Sistemas Distribuídos

Tolerância a Falhas

Prof. Elias P. Duarte Jr.
Universidade Federal do Paraná (UFPR)
Departamento de Informática
www.inf.ufpr.br/elias/sisdis

Sumário

- Revisão das definições importantes da aula anterior
- Plano de tópicos desta edição de Sistemas Distribuídos
- Tolerância a Falhas → *Dependability*
  - Introdução
  - Nomenclatura da área
  - Modelos de falhas
  - Um resultado surpreendente
- Conclusão

Revisão dos Conceitos da Aula 1

- A aula 1 é extremamente importante: conceitos básicos que serão usados em todas as demais aulas
  - **Sistemas Distribuídos**: conjunto de processos que se comunicam e cooperam para realizar uma tarefa
  - Sistemas Distribuídos **Dinâmicos/Estáticos**
  - Paradigmas de Comunicação
    - ***Shared Memory & Message Passing***
  - Primitivas: ***send(msg), receive(msg), deliver(msg)***

Revisão dos Conceitos da Aula 1

- Modelos temporais de sistemas distribuídos
- **Sistemas Síncronos**: existem limites de tempo conhecidos para a transmissão de uma mensagem e a execução de uma tarefa
- **Sistemas Assíncronos**: sem limites conhecidos
- **Sistemas Parcialmente Síncronos**:
  - GST: *Global Stabilization Time*
  - *O sistema inicia assíncrono, mas a partir de um determinado instante de tempo fica síncrono para sempre*

Os Tópicos desta Edição de SisDis

- De acordo com a ementa na página da disciplina:
  - Introdução aos Sistemas Distribuídos √
  - Modelos Temporais: Sistemas Síncronos, Assíncronos e Parcialmente Síncronos √
  - Confiança no Funcionamento de Sistemas (*Dependability*) e Modelos de Falhas (hoje √)
  - Modelos de Enlaces (segunda)
  - Diagnóstico em Nível de Sistema (quarta)
  - Algoritmo Distribuído em Anel: VRing (segunda...)
  - Simulação com SMPL
  - Algoritmo Distribuído em Hipercubo: VCube
  - Ordenação de Eventos e Relógios Lógicos → Prova 1 em seguida
  - Difusão Confiável, FIFO, Causal e Atômica
  - Detectores de Falhas
  - Eleição de Líder
  - Consenso & Paxos

Tolerância a Falhas

- Por que é importante?

  As organizações (incluindo empresas) estão se tornando indistinguíveis de seus sistemas computacionais.

- Consequência: se a rede (o sistema computacional) não funciona: a própria organização entra em colapso!
- Fundamental garantir o funcionamento correto do sistema

Vulnerabilidades

- Pergunta:

  Se esperarmos tempo suficiente, qual a probabilidade de um componente computacional falhar?

  - no sentido de deixar de funcionar como esperado

Vulnerabilidades

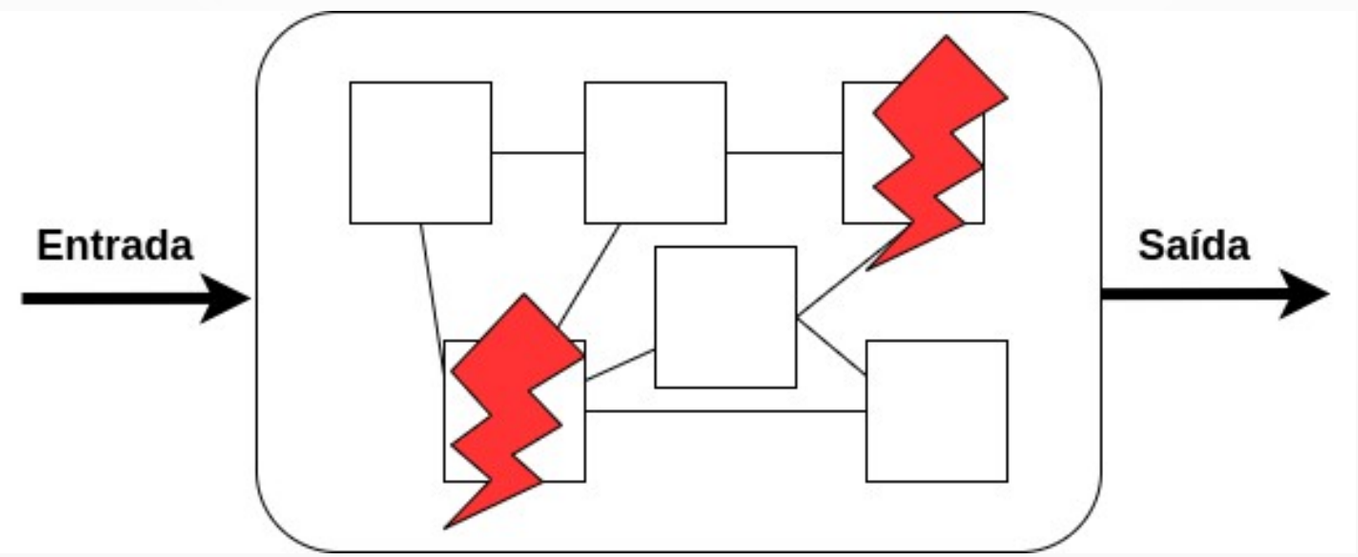
- Pergunta:

  Se esperarmos tempo suficiente, qual a probabilidade de um componente computacional falhar?

  - no sentido de deixar de funcionar como esperado
- Resposta: a probabilidade é 100%
- Sistemas computacionais falham, cedo ou tarde

Tolerância a Falhas

- Um sistema computacional é tolerante a falhas se *continua funcionando* (produzindo as saídas corretas para as entradas correspondentes) *mesmo se alguns de seus componentes estejam falhos*



Alguns Sistemas Devem ser *Obrigatoriamente* Tolerantes a Falhas

- Sistemas de aviação: devem ser comprovadamente tolerantes a falhas
- Sistemas de Missão Crítica (*Mission Critical*)
- Controle de Usinas, Monitoramento Hospitalar, Ferroviários...
- A falha do sistema causa desastre, possivelmente com a perda de vidas humanas

Em Outros Casos: Desejável

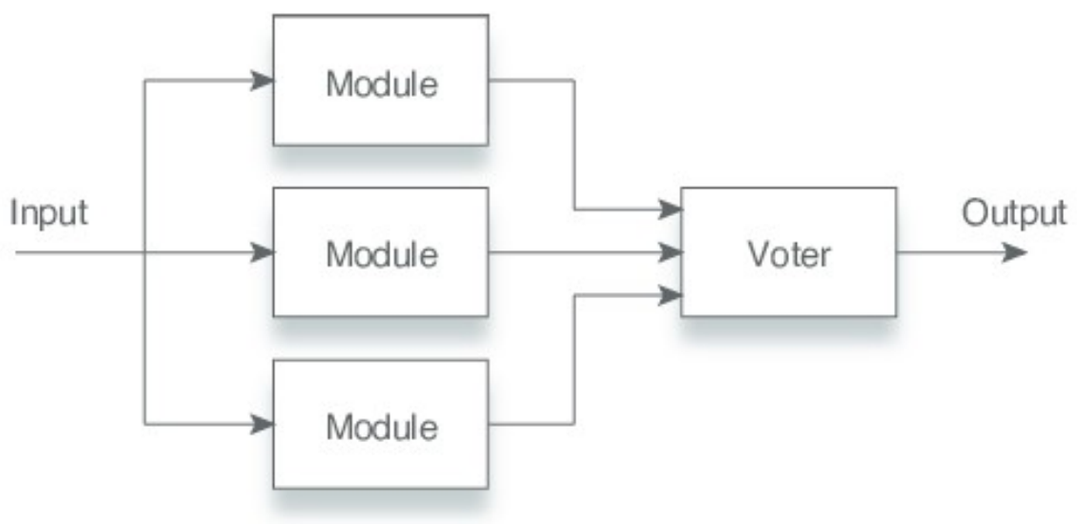
- Desejamos que o sistema computacional seja robusto, resiliente, confiável, disponível
- Em alguns casos, é mais que simplesmente um “desejo” de confiabilidade
- Por exemplo: falhas de um sistema bancário podem causas prejuízos de milhões
- Outro exemplo: falha de uma loja on-line na *Black Friday*
- O sistema simplesmente “não pode” falhar!

Base da Tolerância a Falhas: **Redundância**

- Para o sistema continuar funcionando mesmo que um componente falhe...
- ... deve haver outro componente que substitui aquele que falhou
- A redundância pode ser explícita: e.g. para tolerar a falha de um HD você pode instalar um segundo HD e fazer espelhamento
- A redundância pode ser também implícita: e.g. no roteamento tolerante a falhas, quando falha a rota principal, outra é definida com base na topologia

Uma Técnica Popular (não SisDis): *Triple Modular Redundancy*

- Um módulo é triplicado, as 3 unidades executam em paralelo, mesmas entradas
- Uma unidade de votação (*voter*) recebe as três saídas e seleciona a resposta da maioria
  - tolera a falha de 1 unidade
- Para bugs de software: *design diversity, N-version programming*



Redundância *Implícita* em Sistemas Distribuídos

- Por definição, um sistema distribuído consiste de um grupo (>1) de processos
- Desta forma: intrinsecamente redundante!
- “Sopa no mel” para tolerância a falhas

Dependability: Os Termos da Área

- Em português, a CE-TF da SBC define que o termo *dependability* deve ser traduzido como “confiança no funcionamento” [do sistema]
  - *Comissão Especial de Computação Tolerante a Falhas da Sociedade Brasileira de Computação*
- Dependability é um conjunto atributos, todos precisamente definidos

Dependability: Atributos

- Os atributos *clássicos* que compõem a *Dependability* de um sistema são:

1) *Reliability* (Confiabilidade) – medida quantidade da continuidade de funcionamento correto
   - Probabilidade do sistema oferecer serviço correto ao longo de um intervalo de tempo de duração *t*
   - Mede-se até o sistema falhar! Daí acabou, só mede de novo a partir da recuperação
   - Outra medida relacionada: *Mean Time Between Failures (MTBF)*, muito usada, o tempo médio entre a ocorrência de falhas

Dependability: Atributos

2) Availability (Disponibilidade) – medida da esperança de obter serviço ao acessar o sistema

- Medida que inclui períodos de falha e recuperação: a probabilidade do sistema estar funcionando corretamente em um instante de tempo
- Após observar o sistema durante um intervalo de tempo (p.ex. 1 semana) → qual a porcentagem do tempo em que o sistema ficou disponível?
- Termo correlato muito importante: *high-availability*, *highly available system:* observando o sistema durante 1 ano, os períodos de indisponibilidade podem ser expressados [confortavelmente] na ordem de minutos

Dependability: Atributos

3) *Safety* (Segurança-Safety) – medida da probabilidade do sistema evitar consequências catastróficas

- Baseada em modelos matemáticos do sistema e seu ambiente, com provas formais da capacidade do sistema de evitar catástrofes
- Imprescindível nos sistemas de missão crítica

4) *Security* (Segurança) – a área clássica de segurança computacional

- Também consiste de atributos, no mínimo:
- Sigilo, Autenticidade, Integridade
- Segurança é parte de Tolerância a Falhas e vice-versa ;-)

Dependability: A Referência Clássica

- Leitura obrigatória!
- Altera levemente estes atributos
- Ao invés de *segurança*: *integridade*
- Inclui *maintainability* – facilidade de alteração, manutenção do sistema

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl E. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transations on Dependable and Secure Computing*, pp. 11-33, Vol. 1, No. 1, 2004.

MTBF & MTTR

- Duas métricas importantes para a gerência de falhas de um sistema
- Como medir?
- Monitorando o sistema: nas redes pode-se usar um sistema de gerência de redes
- MTBF: *Mean Time Between Failures*
  - Intervalo de tempo em que o sistema fica continuamente correto
- MTTR: *Mean Time To Repair*
  - Intervalo de tempo em que o sistema fica continuamente falho

Um Conceito Chave: **Falha**

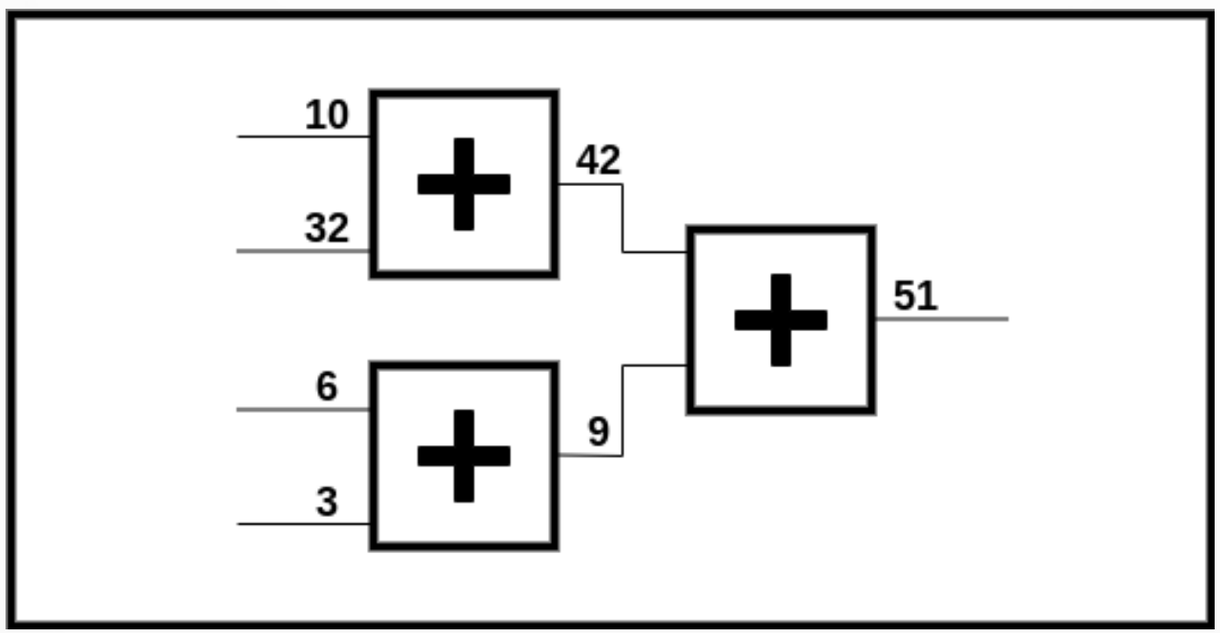
- Estamos aqui falando em tolerância a falhas mas ainda não definimos precisamente o que é uma falha
- Palavra muito usada na Ciência da Computação, nos mais diferentes contextos
- Na área de *Dependability*, definição precisa!
- Intuição: o sistema (ou módulo) não está funcionamento como deveria – isto é: de acordo com sua especificação

Fault-Error-Failure

- Três níveis diferentes do mesmo fenômeno
- *Fault*: defeito do sistema, decorrente de sua fabricação, projeto ou implementação
  - Está lá no sistema, mas não se manifestou!
  - No Brasil e Portugal: alguns grupos traduzem como ***falha*** outros como ***falta***, na Engenharia de Software: ***defeito***
- *Error*: ***erro***, ocorre quando a falha (fault) é ativada e se manifesta, por exemplo na saída de uma operação
- *Failure*: ***falha-colapso***, a falha se propaga para a saída do sistema

Um Exemplo: *Fault-Error-Failure*

- Considere um sistema que soma 4 inteiros a partir de módulos que somam 2 inteiros cada:

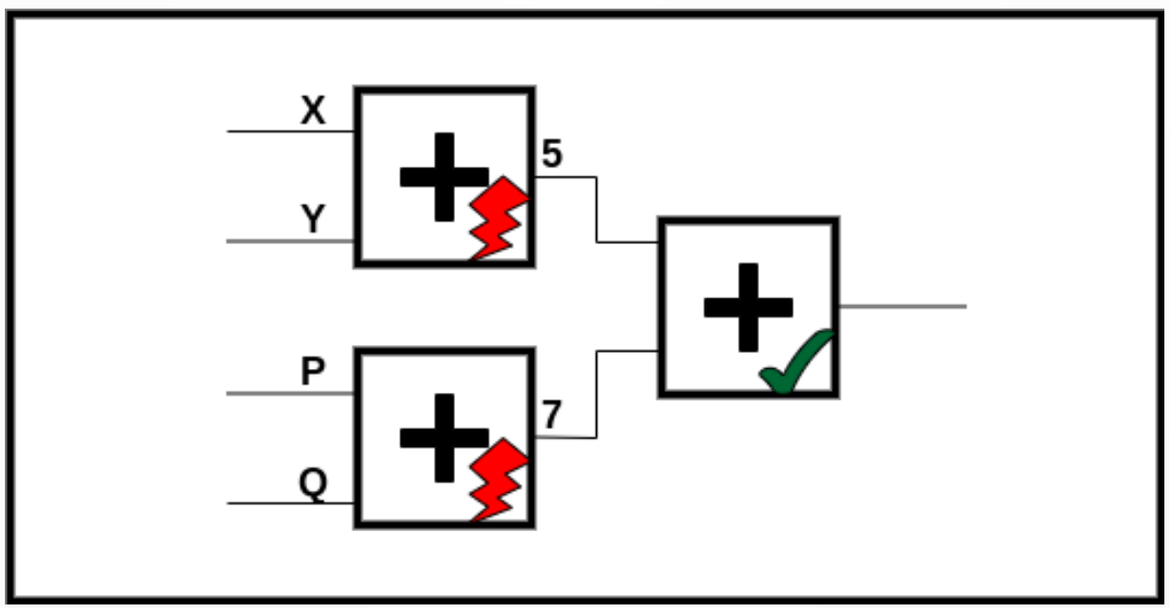


Exemplo de Falha (*Fault*)

- Um tipo de falha comum em hardware: *stuck-at-0* e *stuck-at-1*: a saída da porta lógica/circuito não varia: sempre 0 e sempre 1
- Considere que os dois primeiros somadores estão com falhas deste tipo produzindo sempre as saídas 5 & 7

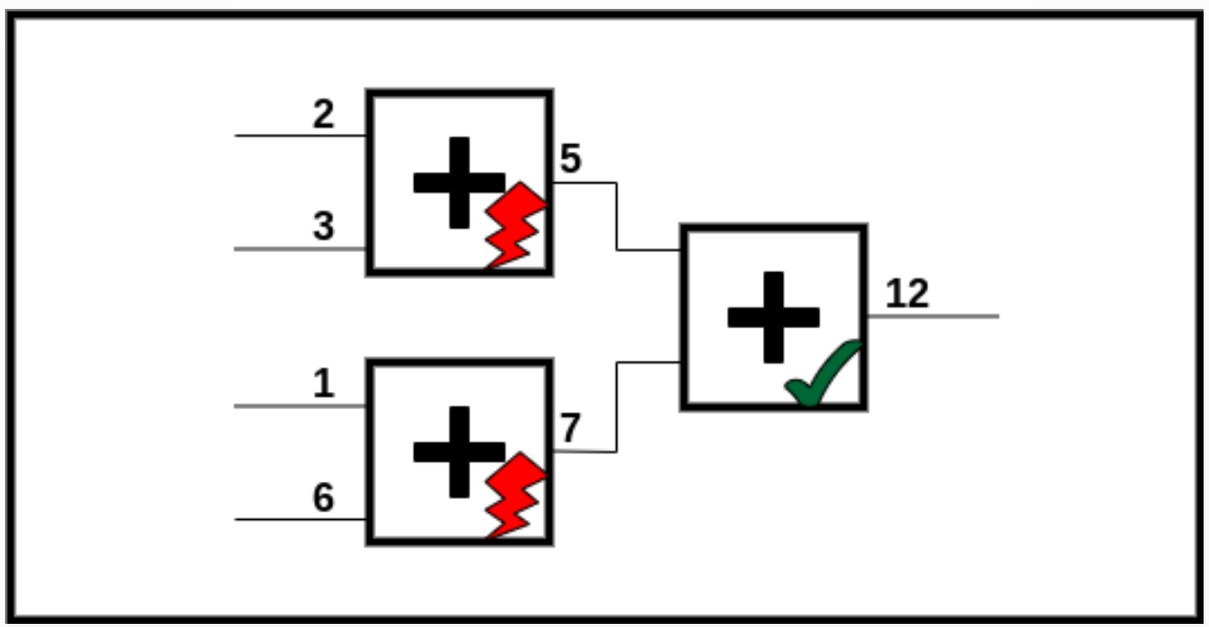
Um Exemplo de Falha (***Fault***)

- Dois somadores falhos: *stuck-at-5*, *stuck-at-7*



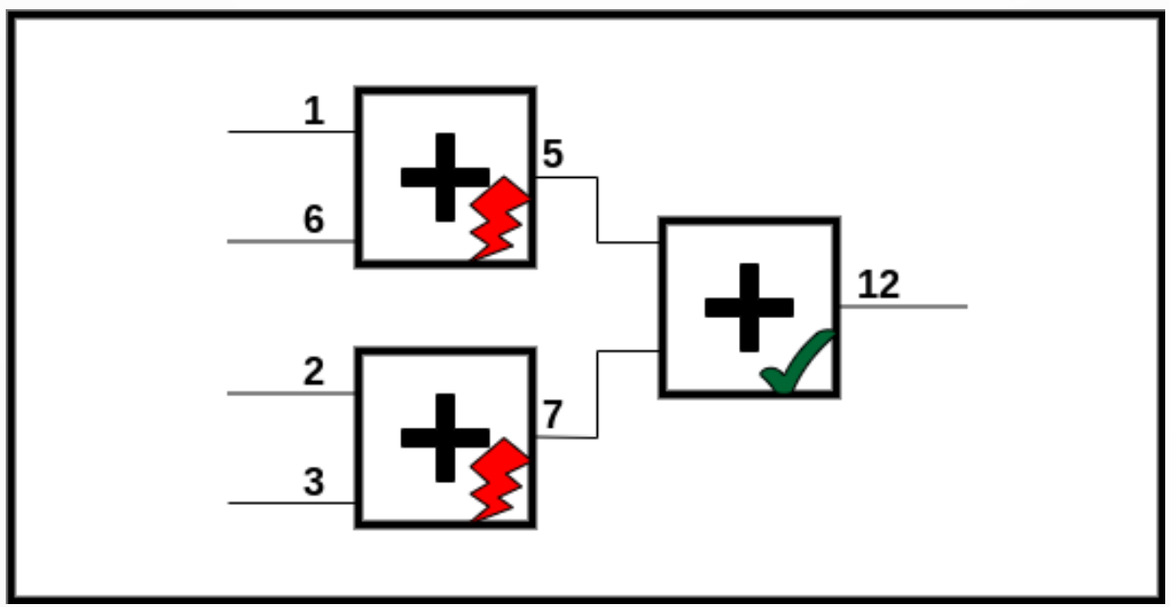
Fault sim, *Error-Failure* não

- Veja que as saídas de todos os somadores estão corretas!



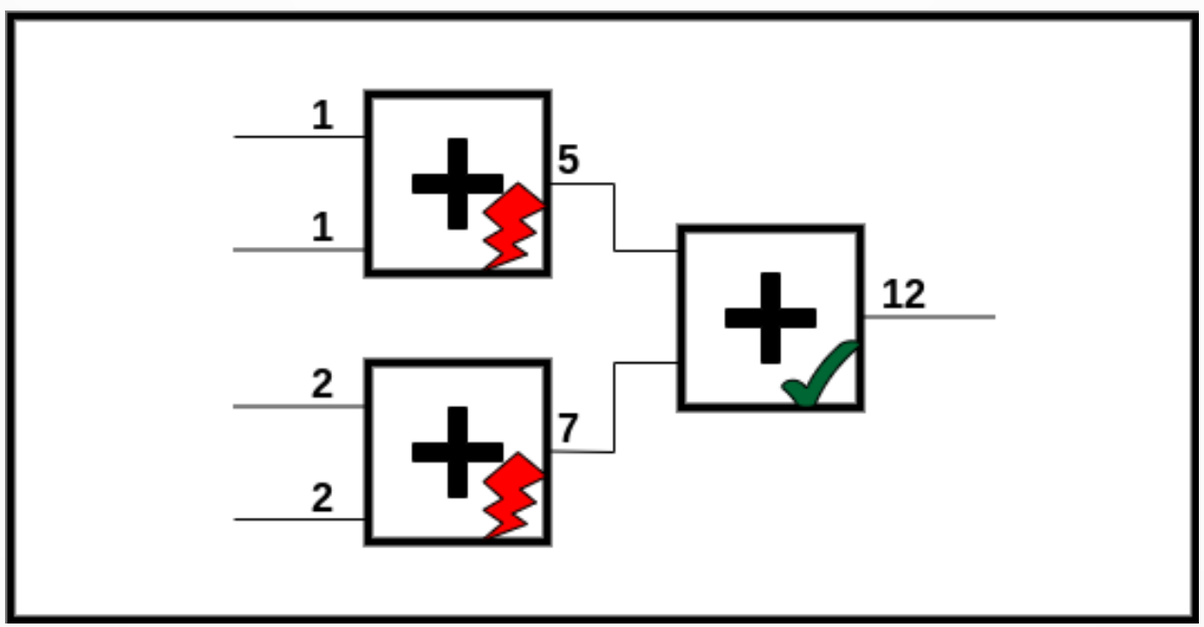
Fault sim, *Error* sim, *Failure* não

- Veja que apesar de erros terem se manifestado nos somadores falhos, o resultado final é correto!



Fault-Error-Failure: sim, sim, sim

- Agora temos o ciclo completo: as falhas se manifestaram em erros que se propagaram para a saída final do sistema (*failure! colapso!)*



Modelos de Falhas

- Exatamente como um processo falha?
  - Vamos também chamar processos de nodos aqui
- Há vários modelos, alguns clássicos:
  - Por parada (Crash)
    - Modelo de Falhas Crash
    - Crash-Recovery
    - Fail-Stop
  - Omissão
  - Bizantino
  - Temporização

Modelo de Falhas Crash

- O componente simplesmente pára de funcionar (acento incorreto no português moderno ;-)
- Em português: *crash* → falha por parada
  - Informalmente: verbo *crasheou, crasheareis...* :)
- Além de não reagir a qualquer estímulo → não produz qualquer saída para nenhuma entrada...
- ... o componente perde seu estado interno completamente

Modelo Crash-Recovery

- No modelo “crash” simples: um componente falho pode recuperar e retornar ao sistama...
- ... mas sem qualquer informação sobre seu estado antes da falha
- No modelo *crash-recovery:* o componente mantém informações chave em memória não-volátil
- Lembre-se: modelo *crash-recovery* implica em memória secudária

Modelo Fail-Stop

- Este é um modelo **muito** usado na construção de sistemas distribuídos tolerantes a falhas práticos
- As falhas dos nodos (componentes) são *crash* e todos os nodos sem-falha “sabem” quais nodos estão falhos
- Pode-se dizer que o modelo *fail-stop* inclui o sistema de monitoramento de falhas
- O modelo *fail-stop p*ode ser considerado um *baseline*: com este modelo podemos focar no algoritmo
- Depois, com mais maturidade no domínio do problema, pode-se considerar premissas mais complexas

Modelo de Falhas por Omissão

- Um nodo que falha por omissão não produz todas as mensagens que deveria
- Recebimento & transmissão
- Um subconjunto das mensagens é omitido

Modelo de Falhas Bizantina

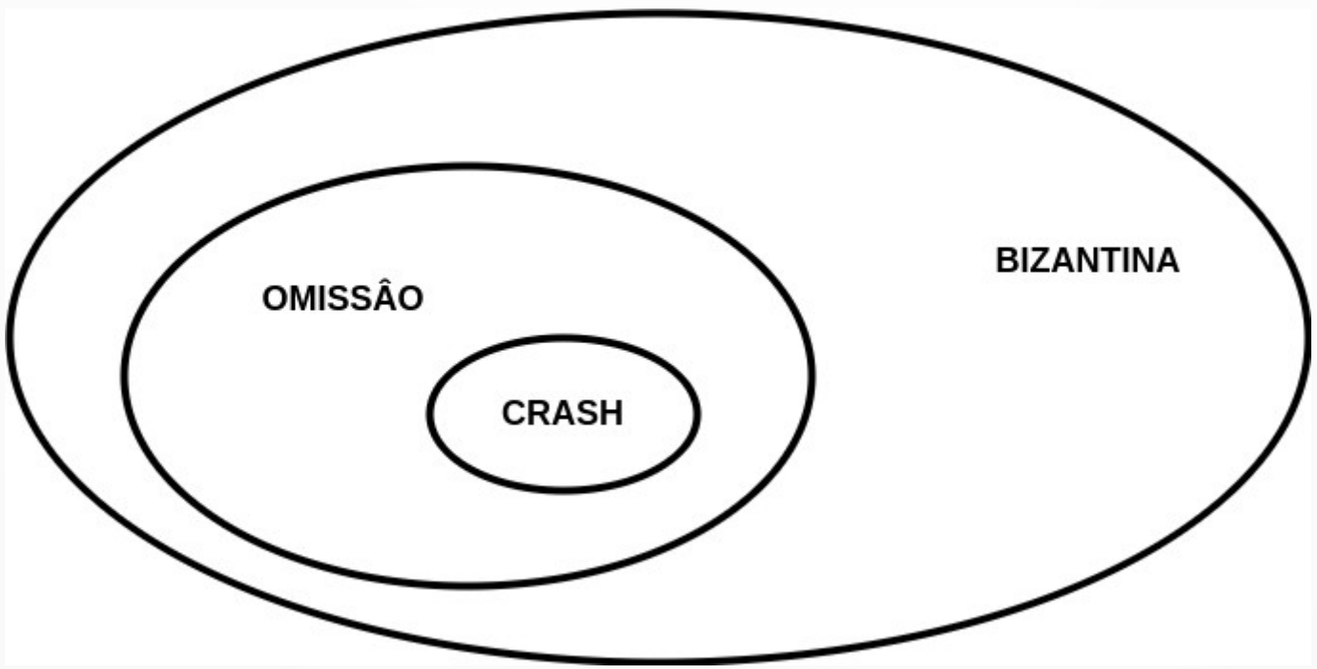
- Falhas arbitrárias
- Um sistema invadido: comportamento arbitrário, portanto corresponde à falha bizantina
- Sistemas Tolerantes a Intrusão
- Em inglês com maiúscula: *Byzantine*
  - *Um pouco da história...*

Modelo de Falhas de Temporização

- Atenção: só faz sentido em sistemas síncronos
- A transmissão de uma mensagem demora mais do que deveria
- A execução de uma tarefa demora mais do que deveria

Os 3 Principais Modelos

- Crash, Omissão e Bizantino: os 3 principais modelos

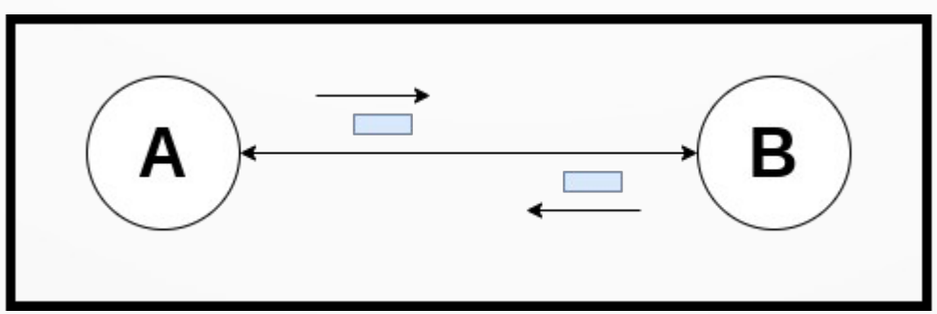


Bizantina → Omissão → Crash

- As falhas bizantinas correspondem a qualquer comportamento fora da especificação
- Inclui portanto as falhas por omissão, em que processos não transmitem todas as mensagens que deveriam
- Omissão inclui portanto crash, caso em que o conjunto de mensagens omitidas corresponde a *todas* as mensagens que deveria transmitir

Um Exemplo da Importância do Modelo

- Considere o Problema da Coordenação, especificado da seguinte maneira
  - Dois processos A e B conectados por canal de comunicação, através do qual se comunicam
  - Os processos A e B *nunca* falham
  - O canal de comunicação perde mensagens ocasionalmente



O Problema da Coordenação

- Defina um algoritmo distribuído a ser executado pelos processos A e B que garanta que, ao término da execução, dadas duas ações α e ß:
  - Ou ambos os processos executam a mesma ação, que é ou α ou ß;
  - Ou nenhum dos dois processos executa nenhuma ação.

Impossível!

- Prova de impossibilidade
- Considere que sim, há [várias] soluções para este problema
- Todas estas soluções envolvem rodadas de trocas de mensagens entre os processos A e B
- Conte o número de mensagens que cada solução necessita
- Escolha agora a solução que precisa do menor número de mensagens

A Solução com o Menor Número de Mensagens

- Sem perda de generalidade considere que a última mensagem é transmitida de A para B
- Ora, o algoritmo não pode depender desta mensagem, pois ela pode se perder!
- Lembre-se: o canal de comunicação pode perder mensagens
- Ou seja: a solução funciona *sem a última mensagem*

A Solução com o Menor Número de Mensagens: Ops!

- Agora temos um problema: escolhemos a solução com o menor número de mensagens
- Mas mostramos que há uma solução com número ainda menor de mensagens!
- ABSURDO!
- Impossível portanto resolver o problema

Seria este um problema puramente teórico?

- **Não!!** Ele ocorre todos os dias, várias vezes por dia, nos sistemas que nós usamos!
- Encerramento da conexão TCP envolve justamente este problema
- Absolutamente prático, vivemos este problema no nosso dia-a-dia

Conclusão

- Hoje tivemos nossa aula de Tolerância a Falhas
- Vimos vários conceitos importantes
  - *Dependability* e seus atributos
  - *Fault-Error-Failure*
  - Modelos de Falha: Crash, Omissão, Bizantina
  - A impossibilidade da coordenação de dois processos sobre um canal de comunicação que perde mensagens [ocasionalmente]

Obrigado!
Página da Disciplina
Sistemas Distribuídos
www.inf.ufpr.br/elias