Computação Distribuída I

Prof. Lau Cheuk Lung
E-mail: lau.lung@inf.ufsc.br
Programa de Pós-Graduação em Ciência da Computação - PPGCC
Departamento de Informática e Estatística – INE
Universidade Federal de Santa Catarina - UFSC

Tópicos da disciplina

Módulo I

- Segurança de funcionamento
- Modelo de Sistema
- Tolerância a faltas de software
- Comunicação de grupo: difusão confiável e difusão atômica
- Sincronização de relógio

Módulo II

- Algoritmos de eleição e exclusão mútua
- Detecção de *deadlocks* (Impasses)
- Algoritmos de acordo (consenso distribuído)
- Transações distribuídas
- Memória compartilhada e distribuída (DSM)

Roteiro da apresentação

1. Segurança de funcionamento
2. Taxonomia
3. Classificação de faltas e semântica de falhas
4. Tolerância a Faltas de hardware e software

Prof. Lau Cheuk Lung – INE/UFSC
www.inf.ufsc.br/~lau.lung

Preâmbulo

Como surgiu a Tolerância a Falta, Engenharia de Software e Segurança de Sistemas

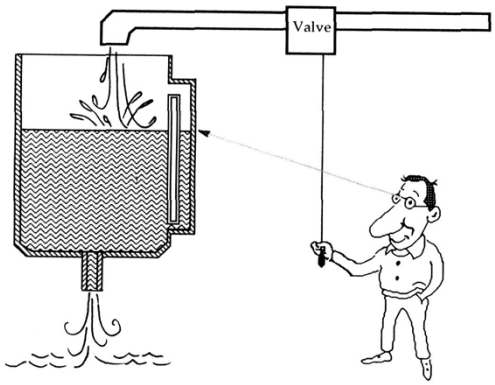
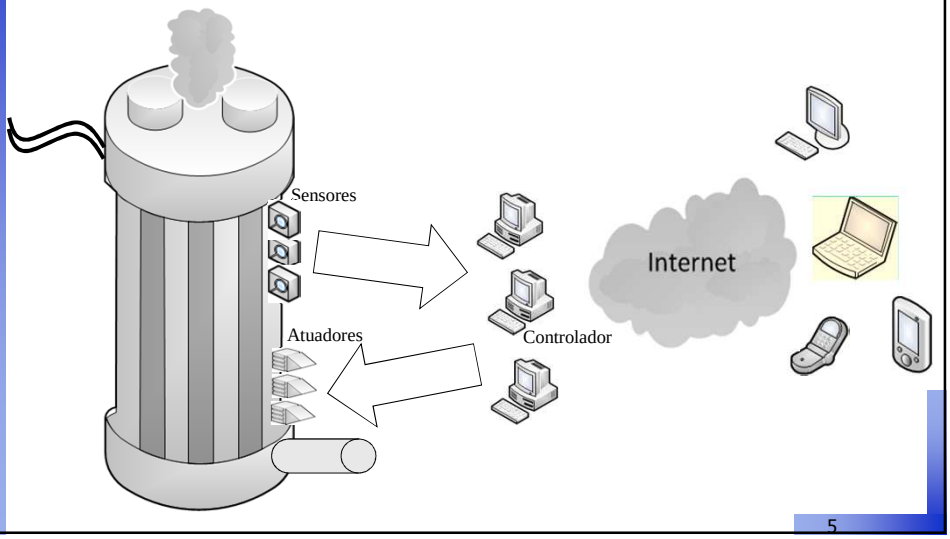


FIGURE 1.1. Level Control System. A sight tube and operator's eye form a sensor: a device which converts information into electrical signal.

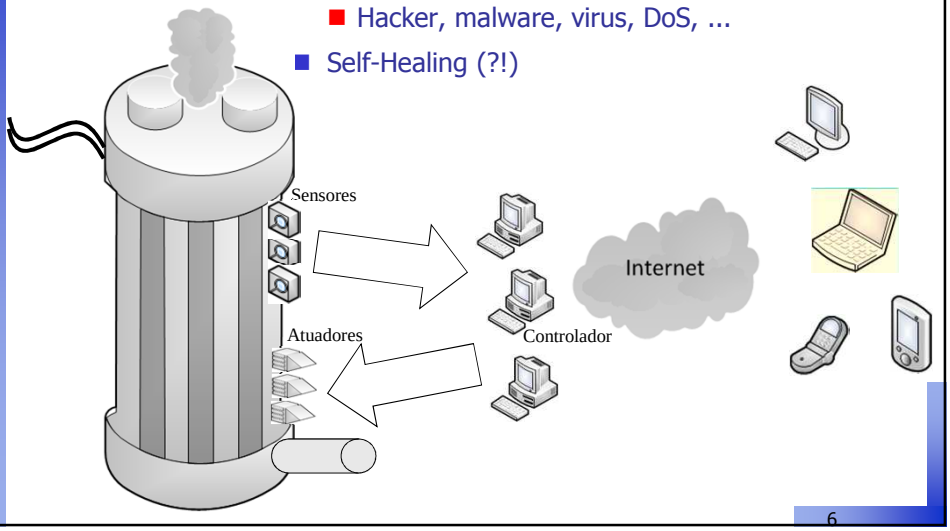
Preâmbulo

Como surgiu a Tolerância a Falta, Engenharia de Software e Segurança de Sistemas



Preâmbulo

- Tolerância a Faltas + Segurança = Tolerância a Intrusões -> FRS e Atomicidade (acordo)
  - Hacker, malware, virus, DoS, ...
- Self-Healing (?!)



Segurança de Funcionamento

- Sistemas computacionais são artefatos frágeis, muitas vezes não fazem o que é esperado ou, quando fazem, em um momento inconveniente;
  - Mas qual a causa disso ?
    - Os computadores são sistemas complexos, feitos de diferentes hardware e componentes de software;
      - Esses componentes interagem com os outros muitas vezes de forma imprevista pelo projetista do sistemas;
    - Velha da lei da engenharia - Lei de Murphy:
      - Se a possibilidade de ocorrência de um perigo é negligenciada, ela tende a ocorrer da pior maneira possível e no pior momento possível.

Segurança de Funcionamento

- Conceito:
  - *Segurança de funcionamento* é a qualidade do serviço fornecido de modo que os seus usuários possam depositar no mesmo uma confiança justificada.
  - Um sistema é confiável (*dependable*) se tem uma alta probabilidade de proceder sua função de acordo com sua especificação;
    - A completa especificação não deve ser limitada para o que o sistema faz, mas deve também especificar as condições ambientais requeridas para o sistema fornecer o serviço requerido.

Segurança de Funcionamento

- Quando o procedimento do sistema viola sua especificação dizemos que ele falhou, mas o que leva um sistema à falha ?
  - São causas externas ou internas, chamada de falta.
    - A falta pode permanecer adormecida (inativa) por um momento, até ser ativada.
    - Exemplo: defeito em um registro de um sistema de arquivo que pode permanecer despercebido até ser ativado. O registro corrompido é um erro no estado do sistema que o levará a falha.

Taxonomia de Segurança de Funcionamento

- Imperfeições: faltas, erros, falhas
  - Falta é a causa, no sentido fenomelógico, de um erro;
  - Um erro é a parte do estado do sistema (estado errôneo) que pode conduzir a uma falha no sistema;
  - A falha de um sistema ocorre quando o serviço fornecido desvia do serviço especificado. A falha é o efeito observado externamente do erro.
    - Falha é sentida e/ou avaliada.
    - Erro é detectado/recuperado.
    - Falta é evitada ou tolerada.

Um sistema é tolerante a falhas ou a faltas ?

Taxonomia de Segurança de Funcionamento

- Falhas similares podem ser derivadas de erros completamente diferentes, exemplo:
  - Caractere estranho na tela:
    - Rotina falha do sistema operacional;
    - Defeito da placa de vídeo.
- Também, erros podem ser causados por diferentes faltas, exemplo:
  - O erro do disco pode ser devido à falta física na superfície do disco ou pode ser devido à defeito de fabricação do cabeçote.

Taxonomia de Segurança de Funcionamento

- A falha de um sistema é provocada por uma falta.
- Mas, qual a origem das faltas ?
  - Física: gerados por causas físicas (hardware);
    - Ex: desgaste e fadiga dos materiais.
  - Projeto: quando introduzidos na fase de projeto;
    - Ex: programador incompetente.
  - Interação: quando ocorrido nas interfaces entre os componentes do sistema, ou na interface com o mundo exterior.

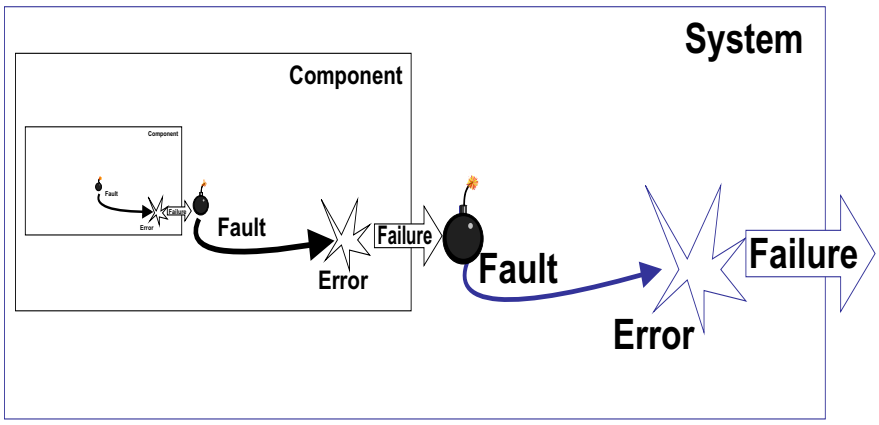
Faltas de projeto e algumas faltas de interação são causadas por falha humana.

Taxonomia de Segurança de Funcionamento

- Faltas também podem ser classificadas quanto a:
  - Natureza: acidental ou intencional (maliciosa ou não);
  - Fase de criação na vida do sistema: desenvolvimento ou operação;
  - Local: interno ou externo;
  - Persistência:
    - **Falha transiente**: ocorre apenas uma vez; se a operação (ou requisição) é repetida, a falha desaparece. Exemplo: falha numa transmissão usando micro-ondas porque algum objeto obstrui temporariamente.
    - **Falha intermitente**: ocorre de maneira aleatória e imprevisível. Exemplo: um conector com mau contato.
    - **Falha permanente**: ocorre sempre até que o componente seja substituído. Exemplo: um chip queimado, erros em software.

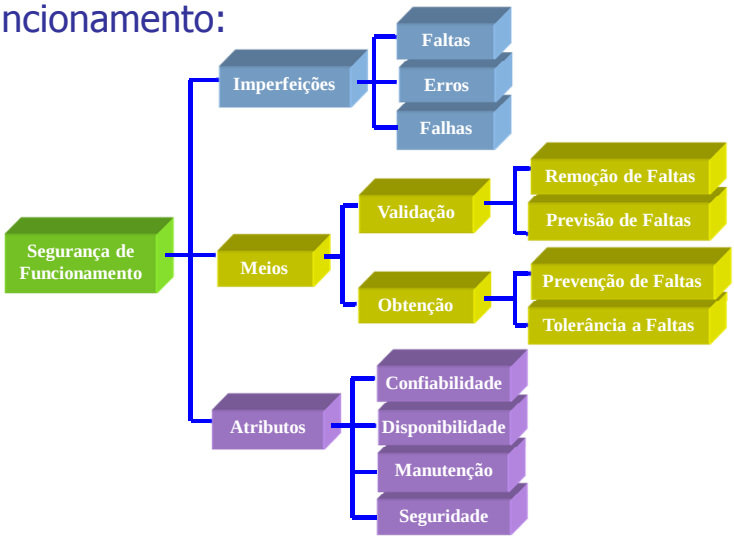
Taxonomia de Segurança de Funcionamento

- Sistema: faltas, erros, falhas



Taxonomia de Segurança de Funcionamento

- Principais conceitos relacionados com segurança de funcionamento:



Taxonomia de Segurança de Funcionamento

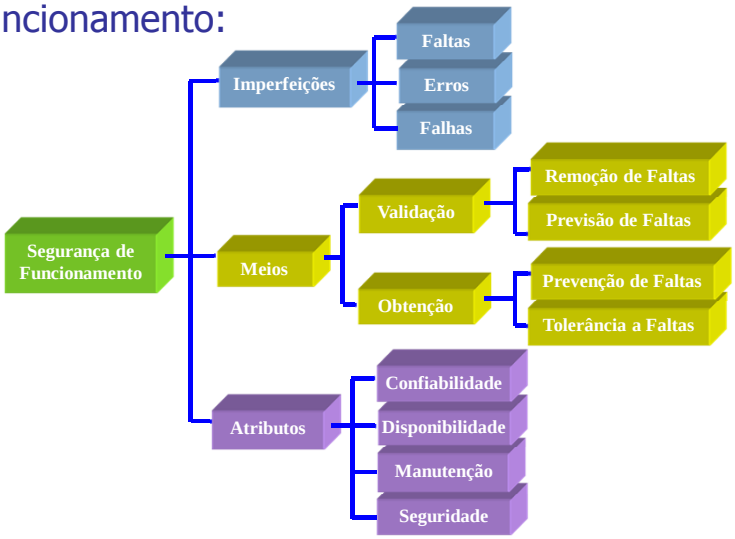
- Métodos de Validação da *Segurança de Funcionamento*
  - **Remoção de faltas** (verificação): métodos que minimizam a presença de faltas. Consiste em detectar as faltas e remover eles antes que causem um erro, exemplo:
    - Removendo bugs de software;
    - Detectando defeito de hardware;
  - **Previsão de faltas** (avaliação): métodos de avaliação que estimam a presença de faltas e suas conseqüências.

Taxonomia de Segurança de Funcionamento

- Métodos de Obtenção da *Segurança de Funcionamento*:
  - **Prevenção de faltas**: consiste de prevenir (por construção) as causas dos erros, eliminando as condições para uma provável ocorrência de falta durante operação do sistema, exemplo:
    - Usando componentes de alta qualidade;
    - Usando componentes com redundância interna.
  - **Tolerância a faltas**: fornecimento do serviço de acordo com as especificações, mesmo em presença de uma ou mais faltas ativas, pelo uso de redundâncias.

Taxonomia de Segurança de Funcionamento

- Principais conceitos relacionados com segurança de funcionamento:

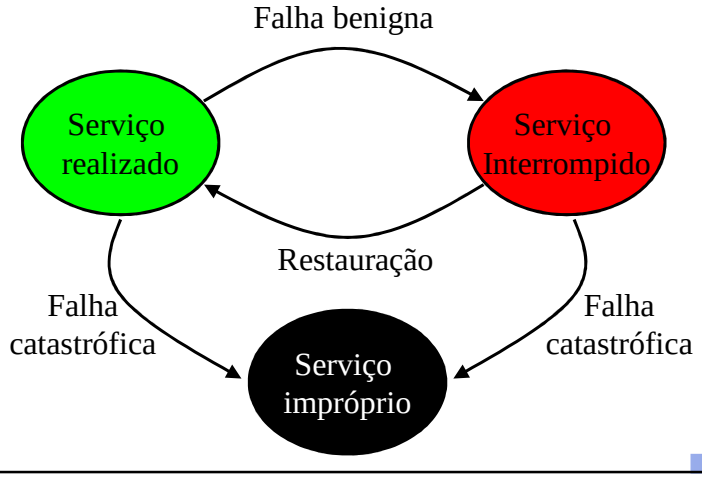


Taxonomia de Segurança de Funcionamento

- Medidas
  - A vida de um sistema é percebida pelos seus usuários como a alternância entre dois estados do serviço fornecido com respeito as especificações:
    - *Serviço interrompido* (impróprio): serviço fornecido é diferente do especificado;
    - *Serviço realizado* (próprio): serviço é fornecido como foi especificado;

Taxonomia de Segurança de Funcionamento

- Os eventos que provocam a transição entre estes estados são a *falha* e a *restauração*.
  - Falhas *catastróficas* e *benignas*



Taxonomia de Segurança de Funcionamento

- Como se mede a *Segurança de Funcionamento* de um sistema ?
  - Através da quantificação da alternância entre estes estados se mede (atributos de SF):
    - **Confiabilidade** (*reliability*): é a medida do fornecimento continuo do serviço correto a partir de um instante inicial (tempo para falha);
    - **Disponibilidade** (*Availability*): é a medida da liberação de serviço correto considerando a alternância de serviço correto e serviço incorreto;
    - **Manutenção** (*Maintainabiliy*): medida da interrupção continua do serviço, a partir do instante de falha, até a sua restauração (tempo de restauração);
    - **Seguridade** (*safety*): o grau para o qual um sistema quando sob falha ocorre de uma maneira não catastrófica.

Taxonomia de Segurança de Funcionamento

- Confiabilidade
  - Pode ser equacionada como a probabilidade que o sistema não falha durante o período de missão do sistema (ex: um vôo aéreo);
  - Para sistemas de missão contínua (ex: servidor Web), confiabilidade pode também ser expresso pelo *tempo médio para a falha* (MTTF) ou pelo *tempo médio entre falhas* (MTBF);
  - Confiabilidade pode ser expresso como uma probabilidade de taxa de falha, exemplo:
    - 10^{-9} falhas por hora.

Taxonomia de Segurança de Funcionamento

- Disponibilidade
  - É a probabilidade do sistema estar operacional em um dado tempo, quando ele alterna para um estado falho.

Você como administrador de sistema, quantas horas (ou dias) sua rede fica interrompida durante um ano ?

Taxonomia de Segurança de Funcionamento

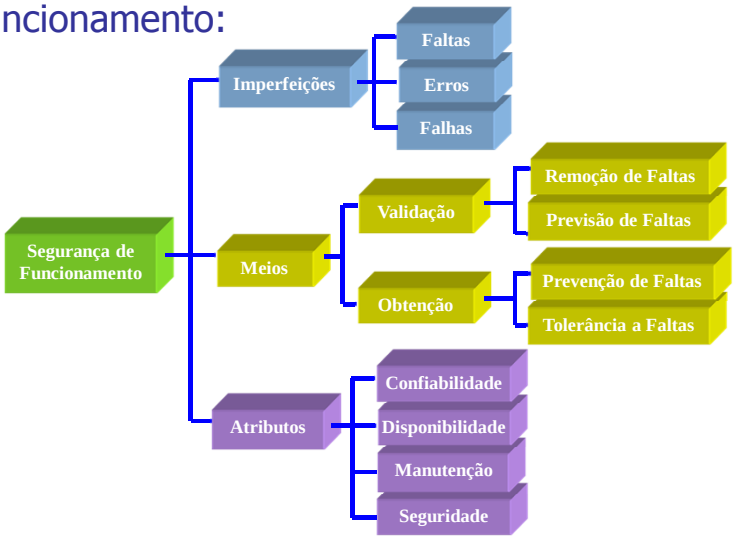
- Manutenção
  - É o atributo que define o tempo necessário para o sistema recuperar de uma falha:
    - Dado confiabilidade MTBF e manutenção MTTR (tempo médio de reparação) de um sistema, disponibilidade pode ser expresso como:

Disponibilidade = MTBF/(MTBF+MTTR)

Disponibilidade	Tempo ocioso/ano	Componente exemplo
90%	> 1 mês	PC sem manutenção
99%	≈ 4 dias	PC com manutenção
99.9%	≈ 9 horas	Cluster
99.99%	≈ 1 hora	Multicomputador
99.999%	≈ 5 minutos	Sistema embutido (tecnologia PC)
99.9999%	≈ 30 segundos	Sistema embutido (hardware especial)

Taxonomia de Segurança de Funcionamento

- Principais conceitos relacionados com segurança de funcionamento:



Classificação de faltas e semântica de falhas

- A classificação de faltas e o conhecimento de sua natureza são de grande importância no projeto de sistemas capazes de tolerar faltas.
  - Várias classificações: *origem* (humanas/projeto); *persistência* (transitórias/intermitentes/permanentes);
  - Faltas podem ser classificadas segundo a ótica de seus *efeitos*. Fortemente vinculado com as *semânticas de falhas*.
- Serviço correto:
  - Considerando que os itens de serviço s_i (i=1,2,3,...n) são caracterizados pelos pares $<vs_i, ts_i>$ com vs_i sendo o valor de serviço s_i e ts_i é o tempo ou instante de observação do mesmo pelo usuário.
  - Definição: um serviço s_i é definido como correto se e somente se:

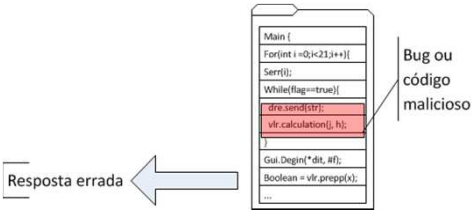
$$(vsi \in VSi) \wedge (tsi \in TSi)$$

onde:

- VSi é o conjunto especificado de valores para o item si;
- TSi é o conjunto especificado de tempos para o item si;

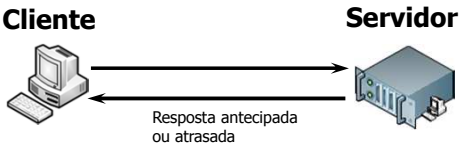
Classificação de faltas

- Classificação de faltas segundo o efeito: basicamente as faltas podem provocar erros no domínio de valores (VSi) e no de tempo (TSi).
  - Faltas no domínio de valores:
    - erro de valor fora do código: $(vsi \notin VSi) \wedge (vsi \notin VS)$
    - erro de valor no código: $(vsi \notin VSi) \wedge (vsi \in VS)$ (faltas maliciosas)
      - onde $VS = VS1 \cup VS2 \cup ... \cup VSi \cup ....$



Classificação de faltas

- Classificação de faltas segundo o efeito: basicamente as faltas podem provocar erros no domínio de valores (VSi) e no de tempo (TSi).
  - Faltas no domínio de tempo:
    - Faltas de temporização "Timing Faults":
      - *erros por antecipação* : tsi < Tsi
      - *erros por atraso* : tsi > Tsi

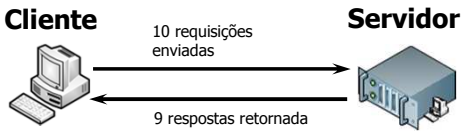


Classificação de faltas

- Faltas de omissão: caso particular de faltas de temporização é aquele em que um item de serviço nunca tem o seu valor liberado por omissão.

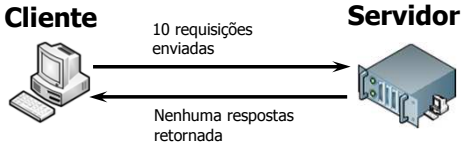
  tsi $\rightarrow \infty$ (erro de omissão)

  - Os serviços podem ter semântica de omissão incorporadas: um serviço de grau de omissão k permite no seu comportamento a omissão consecutiva de k itens de serviço. Atingido este valor é assumido um "crash" (parada) no serviço.



Classificação de faltas

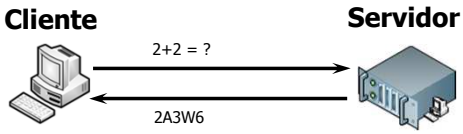
- Faltas de parada ("crash"): caso particular de faltas de omissão: se após a primeira omissão o servidor deixa de responder sistematicamente $\Rightarrow$ *crash* no serviço.



Classificação de faltas

- Faltas arbitrárias: envolvem erros nos domínios de valores e de tempo:

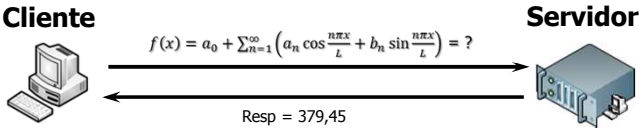
$$(vsi \notin VSi) \vee (tsi \notin TSi)$$



Classificação de faltas

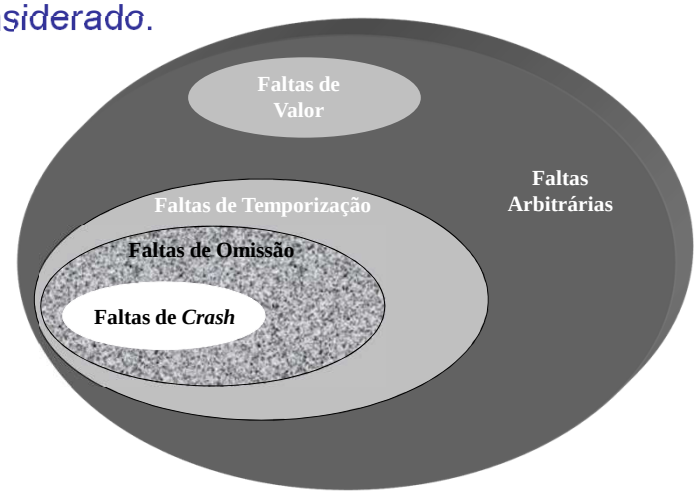
- As faltas maliciosas ou intencionais geram itens de serviços si com valores fora do conjunto VSi em instantes aleatórios. Erros determinam valores vsi fora de VSi, dados mantém alguma propriedade a confundí-los com valores pertencendo a VS.

$$(vsi \notin VSi) \wedge (vsi \in VS) \wedge (tsi \in TSi)$$



Classificação de faltas

- Quanto mais restritiva o modelo de faltas assumido menor a complexidade do algoritmo distribuído considerado.



Tolerância a faltas

- Objetivo: Mascarar falhas
- Causas de falhas:
  - Em hardware: razões físicas;
  - Em software: falhas de projeto.
- Confiabilidade:
  - Hardware: Muito confiável;
  - Software: Pouco confiável.

Tolerância a faltas

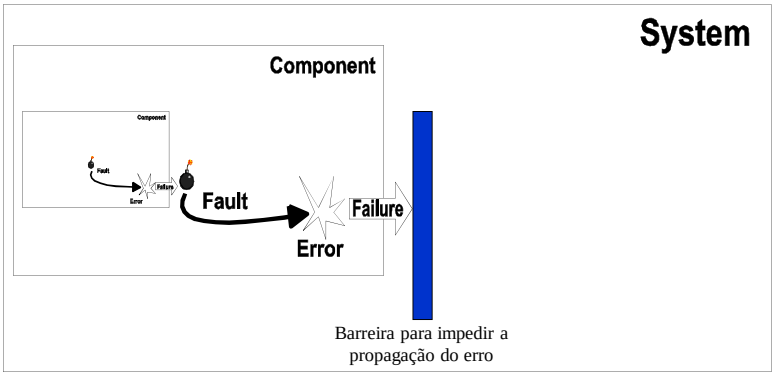
- Sistema Tolerante a Faltas é aquele que pelo uso de redundâncias mantém o serviço correto mesmo em presença de componentes faltosos;
- Redundâncias são partes do sistema que não seriam necessárias para o funcionamento do sistema se não ocorressem faltas no sistema:
  - Redundância de hardware;
  - Redundância de software;
  - Redundância de tempo.
- Nos sistemas atuais ocorrem os três tipos de redundâncias.

Tolerância a faltas de hardware

- Fases da Tolerância a Faltas: Detecção de erros, Confinamento de erros, Recuperação de erros e Tratamento de faltas.
  - **Detecção de erros**: é a fase onde a presença de elementos faltosos é deduzida pela constatação de um erro. São meios que possibilitam a prevenção de uma falha.
    - Testes completos, testes de aceitação, testes independentes, internos, etc.
  - **Confinamento de erros**: técnicas e mecanismos que delimitam os eventuais danos a um sistema devido a propagação de erros. Erro se propaga pela interação de componentes de um sistema.
    - Barreiras incorporadas ao sistema em tempo de projeto.

Taxonomia de Segurança de Funcionamento

- Sistema: faltas, erros, falhas



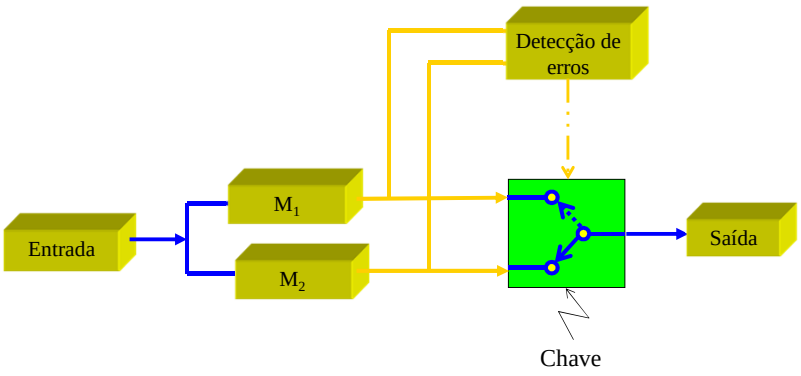
  - Barreiras incorporadas ao sistema em tempo de projeto. Ex: sobre-tensão da rede elétrica -> filtro
  - Honeypot -> confinar as ações de um hacker.

Tolerância a faltas de hardware

- **Recuperação de erros**: restauração da seqüência de estados corretos. Dois mecanismos que tratam erros:
  - Técnicas de processamento de erro:
    - Recuperação em retrocesso (*Backward recovery*): coloca o sistema em um estado correto anterior. A partir deste estado recomeça o processamento;
    - Recuperação em avanço (*Forward recovery*): usando o estado errôneo, coloca o sistema em um estado correto a frente. É importante em aplicações tempo-real (tal como um sistema de controle embarcado de um aeroplano).
  - Mascaramento de erros:
    - Uso de réplicas ativas para que o componente faltoso seja mascarado em seu comportamento.

Tolerância a faltas de hardware

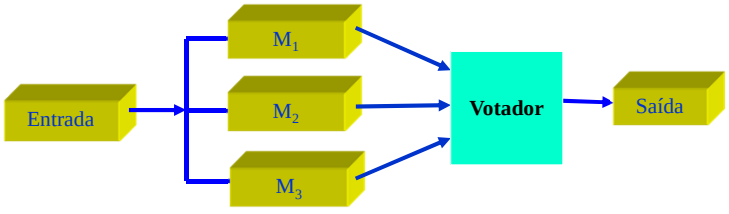
- **Tratamento de Faltas**: configuração dinâmica para tirar elementos faltosos.



Replicação Passiva (configuração dinâmica)

Tolerância a faltas de hardware

- **Tratamento de Faltas**



Replicação Ativa (configuração estática)

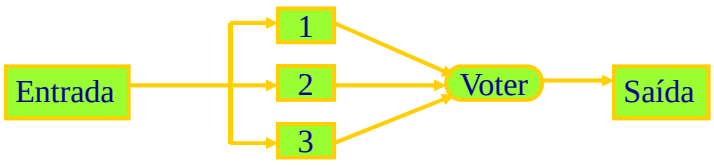
Ignora o elemento faltoso

Tolerância a faltas de software

- Hardware: replicação de componentes
- Software: diversidade de projetos
  - Programação N-versões (N-version Programming)
  - Abordagem Bloco de Recuperação (Block Recovery Approach)

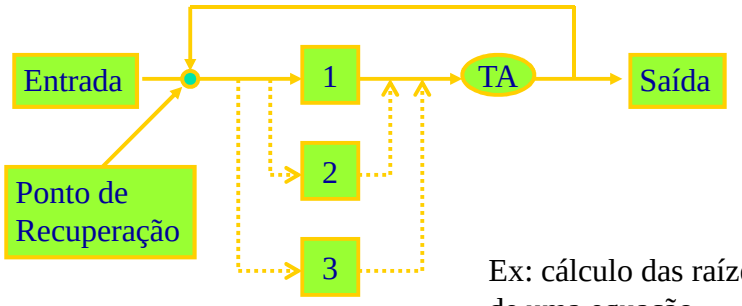
Tolerância a faltas de software

- Programação N-versões (N-version Programming): diferentes implementações de um mesmo algoritmo
  - Usando diferentes linguagens de programação para cada versão;
  - Usando diferentes compiladores e ambientes de suporte;
  - Usando diferentes algoritmos, se possível;
  - Usando diferentes times de desenvolvimento.



Tolerância a faltas de software

- Bloco de recuperação: o programa é dividido em blocos com pontos de recuperação em caso de falha detectado pelo teste de aceitação (TA).



Ex: cálculo das raízes de uma equação quadrática.

Roteiro da apresentação

3. Modelo de Sistema
4. Tolerância a Faltas em sistemas distribuídos

Prof. Lau Cheuk Lung – INE/UFSC
www.inf.ufsc.br/~lau.lung

Modelo de Sistema Distribuído

- Modelo Interação: trata da comunicação e da coordenação (sincronização e ordenação das atividades) entre os processos de um sistema distribuído.
- Modelo Falha: define e classifica as falhas. Fornece as bases para a análise de seus efeitos em potencial e para projetar sistemas capazes de tolerar certos tipos de falhas e continuar funcionando corretamente;
- Modelo Segurança: define e classifica as formas de ataque, dando base para a análise das possíveis ameaças a um sistema e assim guiar o desenvolvimento de sistemas capazes de resisti-los;

Modelo de Sistema Distribuído

- Um sistema distribuído consiste de um conjunto de processadores conectados por uma rede de comunicação;
- Cada processo executa em um processador diferente;
- Um programa distribuído é composto de um conjunto de *n* processos assíncronos *p1*, *p2*, ... *Pi*, ..., *pn* que se comunicam por passagem de mensagem através da rede de comunicação;
- O atraso de comunicação é finito, mas imprevisível;
- O processadores não compartilham uma memória comum;

Modelo de Sistema Distribuído

- Normalmente, não há um relógio físico global no sistema para que os processos possam consultar instantaneamente;
- O canal de comunicação pode entregar as mensagens fora de ordem, perdê-las, corrompê-las ou duplicá-las devido ao timeout e retransmissões, processadores podem falhar e o link de comunicação pode cair (falhar).

Modelo de Sistema Distribuído

- Modelos de iteração: síncrona x assíncrona;
  - Sistema Distribuído Síncrono:
    - Tempo para executar cada etapa de um processo tem limites inferiores e superiores conhecidos;
    - Cada mensagem transmitida em um canal é recebida dentro de um tempo limitado, conhecido;
    - Cada processo tem um relógio local cuja taxa de desvio do tempo real tem um limite conhecido;
  - Desta forma é possível estimar um conjunto de execuções em uma computação distribuída.
  - É possível estipular tempos limites para detectores de falhas em sistemas distribuídos;
  - São difíceis de garantir em ambientes reais.

Modelo de Sistema Distribuído

- Modelos de iteração: síncrona x assíncrona;
  - Sistema Distribuído assíncrono:
    - Não são considerados as velocidades de execução de processos. Pode ser em nano segundos a horas de processamento – arbitrariamente longo;
    - Não são levados em conta os atrasos de comunicação;
    - As taxas de desvio do relógio é arbitrária;
  - Ou seja, não faz nenhuma consideração sobre os intervalos de tempo envolvidos em qualquer tipo de execução.
  - A Internet é perfeitamente representada por este modelo: ftp. e-mail, www, p2p, etc.

Modelo de Sistema Distribuído

- Os sistemas são assíncronos devido, principalmente, à necessidade de compartilhar o tempo de processamento, canais de comunicação e acesso a rede entre os processos;

Modelo de Sistema

- Síncrono x Assíncrono
  - Consenso em Pepperland: os exércitos vermelho devem atacar juntas para vencer o exército azul;
  - Problema: dificuldade de comunicação entre os exércitos vermelhos;

Tolerância a faltas em sistemas distribuídos

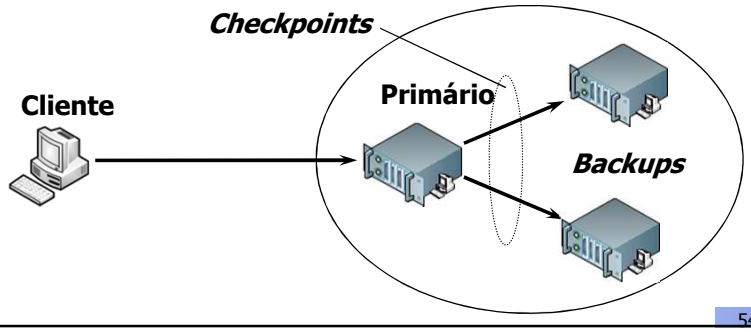
- Desenvolvimento de aplicações confiáveis:
  - Permitir que serviços sejam oferecidos continuamente mesmo em presença de falhas parciais no sistema;
- Técnicas de replicação:
  - Fazer várias cópias (réplicas) de um mesmo servidor e que na falhas de um deles, um outro possa tomar seu lugar;
- Técnicas de replicação:
  - Suporte de grupo:
    - Protocolos de difusão de mensagem (*Multicast*):
      - Atomicidade e ordenação das mensagens;
      - Estado consistente entre as réplicas;
    - Gerenciamento de grupo (*membership*):
      - Gerenciamento dinâmico dos membros;
      - Mecanismos de detecção de falhas;

Técnicas de replicação para tolerância a faltas

- É uma maneira usual para alcançar tolerância a faltas em sistemas distribuídos:
  - Melhorar a confiabilidade do sistema;
  - Aumentar a disponibilidade dos recursos;
- Podem ser classificadas em três abordagens:
  - Replicação passiva;
  - Replicação ativa;
  - Replicação semi-ativa;
  - Devem ser considerados o tipo de aplicação, a classe de falta que se deseja tolerar e as características do sistema distribuído.

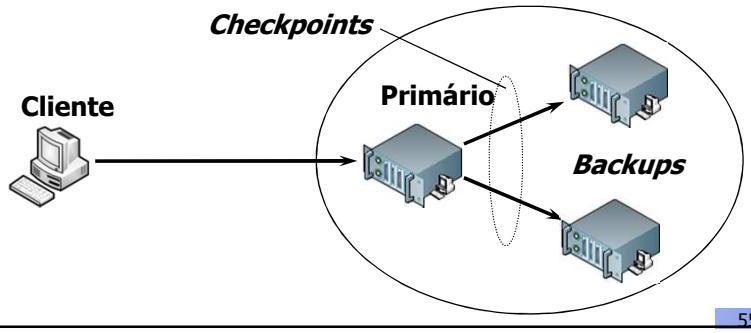
Técnicas de replicação para tolerância a faltas

- Replicação passiva:
  - Somente um membro (o primário) recebe, executa e responde as invocações dos clientes;
  - As réplicas restantes do grupo (os *backups*) têm a função de substituir o primário caso este falhe;



Técnicas de replicação para tolerância a faltas

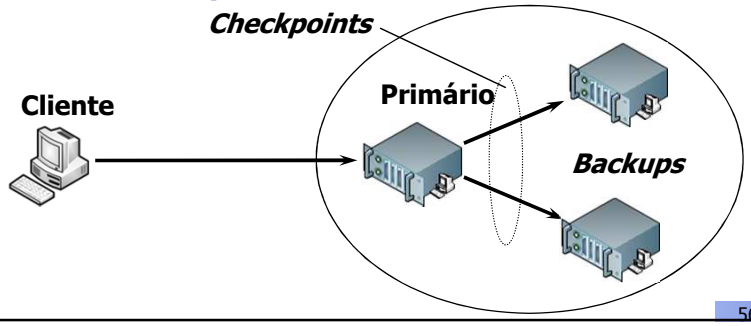
- Replicação passiva:
  - Somente um membro (o primário) recebe, executa e responde as invocações dos clientes;
  - As réplicas restantes do grupo (os *backups*) têm a função de substituir o primário caso este falhe;



Técnicas de replicação para tolerância a faltas

- Replicação passiva - Exemplo:

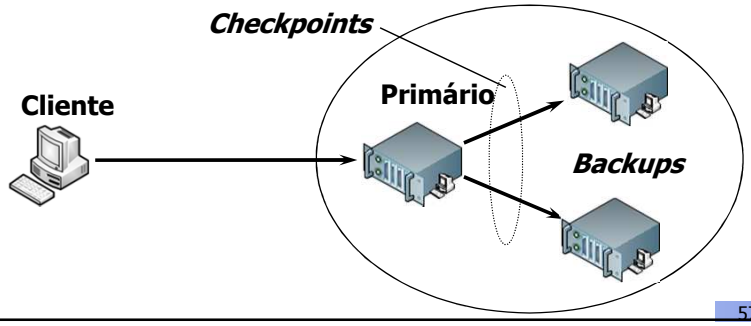
Dep(100);	Dep(100);	Dep(100);	Saldo () = 2100
Dep(100);	Dep(100);	Dep(100);	
Dep(100);	Dep(100);	Dep(100);	
Dep(100);	Dep(100);	Dep(100);	
Dep(100);	Dep(100);	Dep(100);	
Dep(100);	Dep(100);	Dep(100);	
Dep(100);	Dep(100);	Dep(100);	
Checkpoint(700)	Checkpoint(1400)	Checkpoint(2100)	



Técnicas de replicação para tolerância a faltas

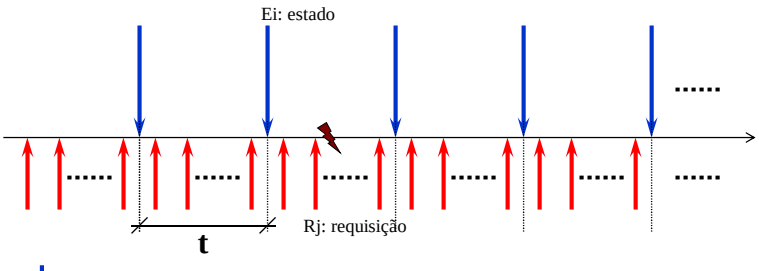
- Replicação passiva - Exemplo:

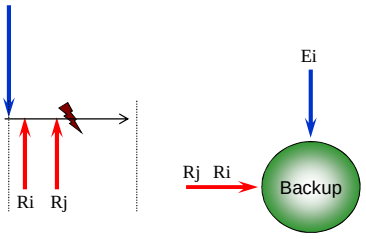
Dep(100);	Dep(100);	Dep(100);
Dep(100);	Dep(100);	Dep(100);
Dep(100);	Dep(100);	Dep(100);
Dep(100);	Dep(100);	Dep(100);
Dep(100);	Dep(100);	D…
Dep(100);	Dep(100);	…
Dep(100);	Dep(100);	
Checkpoint(700)	Checkpoint(1400)	Saldo (?)



Técnicas de replicação para tolerância a faltas

- *Checkpoint* e *log*





Checkpoint **– copia o estado do objeto primário.**

Requisições – salva as requisições no *log*.

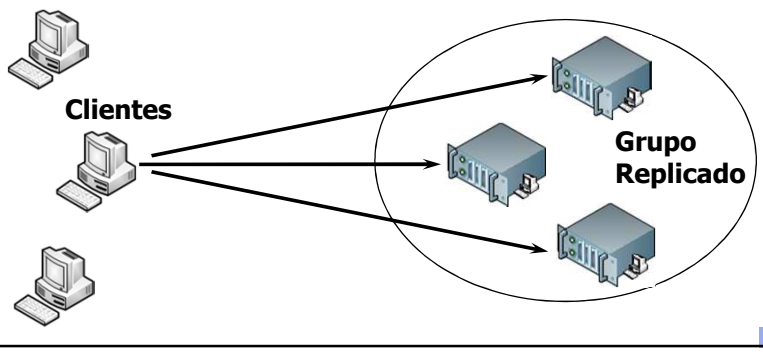
Técnicas de replicação para tolerância a faltas

- A ordenação das mensagens é imposta pelo primário (FIFO);
- O desempenho cai em situação de falha do primário;

- Mecanismos necessários:
  - Transferência de estado: *checkpoint, logs,* get_state e set_state;
  - Detecção de falhas: *timeouts* ou *keepalives*;
  - Variantes (exemplos):
    - *Coordinator e cohorts;*
    - Passiva quente;
    - Passiva fria.

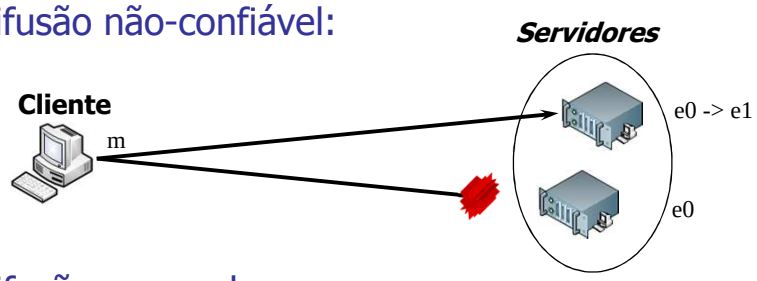
Técnicas de replicação para tolerância a faltas

- Replicação ativa:
  - Todos os membros recebem, executam e respondem as invocações dos clientes;
  - Combinando com outros mecanismos é capaz de tolerar todas classes de falhas;

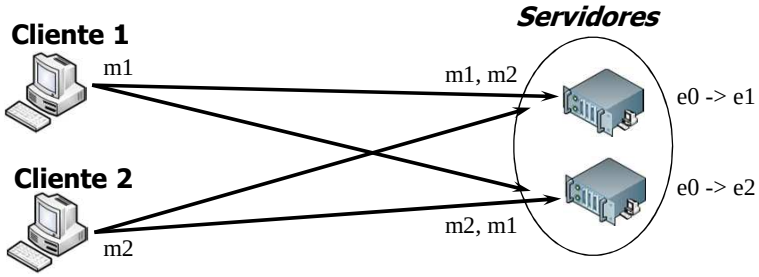


Problemas para difusão atômica

- Difusão não-confiável:



- Difusão sem ordem:

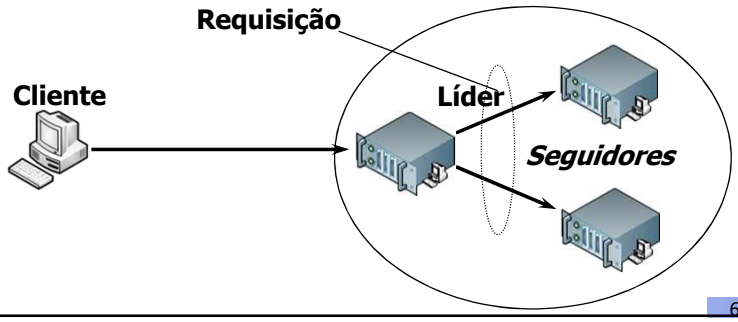


Técnicas de replicação para tolerância a faltas

- Aloca mais recursos do sistema (memória, processador, fluxo de mensagem, etc.);
- Determinismo de réplica:
  - Propriedades de acordo e ordenação;
- Mecanismos necessários:
  - Difusão atômica;
  - Detecção de falhas: *timeouts* e *keepalives*;
  - Gerenciamento de membros (*membership*);
  - Transferência de estados: get_state e set_state;
- Variantes (exemplos):
  - Ativa competitiva;
  - Ativa cíclica.

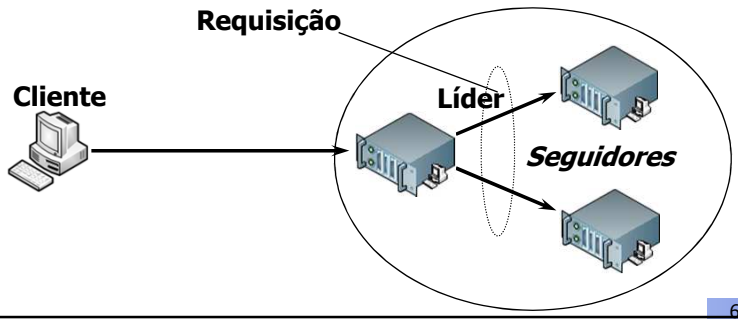
Técnicas de replicação para tolerância a faltas

- Replicação semi-ativa:
  - Todas réplicas são ativas, mas somente o líder recebe e responde as invocações dos clientes;
  - As réplicas restantes do grupo (os seguidores) têm a função de substituir o líder caso este falhe;



Técnicas de replicação para tolerância a faltas

- Replicação semi-ativa:
  - Todas réplicas são ativas, mas somente o líder recebe e responde as invocações dos clientes;
  - As réplicas restantes do grupo (os seguidores) têm a função de substituir o líder caso este falhe;



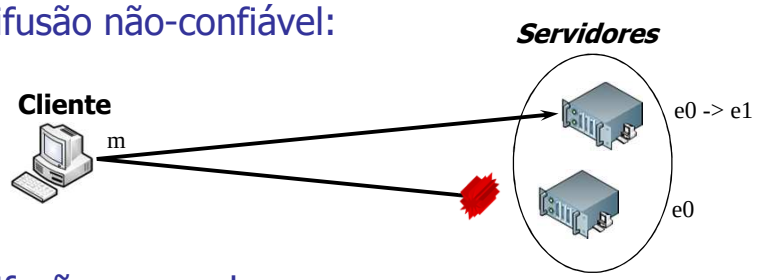
Técnicas de replicação para tolerância a faltas

- Aloca mais recursos do sistema (memória, processador, fluxo de mensagem, etc.);
- O determinismo de réplica é garantido pelo líder;

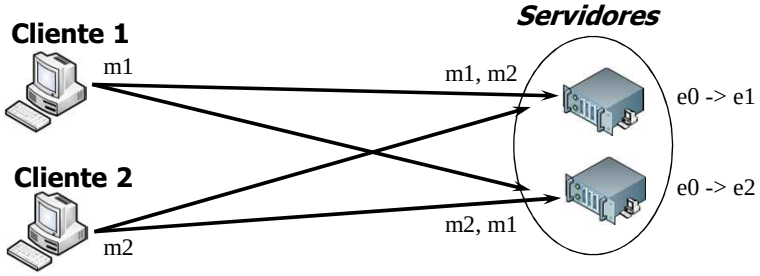
- Mecanismos necessários:
  - Difusão FIFO;
  - Detecção de falhas: *timeouts* e *keepalives*;
  - Gerenciamento de membros (*membership*);
  - Transferência de estados: get_state e set_state;

Problemas para difusão atômica

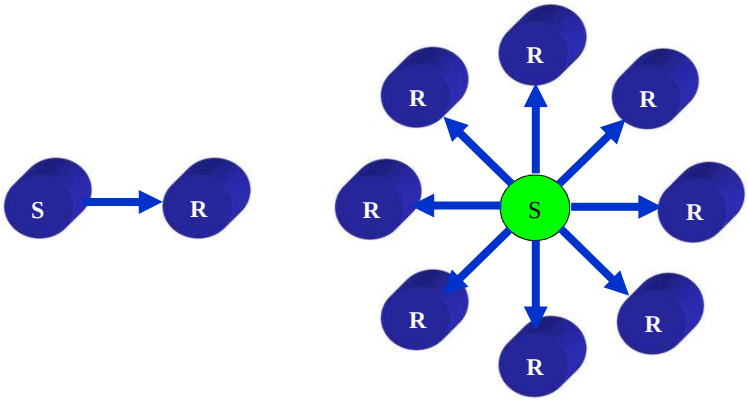
- Difusão não-confiável:



- Difusão sem ordem:

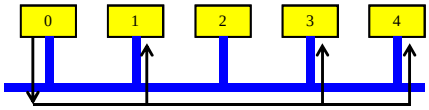


Difusão de mensagem

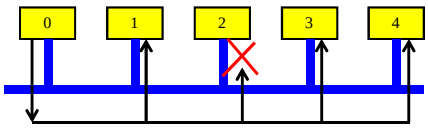


Comunicações ponto-a-ponto e um-para-muitos

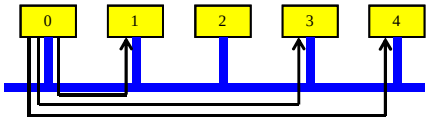
Difusão de mensagem



1 *Multicast*



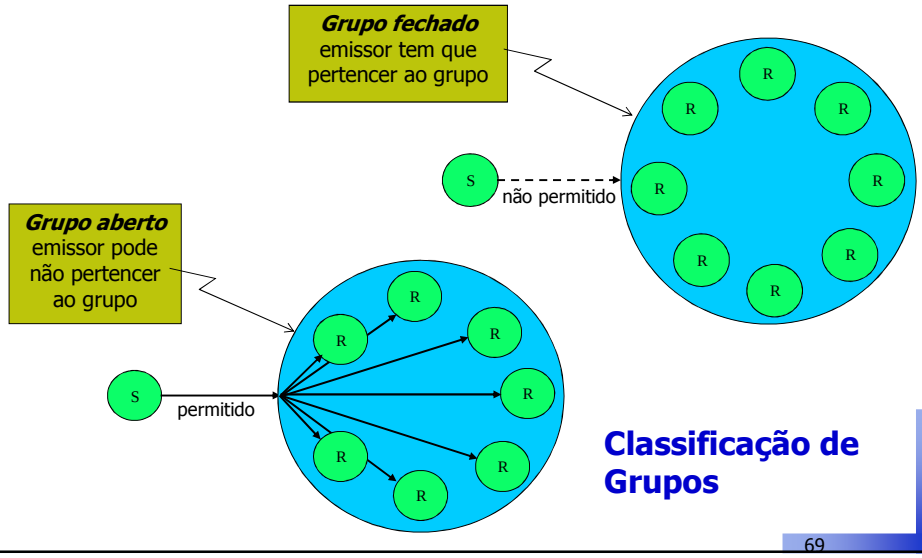
1 *Broadcast*



3 *Unicast*

Grupos: classificação

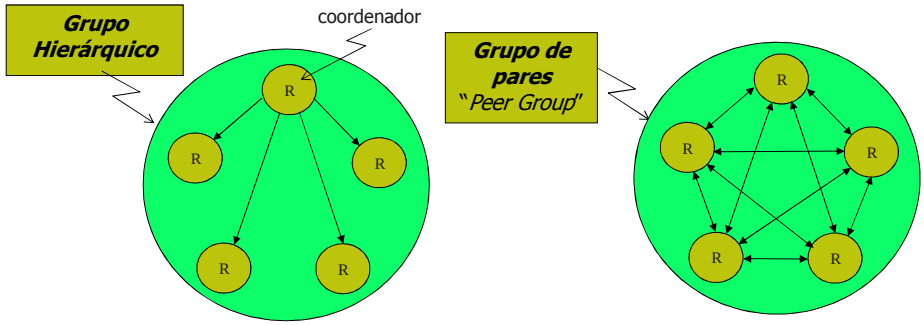
- **Grupos fechados X Grupos abertos**



Classificação de Grupos

Grupos: classificação

- Grupos simétricos X Grupos assimétricos



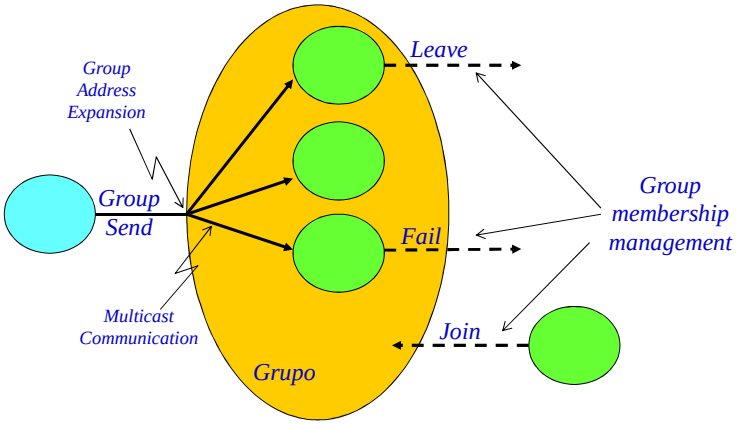
- Grupos simétricos: processos ou membros em papeis iguais na estrutura do grupo, grupo de pares
  - Vantagem sem ponto singular de falha
  - Desvantagem custos devido a gestão distribuída
- Grupos assimétricos: hierarquias de processos (objetos) custo só em tempo de falha; para decidir qualquer coisa é simples.

Pertinência (*membership*)

Grupos dinâmicos: necessidade de um gerenciamento de grupo
- Grupos podem ser criados e destruídos em tempo de execução
- Um processos (objeto) pode se juntar ou sair de um grupo

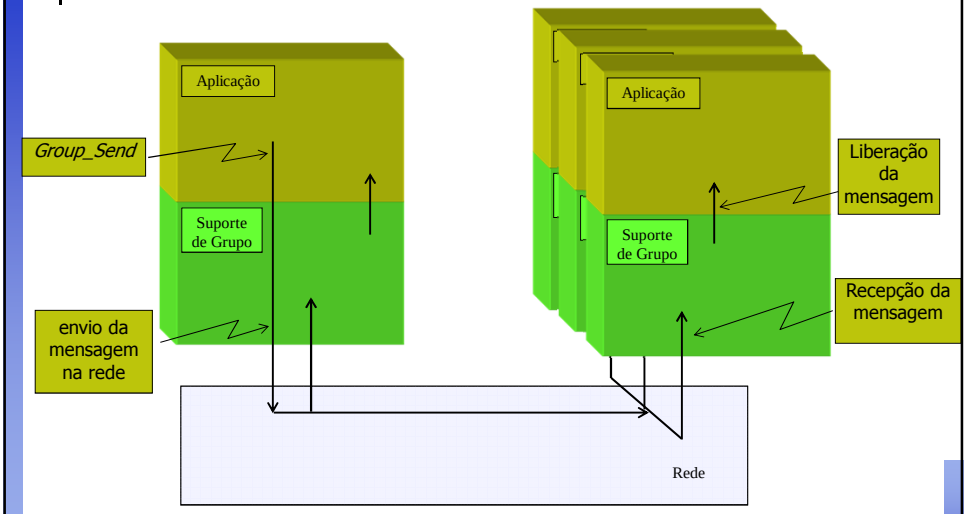
- Como manter as informações de pertinência?
  - Um serviço ⇒ solução centralizada
  - Uma solução distribuída ou descentralizada
- Protocolos distribuídos de *membership* ⇒ vista consistente dos componentes do grupo (*view*) em diferentes membros
  - Detectores de falhas dão o suporte ao serviço de membership
  - A entrada e saídas operações sincronizadas com as mensagens enviadas ao grupo. As informações de *membership* são enviadas pelos mesmos fluxos de mensagens de aplicação e estão sujeitas as mesmas propriedades de entrega.

Comunicação de Grupo



- Grupo aberto no qual um processo fora do grupo envia uma mensagem e sem conhecer o membership.
- O serviço de comunicação de grupo tem que gestionar mudanças no membership enquanto um multicast ocorre simultaneamente.

Modelo de comunicação de grupo



Comunicação de Grupo: Modelo

Algoritmo de difusão confiável

- Algoritmos de multicast confiável partem quase sempre da premissa de primitivas de multicast construídas sobre serviços subjacentes de comunicação ponto a ponto confiável e ordenação FIFO (First In First Out).

Com $p,q \in \wp$ e $m \in M$

{ R-*multicast*(*G*,*m*)}
  for all $p \in G$ **do**
    send (*m*,*p*);
  end for;

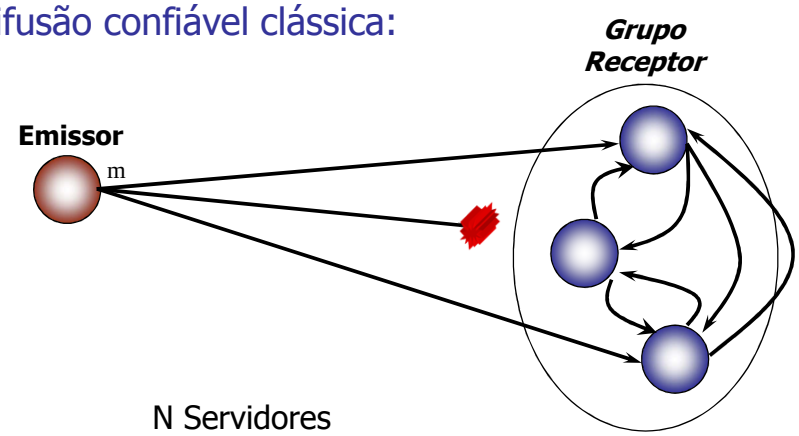
{ R-*deliver* (*m*)}
  receive(*m*) for the first time
  if *sender*(*m*) $\neq p$ **then**
    R-multicast(*G*,*m*)
  end if
  R-*deliver*(*m*)

Algoritmo de multicast Confiável

Alternativa:
Algoritmos probabilísticos

Algoritmo de difusão confiável

- Difusão confiável clássica:



N Servidores

Requer N^2 mensagens

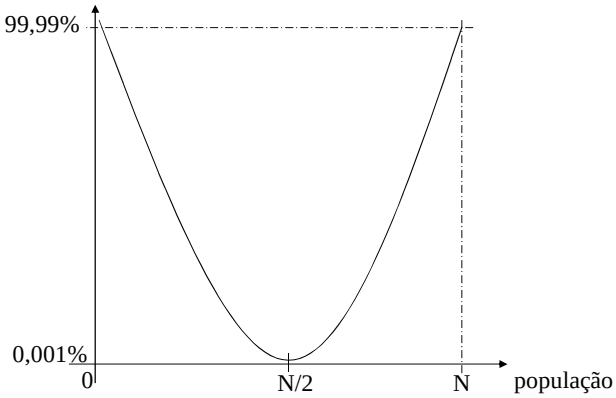
Grupos: propriedades

Propriedade *all-or-nothing* (confiabilidade da comunicação de grupo)

- acordo (ou atomicidade): a mensagem enviada ao grupo deve alcançar todos os processos membros ou nenhum destes.
  - protocolo de difusão confiável (*reliable broadcast*) garante propriedade *all-or-nothing.* implementação difícil várias trocas de mensagens
- Não interessa o número de perdas de pacotes e *crashes* de máquinas todos os processos corretos vão terminar recebendo a mensagem.

Algoritmo de difusão probabilístico

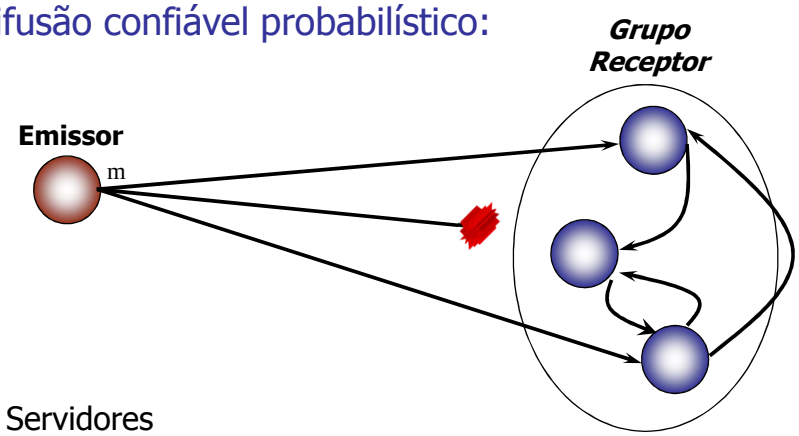
- Modelo epidêmico:



Como uma doença contagiosa
se espalha em uma população

Algoritmo de difusão confiável

- Difusão confiável probabilístico:



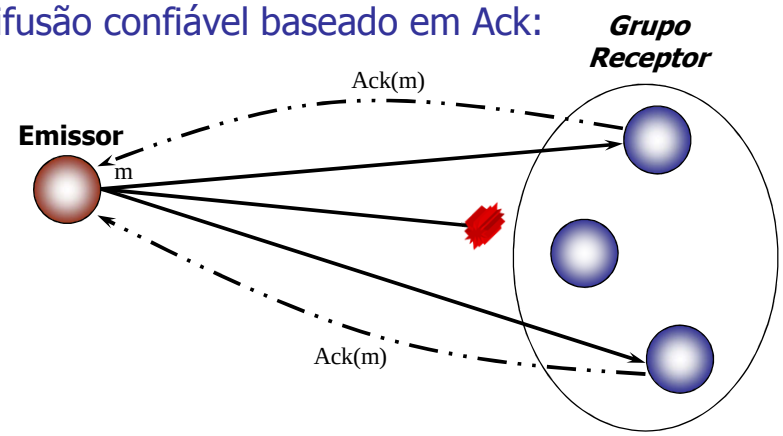
N Servidores

Requer ~$N^2/2$ mensagens

Quanto maior N mais próximo de 100%

Difusão confiável baseado em Ack e Nack

- Difusão confiável baseado em Ack:



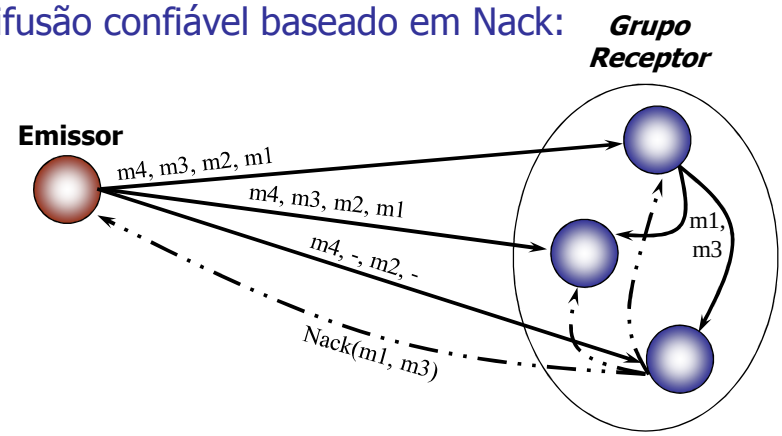
O emissor envia a mensagem e espera pelo reconhecimento (Ack) de cada receptor. Se o emissor não recebe um Ack de um receptor após um *timeout*, ele retransmite a mensagem para este.

Algoritmo de difusão confiável

- Difusão confiável baseado em Ack
  - Nesta abordagem, a correção de erro (retransmissão da mensagem perdida) é feita somente pelo emissor.
    - Problema: e se o emissor cai (*crash*) depois da primeira transmissão, quem vai fazer a correção de erro ?
  - Uma mensagem de reconhecimento (Ack) é enviada por cada receptor para cada mensagem recebida -> muito tráfego;
    - O ideal seria que todos processos receptores pudessem fazer a correção de erro também.
  - Como se calcula o *timeout* de espera para cada receptor ?
    - Testes para medir o atraso de uma mensagem -> timeout fixo;
    - Timeout adaptativo -> timeout variável.

Algoritmo de difusão confiável

- Difusão confiável baseado em Nack:



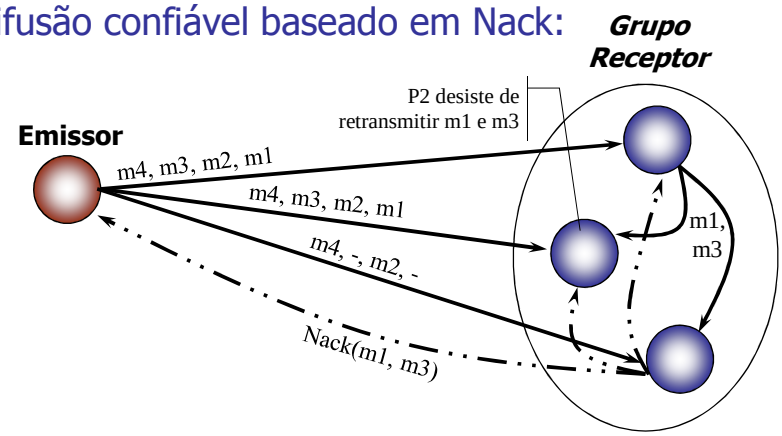
O emissor envia a mensagem para cada receptor. Se um receptor notar a falta de alguma mensagem de uma seqüência, ele envia um Nack para todos processos (emissor e receptores) pedindo retransmissão.

Algoritmo de difusão confiável

- Difusão confiável baseado em Nack
  - A retransmissão pode ser feita por qualquer processo.
  - Como evitar a explosão de retransmissões ?
    - Cada processo ao receber um Nack, espera um tempo aleatório antes de fazer a correção de erro.
      - Se durante esse tempo ele perceber que outro processo já fez a retransmissão solicitada no Nack, ele cancela.
    - Buffer de mensagens recebidas: cada processo deve guardar as mensagens recebidas no buffer para poder retransmiti-las caso algum processo solicite (Nack).
      - Mas o buffer tem tamanho limitado, quando sei que posso eliminar alguma mensagem do buffer ??
      - Uma mensagem *m* só pode ser eliminada do buffer do processo *p* quando *p* tiver a certeza que todos outros processos já receberam esta mensagem.
        - Se todos já receberam *m*, pra que guardá-la no buffer ?☺

Algoritmo de difusão confiável

- Difusão confiável baseado em Nack:

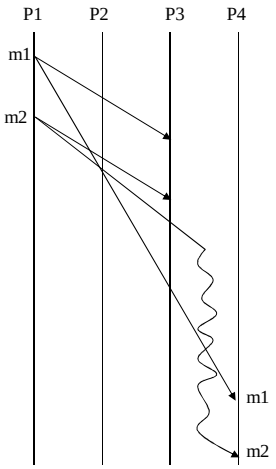


Para descarte de mensagens no buffer, todos processos mandam de tempos em tempos uma mensagem aos outros indicando as mensagens que já receberam.

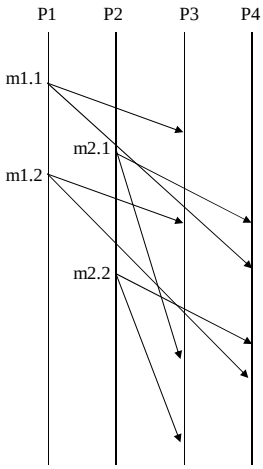
Difusão confiável com ordem FIFO

- A ordem FIFO é baseada na ordem de emissão. Pode ser implementado com número de seqüência na mensagem.

Exemplo 1: ordem FIFO



Exemplo 2: ordem FIFO



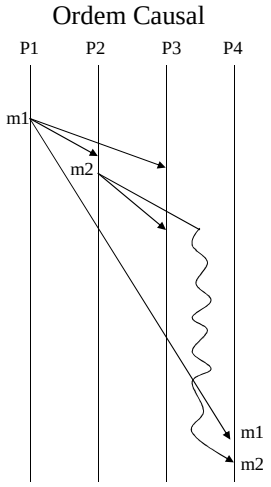
P3 recebe: m1.1, m1.2, m2.1 e m2.2

P4 recebe: m2.1, m1.1, m2.2 e m1.2

Em ambos casos a relação FIFO é atendida em P3 e P4.

Difusão confiável com ordem causal

- A ordem causal pode ser implementada colocando no cabeçalho da mensagem o(s) identificador(es) das mensagens que precedem a atual.



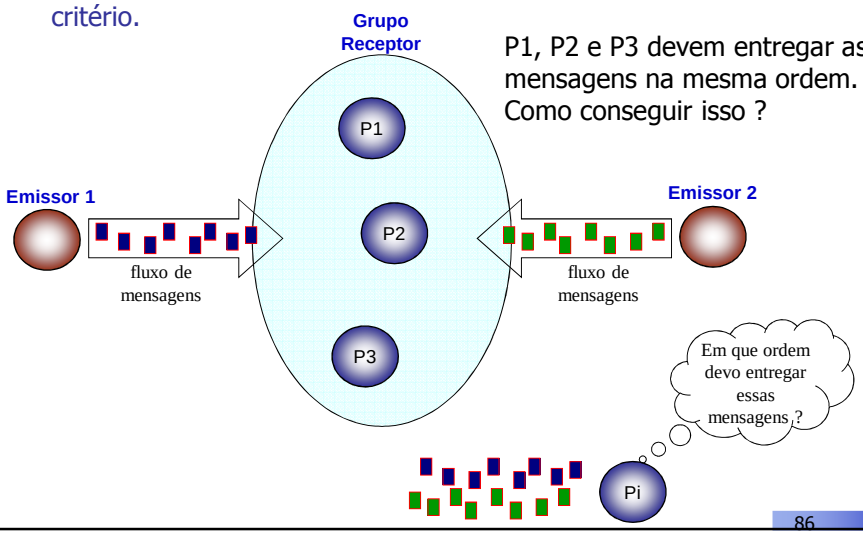
Relação de precedência m1 -> m2

A chegada da mensagem m1 fez com que P2 emitisse m2, logo m1 precede m2.

M_d(M_a, M_b, M_c) -> M_d só pode ser liberada para a aplicação se o processo tiver liberado M_a, M_b e M_c, pois essas mensagens precedem M_d.

Ordem total

- A ordem total visa garantir que todos processos recebam o mesmo conjunto de mensagens e na mesma ordem, baseada em qualquer critério.

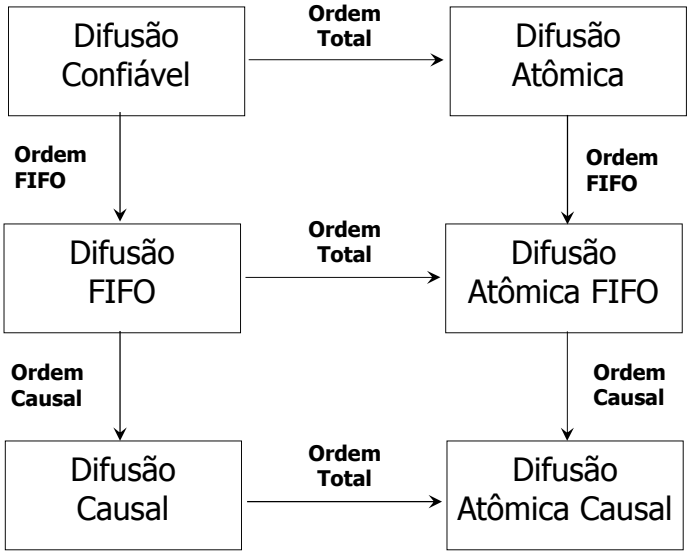


P1, P2 e P3 devem entregar as mensagens na mesma ordem. Como conseguir isso ?

Ordens totais

- ordem cronológica ou ordem de tempo global das emissões
  - Liberação da mensagem na ordem exata de emissão difícil de implementar esta ordem
- ordem consistente o sistema define uma ordem total qualquer das mensagens ⇒ todos os membros liberam segundo esta ordem.
- Protocolo que reúne as propriedades de ordem total (consistente ou de tempo global) e *all-or-nothing* é chamado difusão atômica.
- Outras ordens existem (ordem fifo, ordem causal, etc)

Relação entre as primitivas de difusão confiável

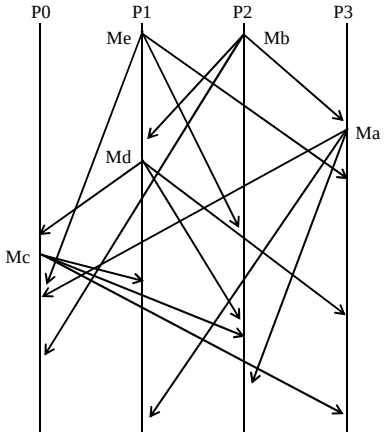


Mecanismos de ordenação total

- Histórico da comunicação: baseado no algoritmo de ordenação de eventos do Lamport;
- Baseado em privilégio: o token dentro de um anel lógico de processos. O processo de posse do token tem direito de multicast;
- Baseado em seqüenciador (fixo ou móvel): a mensagem é enviada pro seqüenciador e ele repassa a mensagem pro grupo;
  - Sequenciador repassa a mensagem (Tandem);
  - Sequenciador define a ordem de entrega (ISIS);
- Acordo no destino: a mensagem em enviada pro grupo que executa um protocolo de acordo para decidir sobre a ordem de entrega da mensagem.

Histórico da comunicação

- Ordenando eventos em sistemas de comunicação:
  - Uso de relógios sincronizados;
  - Histórico de comunicação (algoritmo de Lamport78)



P0: Md, Mc, Me, Ma, Mb
P1: Me, Mb, Md, Mc, Ma
P2: Mb, Me, Md, Mc, Ma
P3: Mb, Ma, Me, Md, Mc

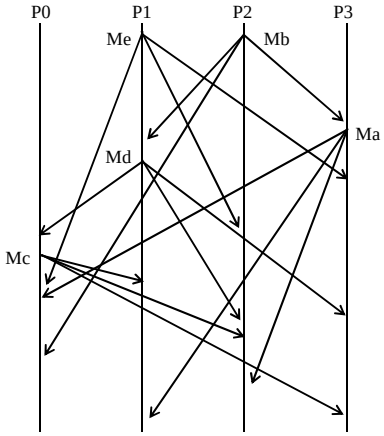
Como fazer com que todos os processos entreguem as mensagens numa única ordem ??

Ordenando eventos baseado em relógios físicos

- Solução 1: relógios sincronizados
  - Se todos os processos tiverem relógios sincronizados a solução é trivial.
    - Cada mensagem, antes de ser difundida, é marcada com o tempo no instante do envio (estampilha de tempo);
      - Uma mensagem m é definida por m(*id*, *ts*), onde:
        - *id* é o identificador do processo que enviou a mensagem;
        - *ts* é o instante de tempo em a mensagem foi enviada
    - Os processos receptores comparam as estampilhas de tempo de cada mensagem para estabelecer uma ordem única;
    - Se duas mensagens tiverem a mesma estampilha de tempo, elas são entregues de acordo com o *id* do emissor.

Ordenando eventos baseado em relógios físicos

- Uso de relógios físicos sincronizados;



Sem ordem:
P0: Md, Mc, Me, Ma, Md
P1: Me, Mb, Md, Mc, Ma
P2: Mb, Me, Md, Mc, Ma
P3: Mb, Ma, Me, Md, Mc

Com ordem baseado nas estampilhas de tempo:
P0: Me, Mb, Ma, Md, Mc
P1: Me, Mb, Ma, Md, Mc
P2: Me, Mb, Ma, Md, Mc
P3: Me, Mb, Ma, Md, Mc

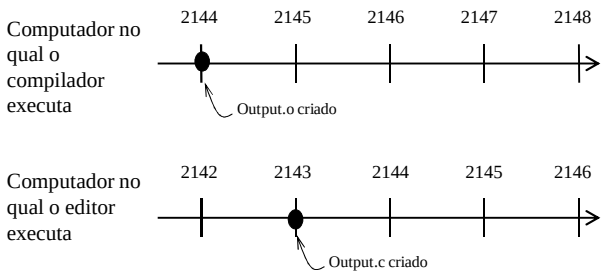
Me e Mb foram enviadas no mesmo instante de tempo (o que é quase impossível), a ordem de entrega dessas mensagens pode ser estabelecida, por exemplo, pelo processo que tiver o menos id (P1), desta forma, Me é entregue antes de Mb.

Tempo Global

- A solução usando relógios sincronizados é trivial, mas sincronizar relógios em sistemas distribuídos NÃO !
- É necessário determinar com exatidão o tempo no qual os eventos ocorrem nos sistemas computacionais
  - Registro de acesso ao sistema (login/logout)
  - Determinação do tempo de uso (ex.: telefonia, acesso à rede, vídeo sob demanda, videoconferência, etc.)
  - Registro de transações bancárias
  - Registro de compra/venda de produtos
  - Auditoria e segurança

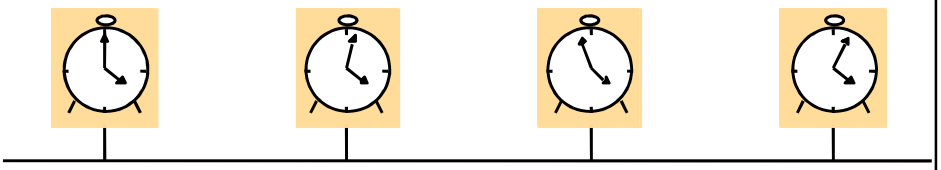
Sincronização de relógios

- A importância da sincronização de relógios, exemplo:
  - Arquivos fonte Input.c com tempo igual a 2151 e arquivo objeto Input.o com tempo igual a 2150 -> ocorre recompilação;
  - Arquivos fonte Output.c com tempo igual a 2144 e arquivo objeto Output.o com tempo igual a 2145 -> nada é feito;



Tempo Global

- Os relógios das máquinas não são sincronizados
  - Os relógios não são precisos: há um desvio entre o tempo real e o marcado pelo relógio
  - Mesmo que se acerte os relógios, com o passar do tempo eles perdem a sincronia



Sincronização de relógios

- **Relógios físicos**
  - Computadores têm um circuito para manter a trilha de tempo. Este circuito depende de um cristal de quartzo que quando mantido sob tensão oscila em uma freqüência bem definida que depende do:
    - Tipo de cristal;
    - Como ele foi cortado;
    - Quantidade de tensão aplicada.
  - Cada cristal tem um contador e um registro. Cada oscilação do cristal decrementa o contador em uma unidade e quando o contador alcança zero, uma interrupção é gerada e o contador recarregado de acordo com o valor do registro.
  - Desta forma é possível programar um temporizador para gerar uma interrupção 60 vezes por segundo, ou em qualquer outra freqüência. Cada interrupção é chamado um tique de relógio (clock tick).

Sincronização de relógios

- Relógios implementados usando cristal de quartzo podem sofrer uma diferença de 1 segundo a cada 1.000.000 de segundos (aproximadamente 11.6 dias). Portanto, em sistemas distribuídos, é necessário sincronizar os relógios dos computadores de tempos em tempos.
- A sincronização de relógios físico é baseada num relógio de referência baseado no movimento solar chamado de UTC (*Universal Time Coordenated*), fornecido pelo NIST (*National Institute of Standard Time*).

Sincronização de relógios

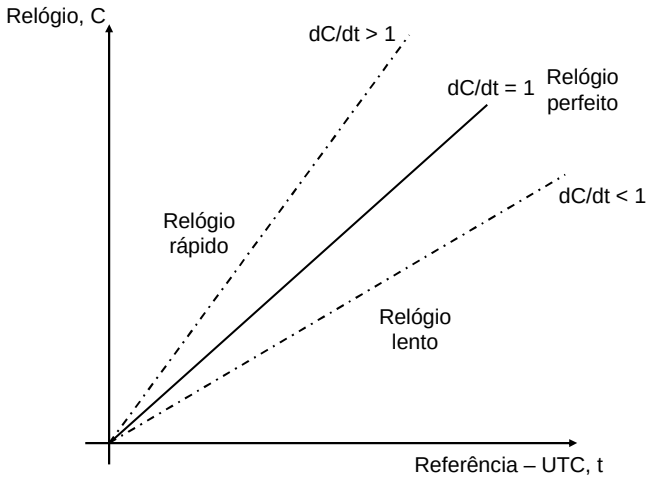
- Chamaremos o relógio local de C. Quando o UTC é t, o valor do relógio na máquina p é $Cp(t)$. Em um mundo perfeito, teríamos então $Cp(t) = t$ para todo p e todo t, ou seja, $dC/dt = 1$.
- No entanto, relógios reais não geram interrupções exatamente H vezes por segundo. Um temporizador usando quartzo com $H = 60$ deveria gerar 216.000 tiques por hora (60 tiques x 3600 seg). Na prática, o erro relativo é de 10^{-5} , isto é, a máquina pode obter valores entre 215.998 a 216.002 tiques por hora. Mais precisamente, se existe alguma constante ρ tal que:

$$(1 - \rho) \leq dC/dt \geq (1 + \rho)$$

o temporizador pode ser dito que trabalha dentro de sua especificação. A constante ρ é especificada pelo manufaturador e é conhecido como a taxa de desvio máximo.

Sincronização de relógios

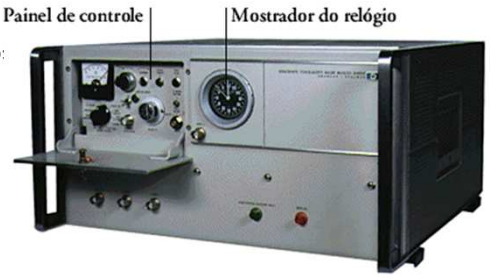
- O desvio de um relógio em relação a uma referência UTC.



Tempo Global

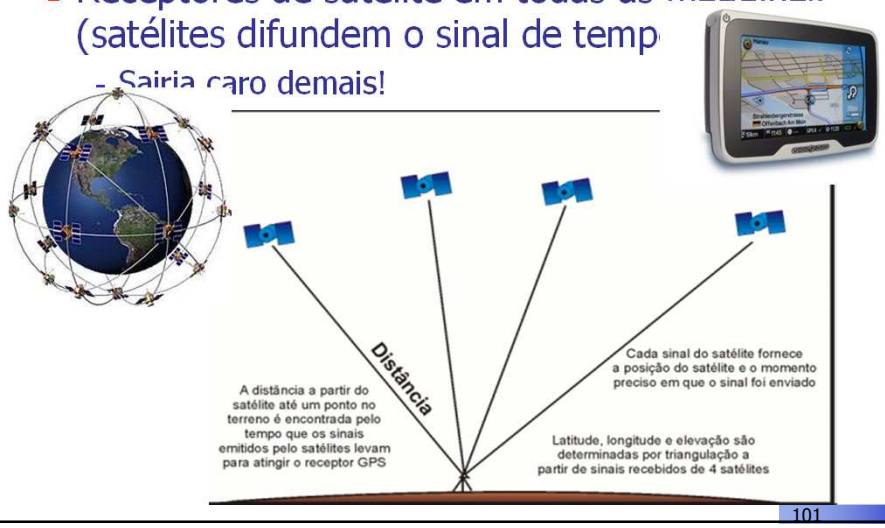
- Possíveis soluções:
  - Relógios atômicos colocados em todas as máquinas, marcando o tempo universal (UTC)
    - Inviável!

A medição do tempo para fins científicos deve ser muito precisa e o Relógio Atômico é o mais preciso de todos que existem atualmente. Ele mede as diminutas trocas de energia do interior dos átomos do metal Césio. Por serem muito regulares, as trocas criam um padrão preciso para medir o tempo. O Relógio Atômico mede as vibrações naturais dos átomos de Césio 133. Eles vibram mais de 9 bilhões de vezes por segundo, com isso, o Relógio Atômico atrasará poucos segundos a cada 100.000 anos.



Tempo Global

- Possíveis soluções:
  - Receptores de satélite em todas as máquinas (satélites difundem o sinal de temp
    - Sairia caro demais!

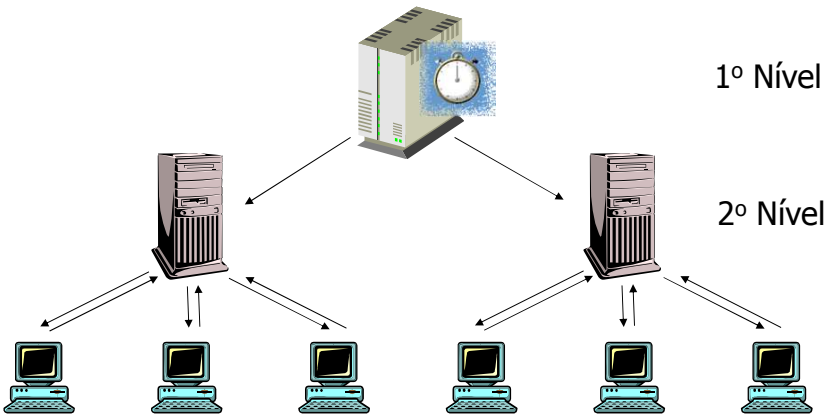


Tempo Global

- Possíveis soluções:
  - Sincronização pela rede
    - Servidores de tempo, com relógios precisos, difundem um sinal de tempo pela rede
- NTP (*Network Time Protocol*)
  - Hieraquia de Servidores
    - Servidores de 1º nível (ou primários): possuem relógios precisos
    - Servidores de 2º, 3º, ..., Nº nível: recebem o tempo do nível anterior
  - Clientes (máquinas dos usuários finais): perguntam o tempo a um servidor e atualizam seus relógios com base na resposta recebida

Tempo Global

- NTP (*Network Time Protocol*)



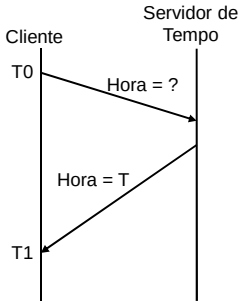
Tempo Global

- Características do NTP
  - Barato por não exigir relógios precisos (atômicos ou com GPS) em todas as máquinas
  - Impreciso devido ao tempo de propagação das mensagens pela rede não ser constante
    - O algoritmo de sincronização dos relógios leva em conta uma estimativa do tempo de propagação das mensagens pela rede
    - Mesmo assim, existe um erro implícito que pode chegar a dezenas de ms, e que deve ser levado em conta pelos sistemas

Sincronização de relógios

- **Algoritmos de sincronização**
  - Os algoritmos de sincronização podem ser centralizados ou distribuídos. Na abordagem centralizada podemos ter duas variantes: servidor passivo e servidor ativo.
- Centralizada passiva: o cliente envia uma mensagem perguntado as horas (Horas = ?) ao servidor no instante T0 no relógio do cliente. No instante T1, também no relógio do cliente, ele recebe a resposta.

$$Tr = \frac{(T1 - T0)}{2} + I$$

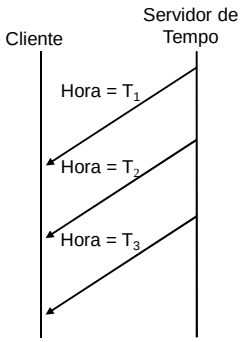


Sincronização de relógios

- Centralizada ativa: o servidor de tempo envia uma mensagem com as horas (Horas = T) aos clientes registrados a cada período de tempo.

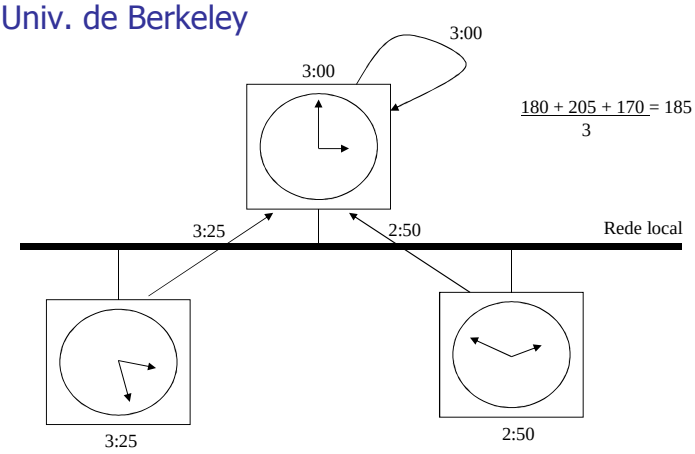
$$Tr = Ti + Tt$$

Tt: Tempo de transmissão



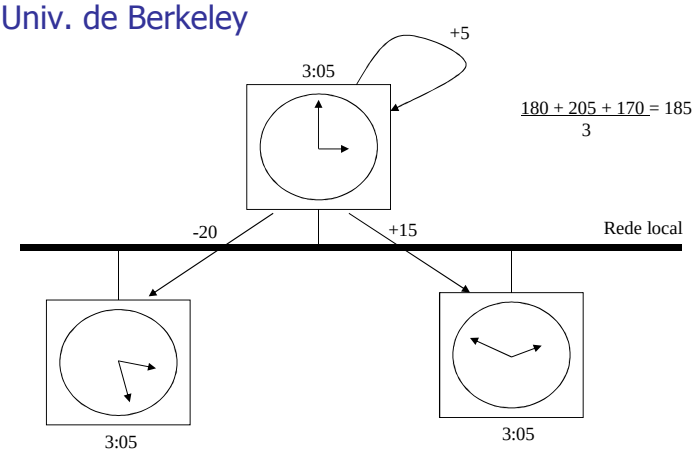
Sincronização de relógios

- Sincronização de relógios distribuída – algoritmo de Univ. de Berkeley



Sincronização de relógios

- Sincronização de relógios distribuída – algoritmo de Univ. de Berkeley

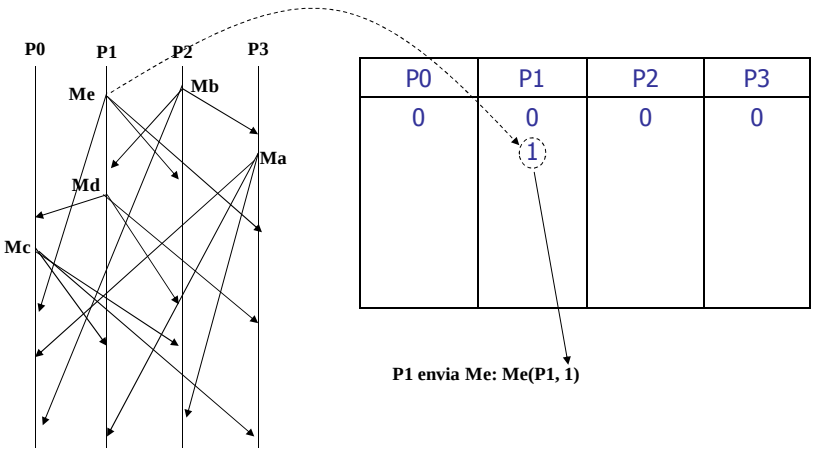


Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78
  - Lamport propôs um algoritmo simples que não precisa de relógios físicos para ordenar eventos.
    - Cada processo possui um contador local (que inicializa em zero) que incrementa cada vez que este difunde uma mensagem. Na mensagem deve conter o valor desse contador:
      - Mensagem: m(Pi, Ci), onde Pi é o id do processo e Ci é o valor do contador.
    - Quando um processo recebe uma mensagem ele compara com o valor do contador da mensagem com o seu contador local. Se for maior, ele adota esse valor. Se for menor ou igual, não faz nada.
    - Uma vez que todos processos vão receber o mesmo conjunto de mensagem m(Pi, Ci). A ordenação é baseada na valor de Ci de cada mensagem.
      - Mensagens com Ci igual, vai primeiro a que tiver o menor identificador Pi.

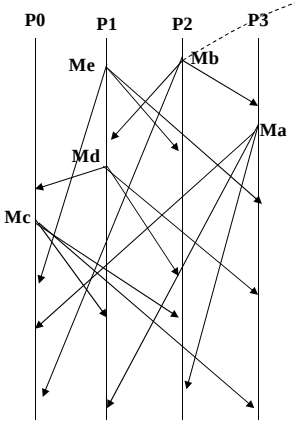
Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0 1	0 1	0

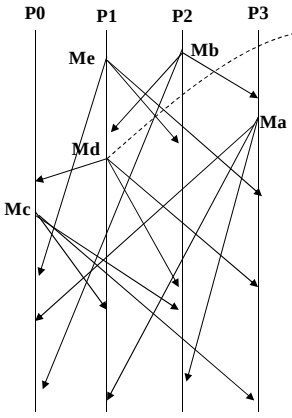
P2 envia Mb: Mb(P2,1)

P1 envia Me: Me(P1, 1)

56

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0 1 2	0 1	0

P1 envia Md: Md(P1,2)

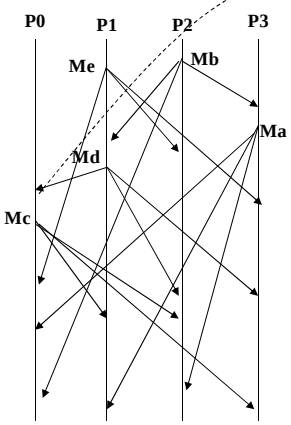
P1 recebeu Mb(P2,1): contador de P1 é 1 e continua com 1

P2 envia Mb: Mb(P2,1)

P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0	0	0
2	1	1	
	2		

P0 recebe Md(P1,2): contador de P0 é 0 e vai a 2

P1 envia Md: Md(P1,2)

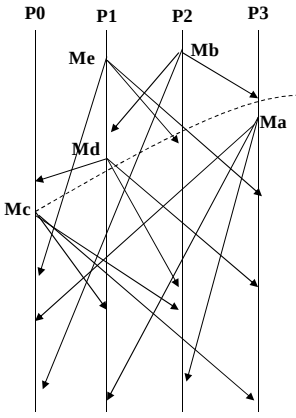
P1 recebe Mb(P2,1): contador de P1 é 1 e continua com 1

P2 envia Mb: Mb(P2,1)

P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0	0	0
2	1	1	
3	2		

P0 envia Mc: Mc(P0,3)

P0 recebe Md(P1,2): contador de P0 é 0 e vai a 2

P1 envia Md: Md(P1,2)

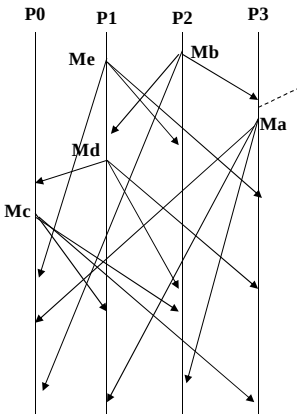
P1 recebe Mb(P2,1): contador de P1 é 1 e continua com 1

P2 envia Mb: Mb(P2,1)

P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78

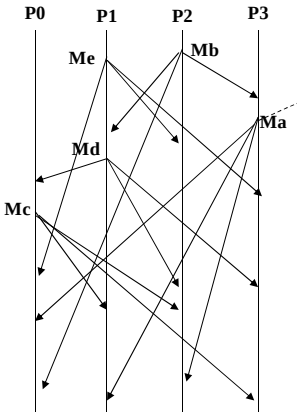


P0	P1	P2	P3
0	0	0	0
2	1	1	1
3	2	2	2
	3	3	3

P3 recebe Mb(P2,1): contador de P3 é 0 e vai a 1
P0 envia Mc: Mc(P0,3)
P0 recebe Md(P1,2): contador de P0 é 0 e vai a 2
P1 envia Md: Md(P1,2)
P1 recebe Mb(P2,1): contador de P1 é 1 e continua com 1
P2 envia Mb: Mb(P2,1)
P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78

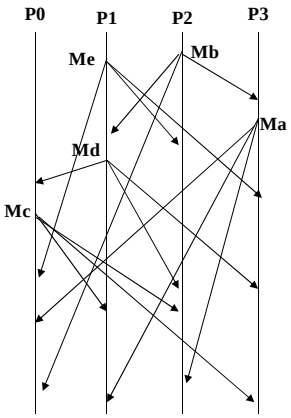


P0	P1	P2	P3
0	0	0	0
2	1	1	1
3	2	2	2

P3 envia Ma: Ma(P3,2)
P3 recebe Mb(P2,1): contador de P3 é 0 e vai a 1
P0 envia Mc: Mc(P0,3)
P0 recebe Md(P1,2): contador de P0 é 0 e vai a 2
P1 envia Md: Md(P1,2)
P1 recebe Mb(P2,1): contador de P1 é 1 e continua com 1
P2 envia Mb: Mb(P2,1)
P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0	0	0
2	1	1	1
3	2	2	2
	3	3	3

No final, todos recebem Mc(P0,3) e seus contadores locais vão a 3

P3 envia Ma: Ma(P3,2)

P3 recebe Mb(P2,1): contador de P3 é 0 e vai a 1

P0 envia Mc: Mc(P0,3)

P0 recebe Md(P1,2): contador de P0 é 0 e vai a 2

P1 envia Md: Md(P1,2)

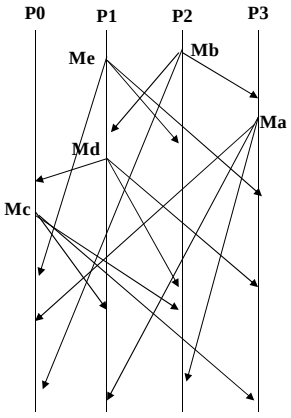
P1 recebe Mb(P2,1): contador de P1 é 1 e continua com 1

P2 envia Mb: Mb(P2,1)

P1 envia Me: Me(P1, 1)

Ordenando eventos baseados em relógios lógicos

- Solução 2: algoritmo de Lamport78



P0	P1	P2	P3
0	0	0	0
2	1	1	1
3	2	2	2
	3	3	3

Mensagens recebidas por cada processo:

Ma(P3,2), Mb(P2,1), Mc(P0,3), Md(P1,2), Me(P1, 1)

Ordenando-as baseado no contador:

Me(P1, 1), Mb(P2,1), Md(P1,2), Ma(P3,2), Mc(P0,3)

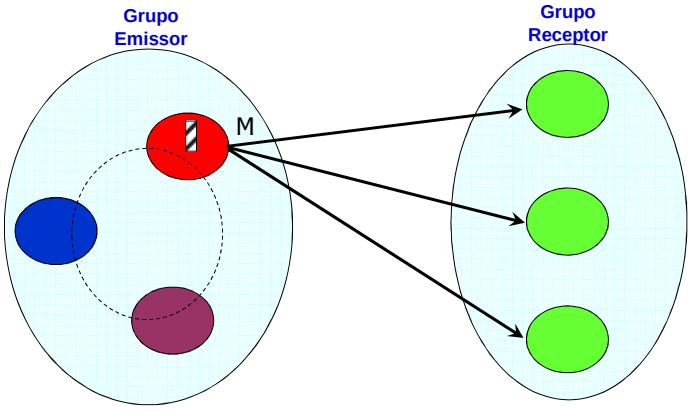
Desta forma todos processos entregam esse conjunto de mensagens e na mesma ordem (ordem total).

Mecanismos de ordenação total

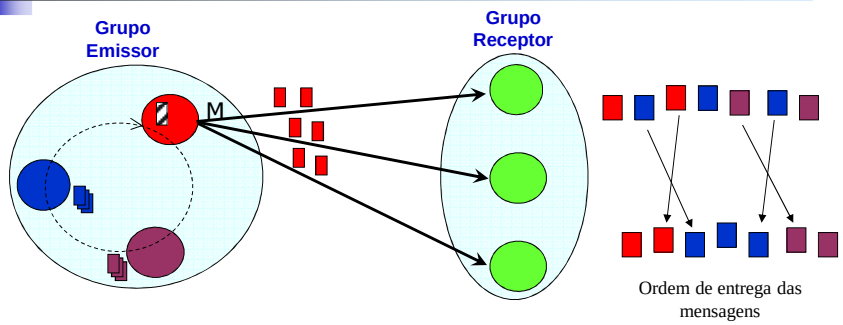
- Histórico da comunicação: baseado no algoritmo de ordenação de eventos do Lamport;
- Baseado em privilégio: o token dentro de um anel lógico de processos. O processo de posse do token tem o privilégio de *multicast*;
- Baseado em seqüenciador (fixo ou móvel): a mensagem é enviada pro seqüenciador e ele repassa a mensagem pro grupo;
  - Seqüenciador repassa a mensagem (Tandem);
  - Seqüenciador define a ordem de entrega (ISIS);
- Acordo no destino: a mensagem em enviada pro grupo que executa um protocolo de acordo para decidir sobre a ordem de entrega da mensagem.

Baseado em Privilégio

- Nesta técnica, os emissores formam um anel virtual onde circula um *token*. O emissor que estiver de posse do *token* tem o privilégio de difundir suas mensagens.



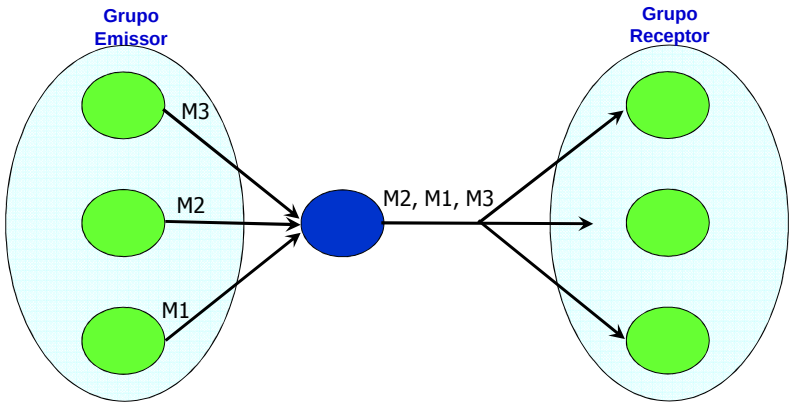
Baseado em Privilégio



- Problemas dessa abordagem:
  - Perda do token: processo que está usando o token falha;
  - Para um grupo com muitos membros, um emissor tem que esperar muito para poder enviar uma mensagem;
  - O grupo receptor tem que saber a seqüência do anel lógico dos receptores.

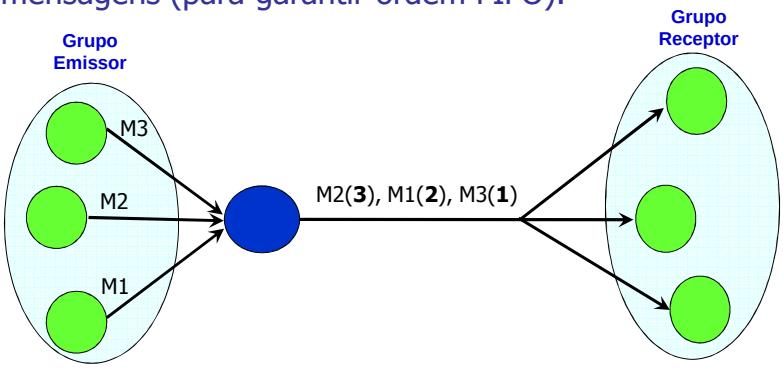
Seqüenciador Fixo

- Nesta técnica, os emissores enviam suas mensagens para um seqüenciador fixo. Este decide em que ordem as mensagens deverão ser entregues.



Seqüenciador Fixo

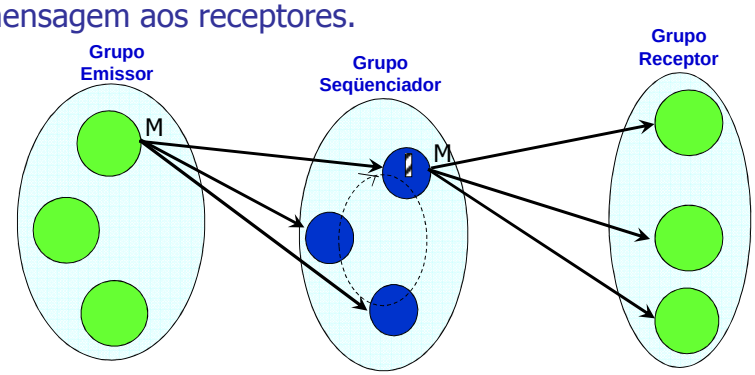
- Para garantir que os receptores entreguem as mensagens de acordo com a ordem estabelecida pelo seqüenciador um número de seqüência pode ser colocado no cabeçalho das mensagens (para garantir ordem FIFO).



- Problema: o seqüenciador é um simples ponto de falha. Se falhar todo o grupo fica indisponível.

Seqüenciador Móvel

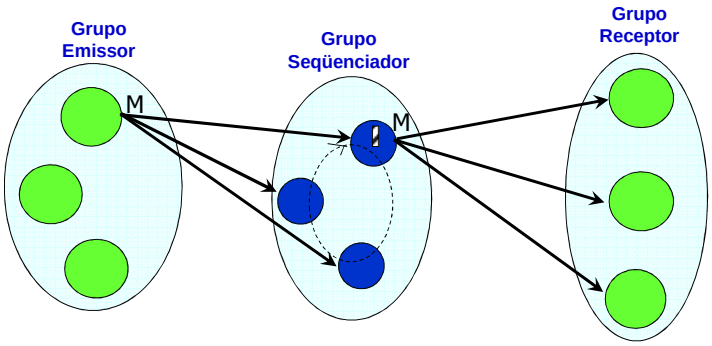
- Quando um emissor deseja enviar uma mensagem ao grupo receptor, ele difunde a mensagem no grupo seqüenciador. O seqüenciador que possuir o privilégio (token) repassa a mensagem aos receptores.



- O grupo seqüenciador não precisa ter muitos membros (2 a 4 membros).

Seqüenciador Móvel

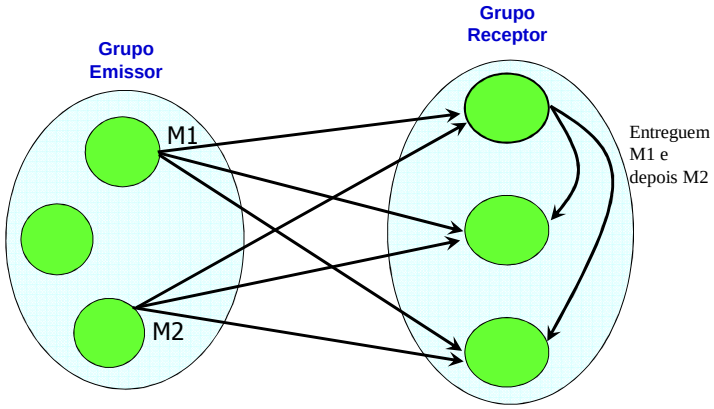
- Esta abordagem resolve o problema dos dois algoritmos anteriores: o grupo emissor poder ser escalável e não há um simples ponto de falha.



- Problemas dessa abordagem:
  - Perda do token: processo que está usando o token falha;
  - O grupo receptor tem que saber a seqüência do anel lógico dos seqüenciadores.

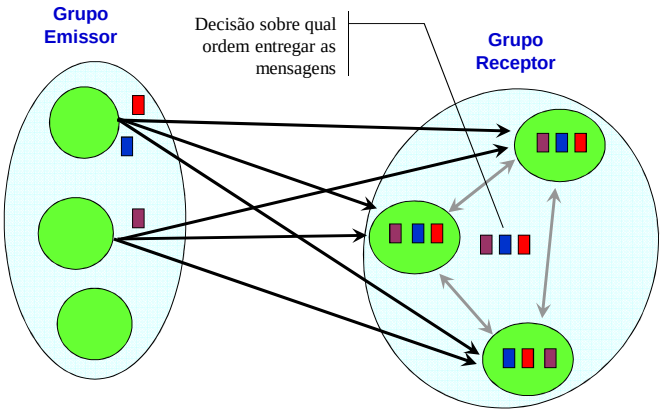
Seqüenciador ISIS

- Esta é uma variante de seqüenciador em que no grupo receptor tem um coordenador na qual, a cada período de tempo, envia uma mensagem aos outros indicando a ordem de entrega das mensagens.



Acordo no Destino

- As mensagens são difundidas (de forma confiável) diretamente para todos os membros do grupo receptor. Cada receptor faz uma proposta sobre a ordem de entrega e a decisão pode ser tomada pelo voto majoritário.



UDP e IP Multicast

UDP Multicast

- Permite o envio simultâneo de datagramas a grupos de destinatários
- Grupos multicast são identificados por endereços IP de 224.0.0.0 a 239.255.255.255

UDP Multicast

- Mais de um emissor pode mandar mensagens para o grupo (ou seja, mensagens vão de M emissores -> N receptores)
- Emissor não precisa fazer parte do grupo para enviar mensagens ao grupo, nem precisa saber quem são os seus membros; basta conhecer o endereço IP do grupo
- O receptor entra em um grupo (se torna um membro do grupo) e passa a receber as mensagens destinadas ao grupo