Universidade da Beira Interior
Mestrado em Engenharia Informática
Sistemas Distribuídos e Tolerância a Falhas

Consensos

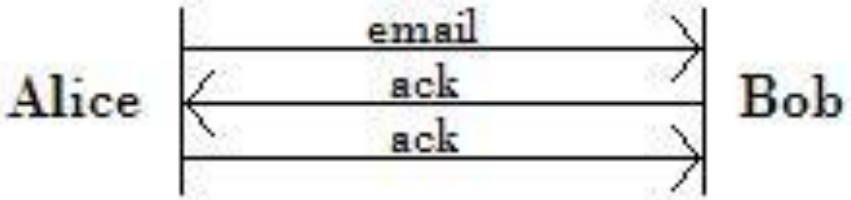
Trabalho elaborado por:
Catarina Nunes nº 2064
Diogo Sousa nº 2289
Vera Marcelino nº 2482

Conteúdo

Parte 1 - Múltiplas Faces de Consensos em Sistemas Distribuídos

Parte 2 – “Consenso: o grande mal entendido”

Teoria de Consensos



- Alice sabe que Bob recebeu a mensagem original através do primeiro acknowledgment.
- Bob tem a certeza que Alice teve conhecimento da chegada da primeira mensagem, pela segunda confirmação
- Pode haver perda de mensagens
  - Alternativa: telefonema (processos síncronos) ou envio de mensagens em duplicado

O problema de consenso

- Todos os agentes dum conjunto devem chegar a acordo sobre uma decisão, com base nos seus estados iniciais
- Tipicamente apenas duas decisões são possíveis: 0 e 1
- Se um protocolo disponibiliza duas decisões, também pode ser alargado para qualquer número de decisões
- Por exemplo, em base de dados distribuídas:
  - 1 pode representar a acção *commit*
  - 0 pode representar *abort*
  - Tem de haver pelo menos um estado para o qual 0 é o output e outro estado para o qual 1 é o output
  - Todos os agentes têm de produzir o mesmo valor de output

Protocolo de consensos

- Satisfaz as seguintes condições:
  - **Consistência** – todos os agentes concordam com o mesmo valor e todas as decisões são definitivas
  - **Validade** – o valor acordado tem de ser o input de um dos agentes
  - **Terminação** – no final de um número finito de etapas cada agente escolhe um valor (acção)

Consensos em Base de dados distribuídas

- Os protocolos de *commit* em bases de dados distribuídas usam consensos
  - todos os servidores devem concordar fazer *commit* ou *abort.* Se algum pretender abortar, então todos os servidores devem abortar
- O problema do *commit* é estritamente mais difícil de resolver do que o problema de consensos
  - Prioridade de abort de algum servidor
    - No *commit* são canceladas as acções que já tiverem sido feitas
    - Em consensos as acções apenas são efectuadas se todos os servidores estiverem de acordo

Consensos em protocolos de broadcast atómicos ordenados

- Tentam garantir que se duas mensagens, **m1** e **m2**, são enviadas, então o receptor receberá primeiro **m1** ou receberá primeiro **m2**
- Qualquer sistema que possa implementar broadcast atómico ordenado também pode obter consenso
- Quando o consenso é impossível, então broadcast atómico ordenado também se torna impossível

Consensos e tolerância a falhas

- Qualquer sistema distribuído assíncrono que implemente mecanismos de tolerância a falhas deve permitir consensos (Herlihy)
- Fischer, Lynch, e Merritt demonstraram a impossibilidade de consenso determinístico entre dois ou mais processadores num sistema distribuído assíncrono
- Desde então, o problema de consensos foi analisado sob diversos pressupostos de sincronização e falhas
- Foi demonstrado que não podem ser alcançados consensos num ambiente síncrono se mais que um terço dos processadores tiverem falhas “maliciosas”

Consensos e tolerância a falhas

- Duas categorias de falhas:
  - **Avarias fail-stop –** ocorrem quando os processadores falham por paragem. Embora este não seja um problema quando os processadores estão sincronizados, a combinação de “assincronismo” e avarias **fail-stop** podem tornar os consensos impossíveis
  - **Avarias bizantinas –** ocorrem quando os processadores falham por actuação maliciosa. Dependendo do número de avarias no sistema, os consensos podem ser impossíveis sob avarias **Bizantinas**, mesmo quando o sistema é síncrono

Consensos e tolerância a falhas

- Fischer, Lynch, e Paterson provaram que num sistema distribuído com atrasos de mensagem ilimitados, não existe qualquer protocolo que possa garantir consensos dentro de um espaço de tempo finito
- Não é possível garantir consensos em sistemas distribuídos assíncronos nos quais um processador pode falhar
- Dolev, Dwork, e Stockmeyer identificaram um conjunto de parâmetros do sistema para classificar os sistemas assíncronos

Parâmetros de classificação dos sistemas assíncronos

- Processadores: **Síncronos** ou **assíncronos**

- Atraso na comunicação: **limitado** ou **ilimitado**. O atraso é limitado se e só se todas as mensagens enviadas por um processador chegarem ao seu destino dentro de **t** passos de tempo real, para algum **t** pré-determinado.

- Mensagens: **ordenadas** ou **desordenadas**. As mensagens são ordenadas se e só se um processador $\mathbf{P_r}$ recebe uma mensagem $\mathbf{m_1}$ antes da mensagem $\mathbf{m_2}$ quando $\mathbf{P_1}$ envia $\mathbf{m_1}$ para $\mathbf{P_r}$ em tempo real $\mathbf{t_1}$, $\mathbf{P_2}$ envia $\mathbf{m_2}$ para $\mathbf{P_r}$ em tempo real $\mathbf{t_2}$, e $\mathbf{t_1 < t_2}$

- Mecanismo de transmissão: **ponto-a-ponto** ou **broadcast**. O mecanismo de transmissão é ponto-a-ponto se um processador pode enviar uma mensagem num passo atómico, no máximo para um outro processador

Classificação dos sistemas assíncronos

Processadores	Ordenação de mensagens: Desordenadas		Ordenadas		Comunicação
Assincronos	No	No	Yes	No	Ilimitada
	No	No	Yes	No	Limitada
Sincronos	Yes	Yes	Yes	Yes	
	No	No	Yes	Yes	Ilimitada
	Ponto-a-ponto	Broadcast		Ponto-a-ponto	
Transmissão					

Possibilidades de consensos

- É possível garantir consensos, no mínimo em três casos:
  1. Os processadores são síncronos e comunicação é limitada
     - cada processador pode utilizar time-out para indicar se um outro falhou

Classificação dos sistemas assíncronos

2. As mensagens são ordenadas e o mecanismo de transmissão é broadcast
    - os processadores podem ser assíncronos e alguns deles podem falhar. No entanto, têm uma primitiva de broadcast atómica ordenada

   Para obter consensos, cada processador envia o seu valor inicial para todos os outros processadores. Os processadores, em seguida, lêem as mensagens da rede e anotam o primeiro valor recebido. Como as mensagens são ordenadas, todos os processadores chegarão a acordo quanto ao primeiro valor que foi colocado na rede
3. Os processadores são síncronos e mensagens são ordenadas
    - Se um processador pode ler, processar, e escrever para a rede num passo atómico, então para obter consensos é suficiente que haja:
        - adição de atrasos limitados na comunicação
        - transmissão broadcast

Consensos em memória partilhada

- Num sistema distribuído com processadores assíncronos e memória partilhada que suporte apenas leitura e escrita, os consensos são impossíveis
- Alcançar consensos requer adicionar primitivas de sincronização para a memória partilhada.
- Herlihy revelou a existência de primitivas de sincronização que permitem aos processadores chegar a consensos na presença de cada vez mais falhas

Consensos em memória partilhada

- Dado um sistema de memória partilhada assíncrono sujeito a falhas fail-stop, Herlihy define o número de consensos de uma primitiva de sincronização
  - Por definição, uma primitiva com **n** consensos pode simular uma primitiva com **n-1** consensos; o inverso não se verifica
  - Os consensos na presença de um número arbitrário de falhas não podem ser alcançados sem o uso uma primitiva universal
- Em memória partilhada com permissões apenas de leitura e escrita, os algoritmos não determinísticos podem alcançar consensos entre dois ou mais processadores

Eventos

- Um evento, **e**, é definido como a recepção de uma mensagem **m** por um processador
  - **m** pode ser uma mensagem vazia
- Dois eventos **e** e **e'** podem ocorrer no mesmo processador ou em processadores diferentes
- Num protocolo de consensos determinístico qualquer par de eventos decisivos deve ocorrer em todos os processadores

Não é possível chegar a um consenso:

- num ambiente assíncrono de *passagem de mensagens* com falhas
- num ambiente assíncrono de *memória partilhada* com falhas

memória partilhada

sistemas que partilham memória resolvem problemas mesmo que a maioria dos processadores falhe (problemas que não podem ser resolvidos num ambiente de *passagem de mensagens* nas mesmas condições)

passagem de mensagens

caso não falhem metade ou mais processadores, o sistema de passagem de mensagens pode emular fiavelmente um ambiente de *memória partilhada*

- *O resultado da emulação também fornece uma framework mais acessível para implementar protocolos em sistemas assincronos de passagem de mensagens*

Porque nenhum algoritmo pode tolerar metade dos processadores a falhar?

Imagine-se que os processadores são particionados em dois grupos de tamanho exactamente igual:

- mensagens de um grupo para o outro são lentas enquanto que mensagens dentro do mesmo grupo seguem as taxas previsíveis
- os processadores de um grupo não conseguem aperceber-se se os processadores do outro grupo estão lentos ou falharam
- se uma adversidade tiver causado uma falha num grupo e o outro assumir que o 1º está lento, o protocolo não terminaria
- se for assumido erradamente que um processador do outro grupo falhou, então os dois grupos podiam terminar com decisões distintas, violando assim a consistência da memória partilhada

memória partilhada

Espera-se de um sistema de memória partilhada a sua habilidade de implementar:

- *shared atomic registers*: se o processador P1 termina de aceder ao registo antes do processador P2 começar a aceder ao registo e um destes acessos é de escrita, então P2 lê ou escreve uma versão tardia em relação a P1. Assumindo que cada valor escrito no registo tem um número de versão único então P2 vai escrever um numero de versão igual ou superior ao escrito por P1

Quórum de Consensos

O seguinte algoritmo mostra como manter várias réplicas de um item de dados num sistema distribuído síncrono propicio apenas a falhas *fail-stop*

- São feitas *m* copias de um objecto de dados X, {X1, X2, ..., Xm}
- Seguidamente, são escritas w>k cópias de X, onde *w* é o processo de escrita (write) e *k* são o número de falhas que podem ser toleradas. Este conjunto de *writes* é chamado de *write quorum*
- De seguida, o processo lê (read) *r* copias de X. Este conjunto de *reads* é chamado de *read quorum*
- A soma de quóruns read e write (w+r) tem que ser maior que *m* para assegurar uma intersecção entre todos os pares read e write

Foi usada esta ideia para ilustrar *como emular memória partilhada num sistema de passagem de mensagens síncrono*:

Associado a cada copia está o numero da versão. Em qualquer ponto no tempo, a copia (ou copias) com o maior numero de versão define a versão actual. Um *read* é executado da seguinte maneira:

- Faz-se um *read quorum* de X
- É seleccionada a copia com o numero de versão maior

Um *write* é executado da seguinte maneira:

- É devolvido o numero de versão mais alto, usando o procedimento *read*
- Incrementa-se o numero da versão
- É enviado o novo valor juntamente com o novo numero da versão para o *write quorum*

Os processadores que recebem o valor novo vão substituir o valor antigo na sua memória local se e só se o numero da versão do novo valor for maior que o numero da versão do valor antigo

[cont.]

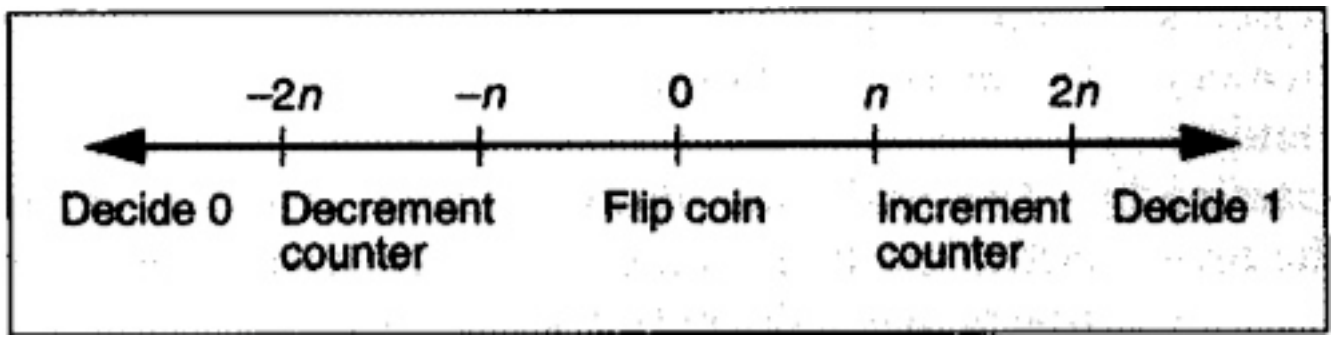
Este sistema não funcionaria num *sistema assíncrono*. A principal dificuldade é a impossibilidade de garantir que as copias serão lidas na ordem correcta

Time step	Processor 1 (W_1)	Processor 2 (R_1)	Processor 3 (R_2)
1	Write X_1		
2		Read X_1	
3		Read X_3	
4			Read X_2
5	Write X_2		Read X_3

- Existem três réplicas - X1, X2, X3 - de um objecto, X
- Um *writer*, W1, pode ter sucesso na escrita de X1 antes de ficar lento
- Um *reader* subsequente, R1, pode ler um *quorum* que contenha X1 e X3 conseguindo assim a nova versão de X escrita por W1
- Mais tarde, outro reader, R2, pode ler um *quorum* que consista em X2 e X3. Este *quorum* não contém a nova versão de X. R2 recebe então uma versão mais recente que R1, violando as condições necessárias para registos atómicos

algoritmo do consenso aleatório

- Assuma-se que os processadores podem lançar uma moeda ao ar e cada um adicionar ou subtrair 1 dum contador global num passo atómico
- Os outros processadores não têm influência no lançamento da moeda (ou na ordem em que os resultados são adicionados ao contador global); são acontecimentos independentes
- Uma vez atingida uma das fronteiras, os restantes processadores irão eventualmente tomar a mesma decisão



acordo Bizantino

Diversas divisões do exercito Bizantino estão destacadas fora de um acampamento inimigo. Cada divisão é liderada por um general que está a tentar decidir se ataca ou não o acampamento inimigo. Contudo, alguns dos generais são traidores e tentarão impedir os generais honestos de chegar a um acordo. Uma falha bizantina é aquela em que um processador passa a traidor e actua maliciosamente

- O problema de atingir consenso num sistema distribuído propicio a falhas bizantinas é conhecido como *acordo Bizantino*
- Falhas bizantinas foram originalmente usadas para modelar falhas de hardware em sensores de aviação mas também servem para modelar falhas de software
- Caso o software falhe não temos ideia do que poderá fazer. Visto que poderá fazer qualquer coisa, a única assunção geral a fazer é que fará a pior coisa possível

acordo Bizantino - evitando traidores

- Devido ás capacidades inerentes de *broadcast* da memória partilhada, as falhas bizantinas não constituem um problema neste ambiente
- Quando a autenticação não está disponível, o acordo bizantino é possível se e só se existirem apenas 3k+1 processadores quando *k* processadores puderem falhar
  - Logo, quando menos de um terço dos processadores numa rede são traidores, o acordo deterministico sem autenticação é possível

algoritmo de autenticação

- Assume-se a existência de um único coordenador, C
  - Quando o coordenador é honesto, todos os agentes honestos irão escrever no *output* o *input* inicial do coordenador
  - Quando o coordenador é desonesto, todos os agentes honestos escreverão 0 no *output*
- O algoritmo prossegue em t+1 fases. Cada mensagem enviada por um processador transporta a assinatura de todos os processadores que viram e transmitiram a mensagem
- Na fase *i*, deverão existir *i* assinaturas (juntamente com a do coordenador) e nenhum duplicado. Isso torna a mensagem legitima

O algoritmo cumpre:

- terminação: termina depois de $t+1$ fases

- validação: se todos os processadores funcionarem correctamente e todos tiverem o mesmo *input*, então chegarão a acordo sobre o *input* inicial

- consistência: todos os processadores que funcionem correctamente verão os mesmos valores que todos os outros processadores que funcionem correctamente e sendo assim chegarão à mesma decisão

conclusão

- para um grupo de processadores chegar a uma decisão comum têm de resolver o problema do consenso
- quanto mais fiáveis os processadores, menos fiável pode ser a rede
- num sistema distribuído síncrono com entrega de mensagens fiável e processadores sujeitos a falhas bizantinas, o consenso é possível desde que menos de um terço dos processadores falhe
- Num sistema distribuído assíncrono com entrega de mensagens fiável e processadores sujeitos a falhas por paragem, o consenso não é possível mesmo que só um processador possa falhar

conclusão [cont.]

- num sistema distribuído síncrono em que mensagens podem ser perdidas, o consenso não é possível mesmo que nenhum dos processadores falhe
- memória partilhada aumenta a fiabilidade dos meios de comunicação. É equivalente a adicionar capacidade de *broadcast* a uma rede. Evita muitos dos problemas criados por falhas bizantinas, mas não os problemas criados por assincronia
- técnicas como aleatoriedade e autenticação oferecem meios para superar muitas impossibilidades e regularmente produzem algoritmos eficientes

2ª Parte:

“Consenso:
o grande mal entendido”

Introdução:

- Esta parte do trabalho ajuda a entender alguns mal-entendidos sobre o problema do consenso.

- São referidos seis frequentes mal entendidos.

Grande mal entendido 1: o consenso é apenas para os teóricos

- O consenso pode ser visto como uma forma geral de acordo nos sistemas distribuídos.

- O problema é constituído por um conjunto de processos *{p1, p2, p3, …, pn}*: cada processo *pi* tem um valor inicial vi, e os processos correctos têm de decidir um valor comum v que é o valor inicial de um dos processos.

- Este problema atraiu muitos teóricos provando no final que o consenso não tem resolução possível num sistema assíncrono se um processo único puder falhar (chamado de resultado impossível FLP).

- Este problema do consenso foi ignorado pelos implementadores dos sistemas, pois estes consideram-no irrelevante para os sistemas reais. O que se argumentava para este facto é que:

  – ***Os sistemas reais têm de resolver problemas de acordo práticos tais como o atomic broadcast, atomic commitment, eleição do líder, o ingresso no grupo, etc. Então porque preocupar com o problema do consenso e os resultados impossíveis do FLP?***

- Foi demonstrado que o problema do broadcast atómico e o problema do consenso são equivalentes.

- Isto quer dizer que qualquer solução para o problema do broadcast atómico pode ser usado para resolver o problema do consenso e vice-versa.

- Embora o broadcast atómico seja equivalente ao consenso, o mesmo resultado não é mantido em todos os problemas de acordo.

Grande mal entendido 2: os tempos de pausa são suficientes

- A impossibilidade de resultados FLP aplica-se em sistemas assíncronos, os quais não têm tempos.

- Por esta razão, a impossibilidade dos resultados FLP é muitas vezes considerado como um resultado óbvio: sem tempo, não há maneira de detectar falhas nos processos:
  - ***O consenso pode ser facilmente resolvido adicionando timeouts no sistema assíncrono.***

- Esta afirmação não está totalmente correcta.
- A ausência do tempo nos sistemas assíncronos não previne os processos de suspeitar de falhas de outros processos.
- Assim um mecanismo de time-out pode ser facilmente implementado baseado num tempo lógico. Mas adicionar time-outs aos sistemas assíncronos não é suficiente para ultrapassar a impossibilidade dos resultados FLP.
- O conteúdo dos resultados FLP é que torna impossível de distinguir falhas de processos, daqueles que estão lentos ou estão ligados a links lentos.

Grande mal entendido 3: não há vida depois do FLP

- A dificuldade da resolução do consenso pode levar à seguinte observação:
  - ***O consenso é algumas vezes resolvido, e outras vezes não o é: portanto porquê preocuparmo-nos com o último caso?***

- Com efeito, a impossibilidade do resultado FLP indica que, num sistema assíncrono não existe algoritmo que resolva o consenso em todas as execuções possíveis.

- Num sistema síncrono o algoritmo para resolver um problema para ser executado correctamente não pode ter mais que um dado número de falhas.

- Os resultados FLP levaram à definição de vários modelos:
  - modelo síncrono parcial
  - modelo assíncrono regulado
  - modelo assíncrono com detector de falhas
- Os dois últimos são menos restritivos que os modelos síncronos e permite a caracterização das execuções sob um dado algoritmo para resolver o consenso.

Grande mal entendido 4: o modelo do detector de falhas é irrealista

- O modelo de detecção (no contexto dos sistemas assíncronos) é muitas vezes mal interpretado e alguns críticos expressaram:

1. ***O detector de falhas S não é implementável num sistema assíncrono.***
   - O detector de falhas tem de ser visto como uma especificação do mecanismo de detecção de falhas.
   - S requer a existência de um processo correcto ***p*** e um tempo ***t*** em que depois desse tempo ***p*** não pode ser detectado por mais nenhum processo.
2. ***O modelo não engloba recuperação de processos: nos sistemas de processos reais faz recuperação.***
   - O modelo de falha/não recuperação pode ser alargado para incluir processo de recuperação.

Grande mal entendido 5: o free-time significa incapacidade

- Os modelos parciais síncronos e os modelos assíncronos regulados são ambos baseados no tempo.

- O modelo assíncrono com detector de falhas é um modelo de free-time (o tempo é escondido no detector de falhas).

- É frequentemente argumentado que:
  - ***Resolvendo o consenso num modelo de free-time leva a uma solução complexa e ineficaz.***

- O algoritmo é complexo porque pode tolerar um número infinito de falhas incorrectas suspeitas.
- Definindo uma medida para a eficiência do algoritmo distribuído não é fácil.
- O número de mensagens enviadas por um algoritmo ***A*** é uma medida possível de eficiência deste.
- O número de passos de comunicação (ou grau de latência), é provavelmente a medida mais adequada.
- Mede-se aqui a eficiência de um algoritmo de consenso quanto ao número de passos na comunicação em execuções boas.

Resolvendo o consenso com três passos de comunicação.

- O algoritmo de consenso Chandra-Toueg é baseado em ‘S’ (detector de falhas) e requer 3 passos de comunicação numa execução boa:

1. Inicialmente, o processo *p1* faz um broadcast ao seu valor inicial ***v1*** (***p1*** propõe *v1* como o valor decidido);
2. Cada processo aceita a proposta de ***p1*** enviando um ack de volta para ***p1***;
3. Uma vez ***p1*** ter recebido a maioria das mensagens de ack, ele faz um broadcast da mensagem de decisão.

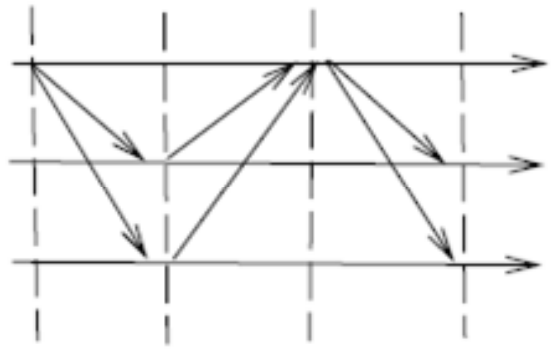


Figura 1 – Algoritmo do consenso Chandra-Toueg baseado em *S* .

Resolvendo o consenso com dois passos de comunicação.

- O algoritmo Chandra-Toueg pode ser melhorado.

- Os primeiros algoritmos do consenso, também baseados no detector de falhas ***S***, resolvem o consenso em 2 passos de comunicação numa execução boa:
  1. Inicialmente o processo ***p1*** faz um broadcast do seu valor inicial ***v1***;
  2. Todos os processos aceitam a proposta de ***p1*** enviando ***v1*** para todos. Um processo decide depois de ter recebido ***v1*** a partir de uma maioria de processos.

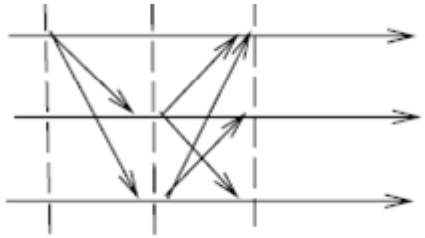


Figura 2 – Algoritmo do consenso baseado em *S* .

É possível resolver o consenso com apenas um passo?

- Este algoritmo deve poder decidir sobre o valor inicial de um dos processos, por exemplo, ***p1***. Contudo, tal algoritmo não pode estar correcto.
- Um processo, depois de ter recebido ***v1***, não pode ter a certeza que os outros processos também receberam ***v1*** (e decidiram sobre ***v1***).
- Somos tentados a resolver o problema num modelo baseado em tempos:
  - Sempre que algum processo ***pi*** receba o valor inicial ***v1*** *do processo* ***p1***, o processo ***pi*** espera durante algum tempo Δ antes de decidir (onde Δ é a duração necessária para detectar que a falha da rede que pode ter ocorrido). Depois de Δ, se ***pi*** não foi notificado da falha da rede, então ***pi*** decide ***v1***.

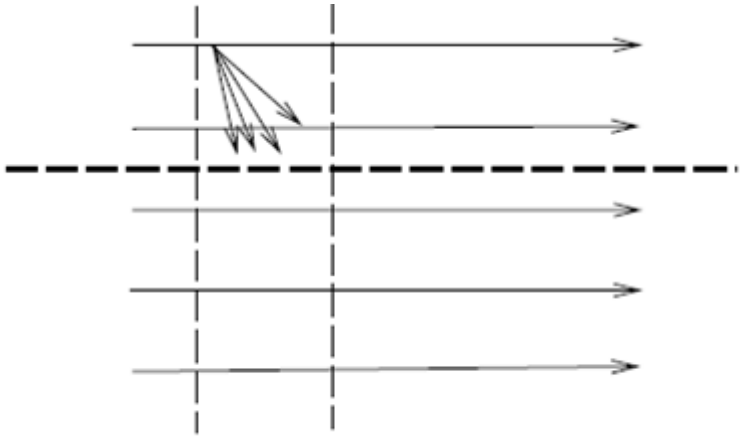


Figura 3 – Algoritmo com um único passo de comunicação: a propriedade do acordo do consenso é violado.

- Isto introduz um custo oculto no algoritmo do consenso, o que tem de ser adicionado ao custo de um passo de comunicação do algoritmo.

- Uma análise cuidadosa é necessária para comparar o custo de **(i)** dois passos de comunicação vs **(ii)** um passo de comunicação mais Δ.

- Contudo o resultado mostrando que **(ii)** é menos custoso, desde que o atraso para detectar a falha da comunicação é pelo menos o atraso necessário para transmitir uma mensagem.

Grande mal entendido 6: algoritmos assíncronos não podem ser usados por aplicações críticas de tempo

- ***Algoritmos de broadcast atómico assíncrono não podem ser usados em aplicações criticas, porque elas não garantem um limite no tempo que leva a fazer o broadcast de uma mensagem na presença de uma falha.***

- Este argumento está incorrecto.

- Se um algoritmo ***A***, baseado num modelo ***M***, garante um limite no tempo que leva a resolver um broadcast atómico (ou algum problema de acordo), isto é devido ao modelo ***M*** restringir o número de falhas e não é devido ao algoritmo mais sofisticado ***A***.

- Acredita-se que a abordagem mais correcta é aquela em que se constroem algoritmos de acordo para aplicações criticas de tempo, em duas fases:

  - **Fase 1**: desenvolvimento de um algoritmo **A** num modelo de tempo livre (modelo assíncrono com detector de falhas) e provar que está correcto (segurança);

  - **Fase 2**: “Imergir” o algoritmo num sistema real, exemplo, um sistema com relógios de tempo real. O tempo real garante que pode, por exemplo, ser obtido como o resultado da imersão do algoritmo ***A***, aceitando para o cálculo as propriedades específicas do sistema real.

- A fase 1 é chamada de fase de “design” do algoritmo e a fase 2 é chamada de fase de implementação.

Conclusão

- Durante as discussões sobre este tema, têm surgido frequentes equívocos sobre como resolver o problema do consenso;

- Foi defendida uma abordagem em que os problemas de acordo são resolvidos com um modelo mais geral (assíncrono com detector de falhas);

- Esta abordagem não leva à perda de eficiência e deve ser adequada mesmo no conceito de aplicações críticas de tempo.