ACH 2147 — Desenvolvimento de Sistemas de Informação Distribuídos

Aula 25: Tolerância a Falhas (parte 3)

Prof. Renan Alves

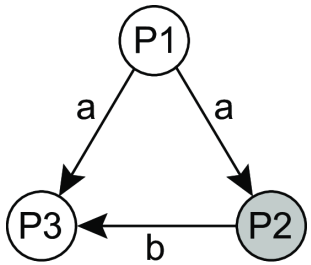
Escola de Artes, Ciências e Humanidades — EACH — USP

10/06/2024

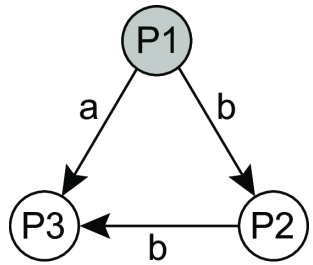
Consenso sob semânticas de falhas arbitrárias

Essência

Consideramos grupos de processos nos quais a comunicação entre processos é inconsistente.



Encaminhamento impróprio



Mensagens diferentes

Consenso sob semântica de falha arbitrária

Modelo de sistema

- Consideramos um primário P e $n-1$ backups $B_1, \ldots, B_{n-1}$.
- Um cliente envia $v \in \{T, F\}$ para P
- Mensagens podem ser perdidas, mas isso pode ser detectado.
- Mensagens não podem ser corrompidas de forma indetectável.
- Um receptor de uma mensagem pode detectar seu remetente de forma confiável.

Acordo bizantino

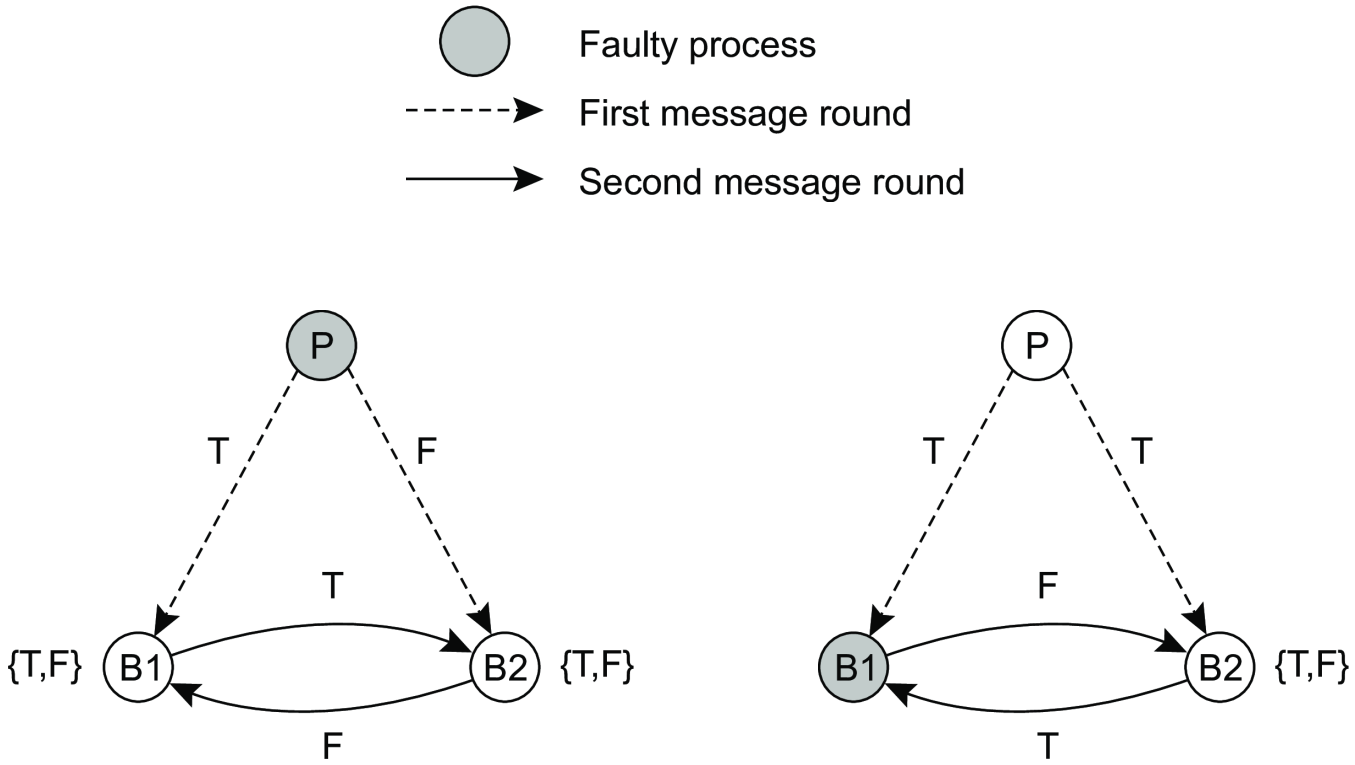
Acordo bizantino: requisitos

BA1: Todo processo de backup não falho armazena o mesmo valor.
BA2: Se o primário não falhar, então todo processo de backup não falho armazena exatamente o que o primário enviou.

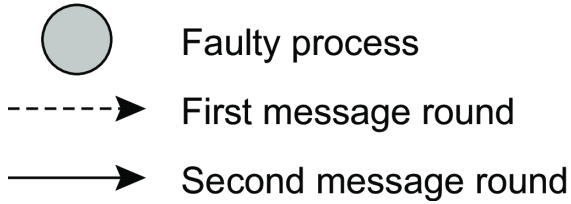
Observação

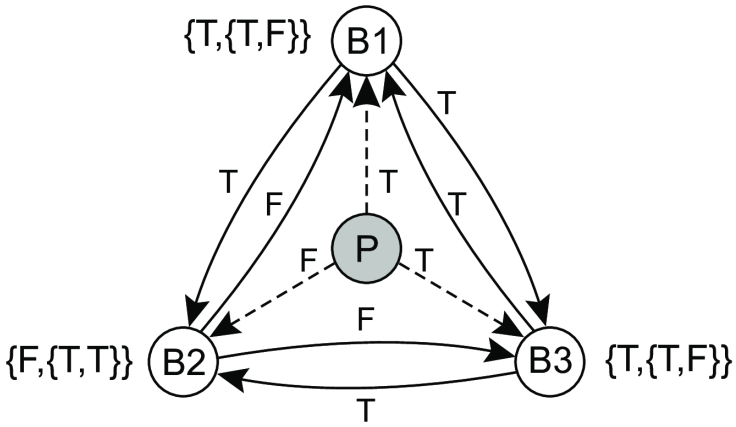
- Primário falho $\Rightarrow$ BA1 diz que os backups armazenam o mesmo valor, porém potencialmente diferente do valor originalmente enviado pelo cliente (e portanto errado).
- Primário não falho $\Rightarrow$ satisfazer BA2 implica que BA1 é satisfeito.

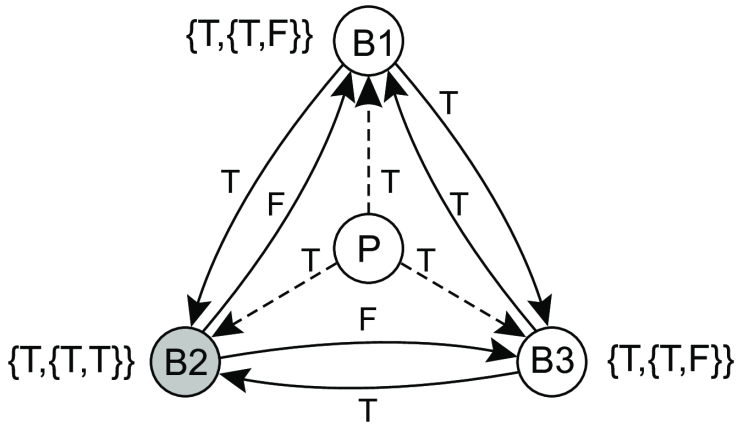
Por que ter **3k** processos não é suficiente



Por que ter **3k + 1** processos é suficiente







Tolerância a Falhas Bizantinas Prática (TFBP)

Background

Uma das primeiras soluções que conseguiu tolerância a falhas bizantinas mantendo o desempenho aceitável.

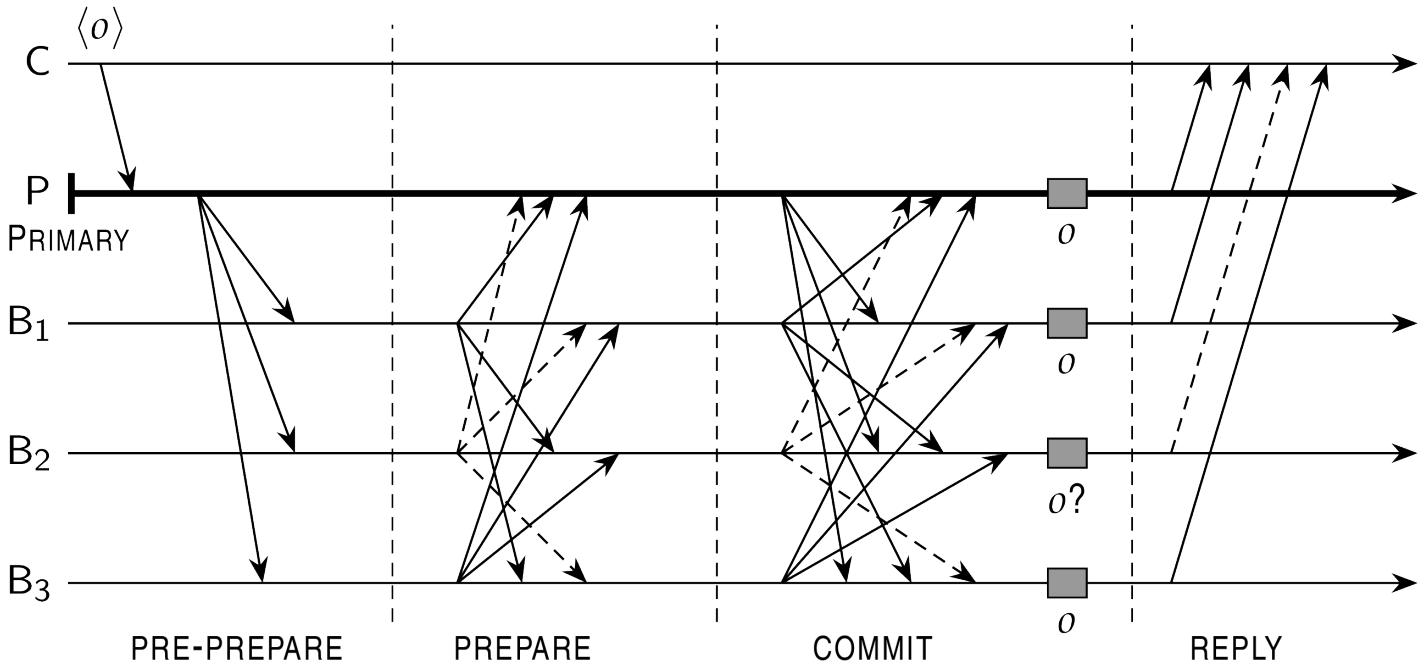
Pressupostos

- Um servidor pode apresentar falhas arbitrárias
- Mensagens podem ser perdidas, atrasadas e recebidas fora de ordem
- Mensagens têm um remetente identificável (ou seja, são assinadas)
- Modelo de execução parcialmente síncrono

Essência

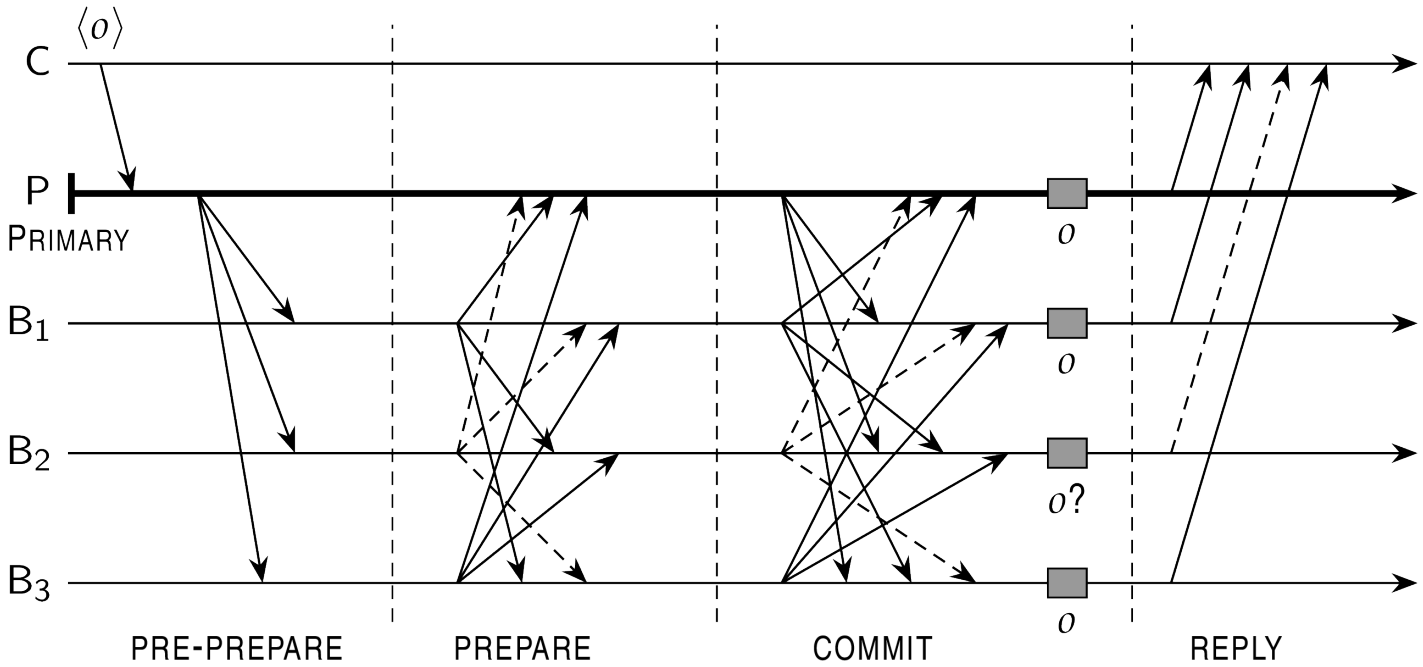
Uma abordagem de primário-backup com $3k + 1$ servidores.

TFBP: quatro fases



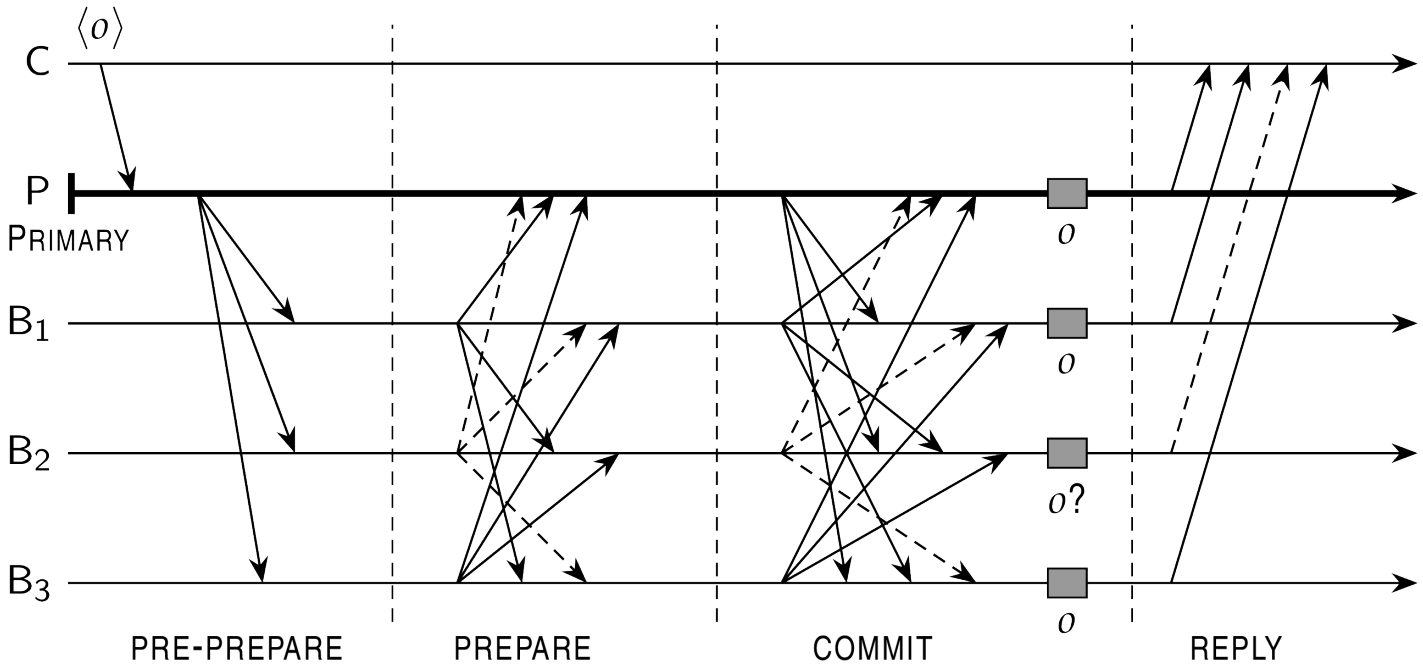
- C é o cliente
- P é o primário
- B_1, B_2, B_3 são backups
- Suponha que B_2 esteja com falha

TFBP: quatro fases



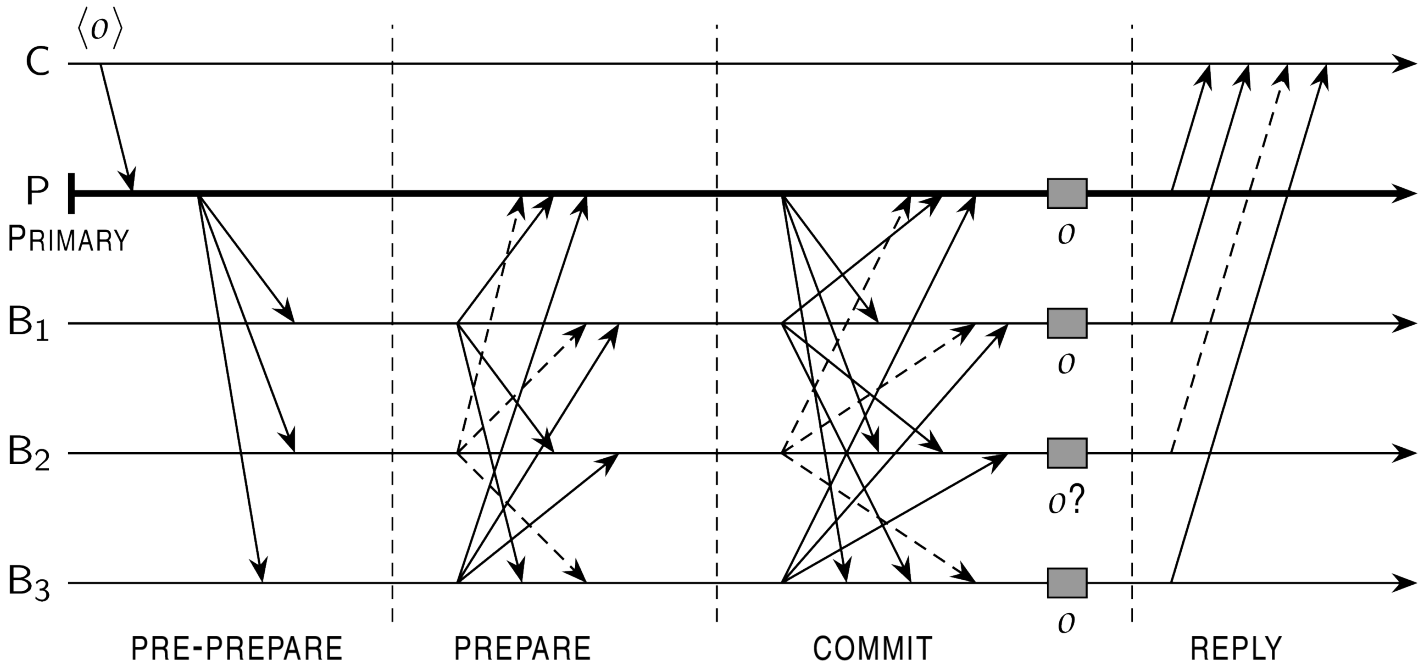
- Todos os servidores assumem que estão trabalhando em uma visão atual v.
- C solicita que a operação o seja executada
- P põe um timestamp em o e envia PRE-PREPARE(t, v, o)
- O backup B_i aceita a mensagem de pré-preparo se também estiver em v e não tiver aceitado uma operação com timestamp t anteriormente.

TFBP: quatro fases



- B_i transmite PREPARE(t, v, o) para todos (incluindo o primário)
- Nota: um servidor não falho eventualmente registrará $2k$ mensagens PREPARE(t, v, o) (incluindo a sua própria) $\Rightarrow$ consenso sobre a ordenação de o.
- Nota: não importa o que o B_2 (em falha) envia, ele não pode afetar as decisões conjuntas de P, B_1, B_3.

TFBP: quatro fases



- Todos os servidores transmitem COMMIT(t, v, o)
- O commit é necessário para garantir que o possa ser executado agora, ou seja, na visão atual v.
- Quando $2k$ mensagens forem coletadas, excluindo a sua própria, o servidor pode executar o com segurança e responder ao cliente.

TFBP: quando o primário falha

Questão

Quando um backup detecta que o primário falhou, ele transmitirá uma mudança de visão para a visão $v+1$. Precisamos garantir que qualquer solicitação pendente seja executada uma vez e apenas uma vez por todos os servidores não falhos. A operação precisa ser transferida para a nova visão.

Procedimento

- O próximo primário P^* é conhecido de forma determinística
- Um servidor de backup transmite VIEW-CHANGE$(v+1, \mathbf{P})$: **P** é o conjunto de PREPARE que ele havia enviado.
- P^* espera por $2k+1$ mensagens de mudança de visão, com $\mathbf{X} = \bigcup \mathbf{P}$ contendo todas as preparações enviadas anteriormente.
- P^* envia NEW-VIEW(v+1,**X**,**O**) com **O** um novo conjunto de mensagens de pré-preparo.
- Essência: isso permite que os backups não falhos reproduzam o que ocorreu na visão anterior, se necessário, e tragam *o* para a nova visão $v+1$.

Realizando tolerância a falhas

Observação

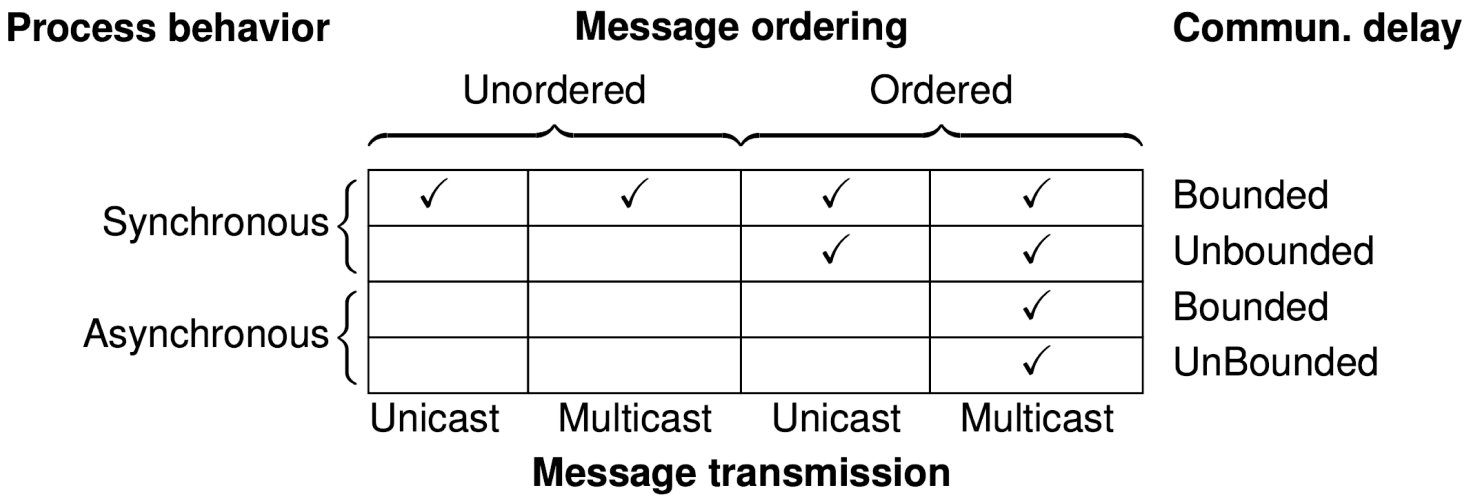
Considerando que os membros de um grupo de processos tolerantes a falhas são tão fortemente acoplados, podemos encontrar problemas de desempenho consideráveis, talvez até situações em que realizar a tolerância a falhas seja impossível.

Questão

Existem limitações sobre que pode ser alcançado facilmente?

- O que é necessário para permitir alcançar consenso?
- O que acontece quando os grupos são particionados?

Consenso distribuído: quando pode ser alcançado



Requisitos formais para consenso

- Processos produzem o mesmo valor de saída
- Cada valor de saída deve ser válido
- Cada processo deve eventualmente fornecer uma saída

Consistência, disponibilidade e particionamento

Teorema CAP

Qualquer sistema de rede que fornece dados compartilhados pode fornecer apenas duas das seguintes três propriedades:

- C: consistência, pela qual um item de dados compartilhado e replicado aparece como uma única cópia atualizada
- A: disponibilidade, pela qual as atualizações serão sempre eventualmente executadas
- P: Tolerância ao particionamento do grupo de processos.

Conclusão

Em uma rede sujeita a falhas de comunicação, é impossível fazer uma memória compartilhada com leitura e escrita atômicas que garanta uma resposta a todas as solicitações.

Intuição do teorema CAP

Situação simples: dois processos (P e Q) interagindo

- P e Q não podem mais se comunicar:
  - Permitir que P e Q continuem $\Rightarrow$ sem consistência
  - Permitir que apenas um de P, Q continue $\Rightarrow$ sem disponibilidade
- devem ser assumido que P e Q continuam a comunicação $\Rightarrow$ sem particionamento permitido.

Intuição do teorema CAP

Situação simples: dois processos (P e Q) interagindo

- P e Q não podem mais se comunicar:
  - Permitir que P e Q continuem $\Rightarrow$ sem consistência
  - Permitir que apenas um de P, Q continue $\Rightarrow$ sem disponibilidade
- devem ser assumido que P e Q continuam a comunicação $\Rightarrow$ sem particionamento permitido.

Questão fundamental

Quais são as consequências práticas do teorema CAP?

Detecção de falhas

Questão

Como podemos detectar de forma confiável que um processo realmente travou?

Modelo geral

- Cada processo é equipado com um módulo de detecção de falhas
- Um processo P sonda outro processo Q por uma reação
- Se Q reagir: Q é considerado vivo (por P)
- Se Q não reagir dentro de t unidades de tempo: Q é suspeito de ter travado

Observação:

Para um sistema síncrono: uma falha suspeita $\equiv$ uma falha conhecida

Detecção de falhas

Possível implementação simples

- Se P não receber heartbeat de Q dentro do tempo t: P suspeita de Q.
- Se Q posteriormente enviar uma mensagem (que é recebida por P):
  - P para de suspeitar de Q
  - P aumenta o valor do timeout t
- Nota: se Q realmente travou, P continuará suspeitando de Q.

Possível melhoria

Fazer com que os nós colaborem