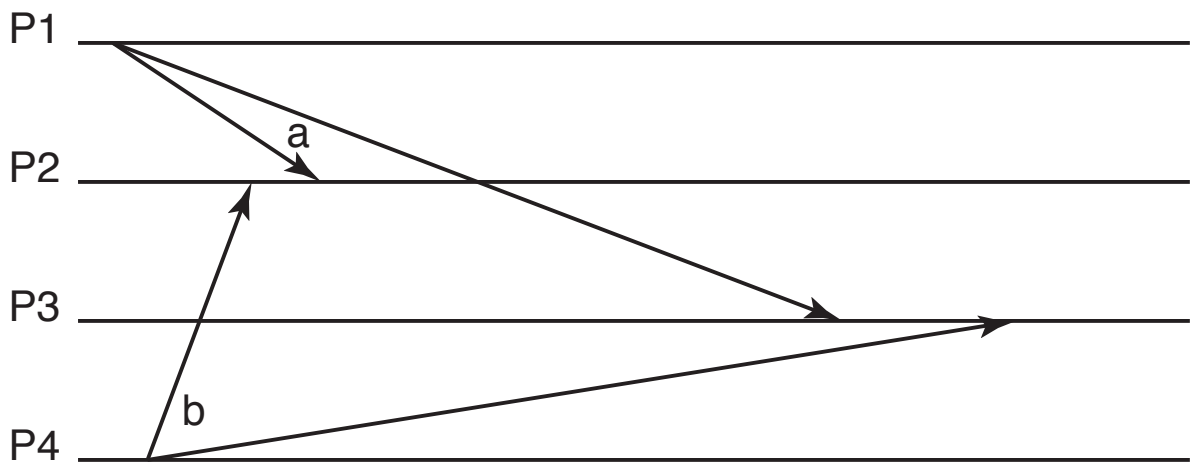
Multicast Atômico

Multicast atômico, mas apenas as mensagens *interessantes* para o exemplo são mostradas:



Multicast Atômico

Multicast atômico, mas apenas as mensagens *interessantes* para o exemplo são mostradas:



- P2 recebe mensagens na ordem b,a
- P3 recebe mensagens na ordem a,b

Multicast Atômico e Confiável

Abordagem: resolver o problema em termos de Grupo de Processos, e mudanças no Grupo quando ocorrem falhas ou recuperações.

- Mensagem deve ser entregue para todos os processos não falhos do grupo corrente.

Multicast Atômico e Confiável

Abordagem: resolver o problema em termos de Grupo de Processos, e mudanças no Grupo quando ocorrem falhas ou recuperações.

- Mensagem deve ser entregue para todos os processos não falhos do grupo corrente.
- Todos os processos devem concordar quanto ao grupo – problema "classico", *Group Membership*.

Multicast Atômico e Confiável

Abordagem: resolver o problema em termos de Grupo de Processos, e mudanças no Grupo quando ocorrem falhas ou recuperações.

- Mensagem deve ser entregue para todos os processos não falhos do grupo corrente.
- Todos os processos devem concordar quanto ao grupo – problema "classico", *Group Membership*.
- Todos os processos devem concordar quanto à entrega de cada mensagem (e ordem, no caso de multicast confiável).