Capítulo 2

Tolerância a Falhas *ou* Dependabilidade

Elias P. Duarte Jr.

versão do dia 25 de outubro de 2022

Lamento te informar que um dia seu computador vai falhar. Quando exatamente eu não sei dizer, mas que vai falhar, isso vai. É o que acontece com *todos* os componentes e sistemas computacionais: se você der tempo suficiente, a probabilidade que o funcionamento vai deixar de ser correto é 100%. Veja que estamos falando de qualquer tipo de indisponibilidade, que muitas vezes é decorrência de situação imprevista. O notebook não liga mais. O serviço de *streaming* fica inacessível. Ou a bateria do celular acaba quando você mais precisa. Estas situações impedem que alguma tarefa necessária seja executada. O sistema fica indisponível para você usar.

A área de tolerância a falhas desenvolve técnicas para permitir que os sistemas continuem funcionando, mesmo que alguns de seus componentes falhem. Na primeira seção deste capítulo vamos justamente compreender o que é um "sistema tolerante a falhas". Em seguida vamos definir o que é uma "falha", afinal o que é exatamente tolerado? Os atributos que permitem afirmar que um sistema é tolerante a falhas e as métricas para avaliar sua robustez são vistos na sequência. Chegamos então ao contexto mais específicio de sistemas distribuídos e vamos ver que diferentes tipos de falhas que afetam estes sistemas, e então sua classifição nos modelos de falhas mais representativos.

2.1 Sistemas Tolerantes a Falhas

As falhas e indisponibilidades são realmente intrínsecas aos sistemas computacionais. O que fazer? Na verdade é possível construir o sistema de forma que ele continue funcionando como um todo, mesmo que alguns de seus componentes falhem. Chamamos um sistema construído desta forma de *tolerante a falhas.* A Figura 2.1 ilustra um sistema tolerante a falhas. A ideia é que o sistema continue produzindo as saídas para as quais foi construído, mesmo que alguns de seus componentes não estejam funcionando como deveriam. Na verdade, a definição mais aceita de tolerância a falhas vai um pouco além, e diz ainda que o *desempenho* do sistema com componentes

falhos deve permanecer em níveis aceitáveis. É claro que para um sistema continuar funcionando, nem todos os seus componentes podem estar falhos, pelo menos alguns devem estar funcionando corretamente e devem, de alguma maneira, fazer a função dos componentes falhos.

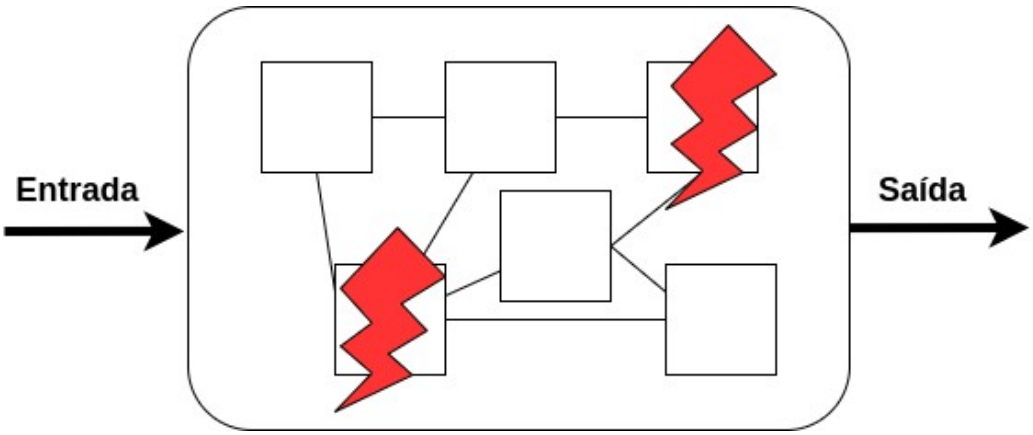

FIGURA 2.1: Um sistema tolerante a falhas continua oferecendo o serviço para o qual foi construído, mesmo que alguns de seus componentes estejam falhos.

Apesar de que para todos nós individualmente é muito bom ter sistemas que continuem funcionando corretamente mesmo após a ocorrência de falhas, em alguns casos esta é uma condição *imprescindível*. Naqueles casos, o sistema não pode sequer existir se não houver garantias matemáticas de sua capacidade de tolerar falhas. Exemplos incluem os sistemas que controlam aeronaves: imagine por exemplo um avião em pleno voo com centenas de passageiros a bordo -- o sistema computacional que controla a aeronave deve ter garantias explícitas que continua a funcionar perfeitamente mesmo após falhas de seus componentes. Em outros casos, menos dramáticos, a tolerância a falhas é também essencial. Por exemplo, a indispobilidade do sistema computacional de um banco pode causar prejuízos gigantescos. Na verdade, já há algumas décadas alguém disse uma frase muito curiosa:

As organizações têm se tornado indistiguíveis dos seus sistemas computacionais.

Uma consequência deste fato é que uma falha do sistema computacional representa, na verdade, uma falha da própria organização. Por exemplo, considere que você foi ao banco para fazer um saque, ou uma aplicação, ou uma transferência. Agora, se o sistema computacional do banco estiver indisponível (um funcionário pode até te dizer: "a rede caiu") na verdade o que está indisponível para você é o próprio banco (não a rede do banco!): vc não consegue fazer depósito, saque, nem transferência -- o banco não está funcionando. Desta forma, o correto funcionamento do sistema computacional é essencial para o bom funcionamento próprio "negócio".

Para que um sistema seja tolerante a falhas deve haver alguma redundância. Desta forma, se um componente falha, há uma alternativa correta para que a sua funcionalidade seja concretizada e o sistema como um todo pode continuar oferecendo o serviço para o qual foi construído. A redundância pode ser tanto explícita, por exemplo você pode comprar um notebook com duas unidades de memória secundária, de forma que se apenas uma falhar e ambas mantiverem os mesmos dados, o uso do computador não é afetado. A redundância pode também ser implícita. Por exemplo, se uma rota de uma rede não funciona mais, pode ser que haja na rede uma rota alternativa, que pode ser usada para substituir a rota falha. É inclusive comum que as redes sejam construídas para ter rotas alternativas.

No caso da redundância explícita, uma técnica comum utilizada em sistemas tolerantes a falhas, incluindo os sistemas aéreos mencionados acima, é a chamada redundância tripla modular (*triple modular redundancy*). Este tipo de redundância permite tornar tolerante a falhas qualquer módulo de qualquer sistema, conforme ilustrado na Figura 2.2. A parte A da figura mostra um módulo qualquer do sistema, sua entrada e saída. Na parte B este módulo tem a mesma entrada e saída, mas internamente é possível ver que a lógica foi triplicada: são três implementações distintas do mesmo módulo ("versões" V1, V2 e V3) e seus resultados são a entrada de um "votador" que produz como saída a maioria das respostas. Assim se um dos módulos estiver falho, a maioria ainda vai produzir a saída correta, que é produzida como resultado final da execução do módulo.

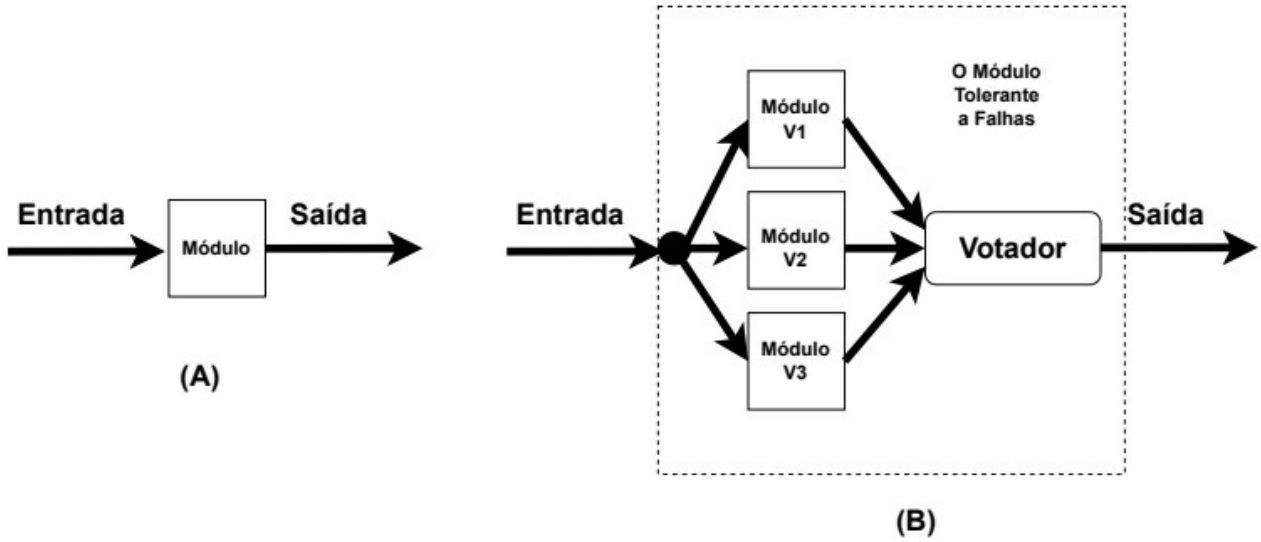

FIGURA 2.2: A técnica de redundância modular tripla (*three-modular redundancy).* O mesmo módulo mostrado em **(A)** pode se tornar tolerante a falhas como mostrado em **(B)**.

No caso dos sistemas distribuídos a redundância vem de graça! Eles são implicitamente redundantes, por definição mesmo, pois um sistema distribuído consiste de um conjunto de múltiplos processos, no mínimo dois, mas o número é geralmente ainda maior. Considerando que processos podem frequente e facilmente sofrer falhas, os sistemas distribuídos são quase sempre

construídos para serem tolerantes a falhas. Quando se fala em sistemas distribuídos, sempre se fala em como toleram falhas. Mas antes de explorarmos este caminho, vamos definir com cuidado o jargão da área de tolerância a falhas, que nem sempre é intuitivo, mas é essencial para trabalhar neste contexto.

2.2 Afinal, o que é uma "falha"?

Antes de mais nada, precisamos definir exatamente o que é uma "falha". Poucos termos são usados na computação com tantos significados tão distintos. É possível generalizar, dizendo que uma falha corresponde a um comportamento incorreto, fora da especificação. Desta forma, o sistema ou não produz resultado algum, ou produz resultado que não está entre os permitidos ou desejáveis. No jargão da área de tolerância a falhas são identificados três "estágios" diferentes referentes à falha de um componente de um sistema, em inglês eles são: *fault, error* e *failure*. A seguir vamos definir cada um destes termos e ver as opções para sua tradução para o português.

Tudo começa com um *defeito* de um componente do sistema. Pode ser um defeito de hardware ou um *bug* de software. Um componente pode ter defeito, mas este não ser "ativado". Por exemplo, pode ser que um procedimento específico de um programa tenha sido implementado incorretamente, mas se aquele procedimento não é executado, então o defeito fica lá, dormente. O termo *fault* se refere justamente a este tipo de defeito. Em português, a comunidade de Engenharia de Software que trabalha com testes usa justamente a palavra *defeito*. As comunidades de tolerância a falhas, tanto do Brasil como de Portugal, adotam duas palavras distintas para *fault* em português: alguns traduzem como "falta", outros como "falha" mesmo. Veja que em inglês "tolerância a falhas" é "*fault tolerance*", ou seja, o que se tolera é uma *fault*, que neste contexto podemos traduzir para "falha". No livro usamos "defeito" ou "falha" em português, ou mesmo "*fault*" em inglês, quando for necessário remover ambiguidade. No fim das contas, o que importa é compreender que *fault* corresponde um defeito latente, intrínseco de um componente, que pode ou não se manifestar.

Quando o defeito (*fault*) se manifesta, ocorre o chamado erro (*error*). Neste caso, em português todos concordam em traduzir como "erro". Voltando ao exemplo acima, se o procedimento implementado incorretamente é executado e produz uma saída fora da sua especificação, então há um erro, a manifestão do defeito (*fault*). Veja que mesmo que o procedimento com defeito seja executado é possível que as saídas produzidas sejam (por alguma coincidência) corretas, neste caso não há erro. Um exemplo bastante impressionante de defeito e erro é o que aconteceu com alguns modelos do processador Pentium da Intel, no começo dos anos 1990 ["Pentium FDIV Bug", Wikipedia]. Os processadores Pentium da Intel já dominavam o mercado na década de 1990. Em 1993 a empresa lançou o Pentium P5, em diversas versões, com grande impacto mundial. Tudo ia bem, até que em 1994 começaram a surgir notícias de que o

processador tinha um *bug*. Apesar de ser capaz de executar a maioria das instruções com perfeição, algumas divisões de números em ponto flutuante vinham com erro. Pela natureza do problema, o defeito ficou conhecido como FDIV (*Floating -point DIVision*) *bug*. Na verdade o defeito se manisfestava em poucos casos. De acordo com uma estimativa da época, apenas 1 de cada 9 bilhões de operações de divisão de ponto flutuante resultariam incorretas. Mas era suficiente para inviabilizar a comercialização do microprocessador. Em 1994, a Intel acabou tendo que fazer -- pela primeira na história da microeletrônica -- um recall dos produtos (algumas versões do Pentium P5 800 e Pentium P54C 600) com prejuízo estimado de cerca de meio bilhão de dólares.

Voltando à nomenclatura de Tolerância a Falhas, um defeito (*fault*) pode se manifestar como um erro (*error*) e, por sua vez, o erro produzido pelo componente com defeito pode se propagar para a saída geral do sistema, causando então sua falha (*failure*). A falha (*failure*) do sistema corresponde ao seu colapso: o sistema como um todo produz resultado incorreto, fora da sua especificação. Veja que é possível que um erro de um componente seja neutralizado, e não se propague até a saída final do sistema. Por exemplo, se a saída de um módulo arimético tem defeito (*fault*) que se manifesta em erro (*error*) na saída, mas esta é então multiplicada por zero, o erro é, por assim dizer, neutralizado e não vai se propagar até a saída final do sistema.

Para reforçar a compreensão destes três conceitos de *fault* (defeito), *error* (erro) e *failure* (falha), vamos ver estes conceitos ilustrados em um sistema como o exemplo da Figura 2.3. Este sistema consiste de três somadores de inteiros. Cada um deles é um módulo que recebe como entrada dois inteiros e produz como saída sua soma. O sistema como um todo recebe dois pares de inteiros como entrada, que são somados por dois somadores cujas saídas são a entrada do terceiro somador, que produz a saída final do sistema: a soma dos quatro inteiros da entrada original.

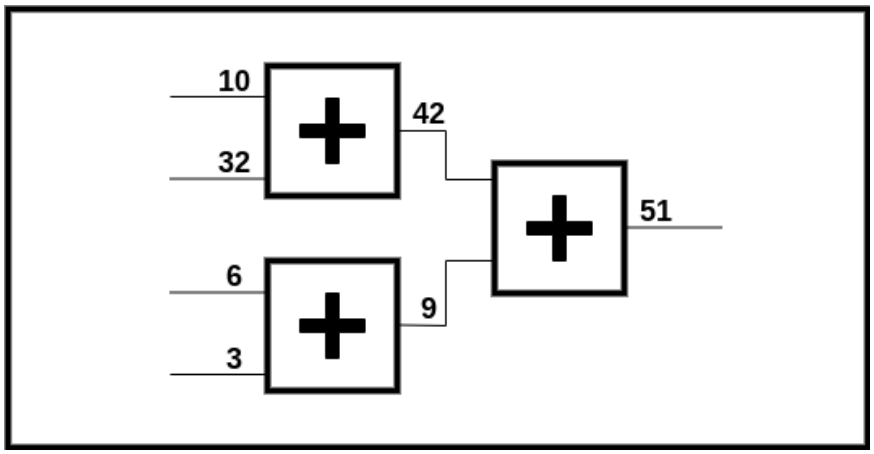

FIGURA 2.3: Exemplo de sistema para ilustrar os conceitos de *fault* (defeito ou falta ou falha), *error* (erro) e *failure* (falha).

Nas Figuras 2.4 a 2.7 os dois módulos que recebem os parâmetros de entrada estão ambos com defeito. O módulo que recebe X e Y como parâmetros sempre produz resultado igual a 5. O módulo que recebe P e Q como parâmetro sempre produz resultado igual a 7. Na Figura 2.5, por coincidência, X = 2, Y = 3, P = 1 e Q = 6, assim as saídas dos módulos são corretas, não há erro, apesar de que os defeitos estão lá. Na Figura 2.6 os valores dos pares de parâmetros são invertidos, o que causa a manifestação de erros. Entretanto, o resultado final do sistema é perfeitamente correto para a entrada, ou seja não há falha (*failure*) do sistema. Por fim, no exemplo da Figura 2.7, os defeitos se manifestam em erros dos módulos que causam a falha (*failure*) do sistema.

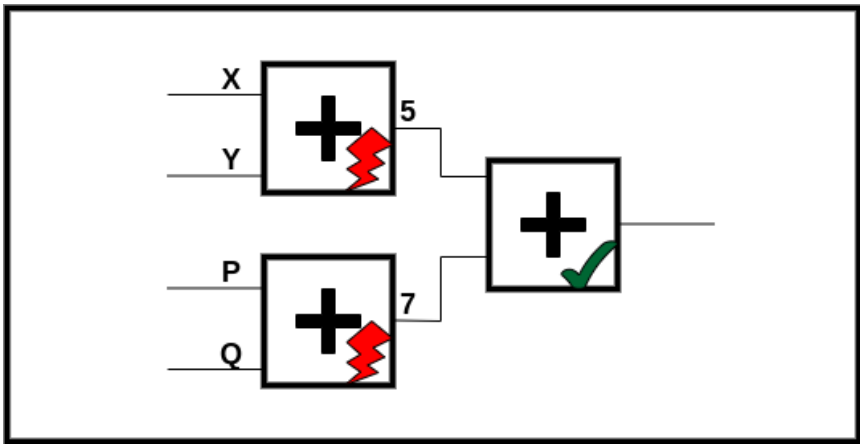

FIGURA 2.4: No exemplo há dois componentes com defeito (*fault*): independente dos pares de inteiros que recebem como entrada os somadores produzem *sempre* 5 e 7 como suas saídas.

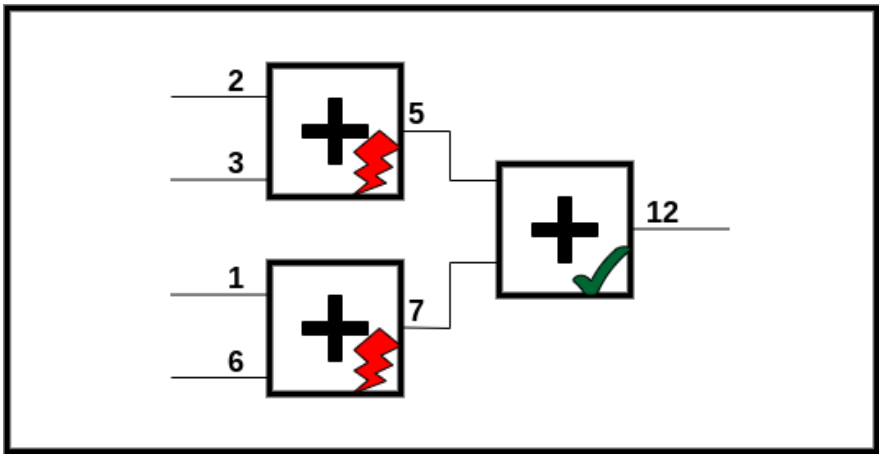

FIGURA 2.5: No exemplo há dois componentes com defeito (*fault*) mas estes não se manifestam como erro (*error*): as saídas produzidas por todos os somadores são corretas.

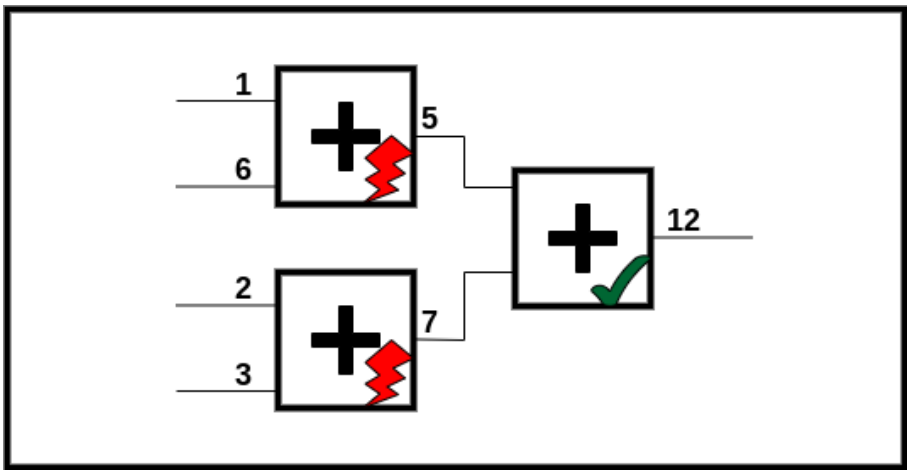

FIGURA 2.6: No exemplo há um componente com defeito (*fault*) que se manifesta como erro (*error*) mas não se propaga para a saída final do sistema: não há *failure*.

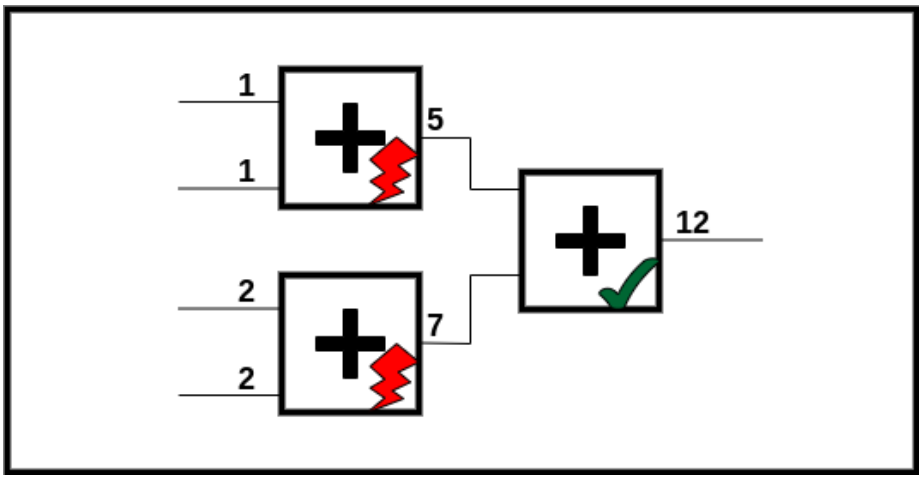

FIGURA 2.7: No exemplo há um componente com defeito (*fault*) que se manifesta como erro (*error*) e se propaga para a saída final do sistema: há *fault* (defeito), *error* (erro) e *failure* (falha).

2.3 *Dependability*: Confiança no Funcionamento do Sistema

Pode causar surpresa o fato de que a comunidade acadêmica não define o termo "tolerância a falhas" ou, em inglês, "*fault tolerance*" no jargão da área [2.1]. O termo que se usa é *dependability*, que em português tem sido traduzido como "confiança no funcionamento de um sistema". A corruptela para o português "dependabilidade" tem sido cada vez mais usada. Neste capítulo vamos optar pelo intuitivo e usar o termo "tolerância a falhas" no exato sentido de confiança no funcionamento de um sistema (dependabilidade). O termo "tolerância a falhas" é compreendido intuitivamente pelas pessoas e faz parte do jargão da Ciência da Computação no Brasil.

A *dependabilidade* não é uma coisa só, ela engloba diferentes atributos. De acordo com o com artigo seminal que define os principais termos da área [*Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE TDSC, Vol. 1, No. 1], estes atributos são:

- **Disponibilidade (*Availability)*** - capacidade do sistema de estar disponível para uso, levando em conta inclusive a ocorrência de falhas que interrompam o serviço e a posterior recuperação com a retomada do serviço;
- **Confiabilidade (*Reliability)*** - capacidade do sistema de oferecer o serviço correto continuamente, sem falhas e interrupções;
- **Segurança-*Safety* (*Safety*)** - capacidade do sistema de evitar consequências catastróficas;
- **Segurança (*Security*)**- inclui os diversos atributos clássicos de segurança, como capacidade de oferecer confidencialidade e integridade;
- **Manutenibilidade (*Maintainability)*** - facilidade de manutenção e realizar alterações.

Todos os atributos da *dependability* têm o objetivo concreto de expressar o comportamento do sistema tendo em vista que falhas podem ocorrer. Dois dos atributos mais relevantes são *confiabilidade* e *disponibilidade*, que vamos trabalhar a seguir. Antes, alguns comentários sobre os outros atributos. A manutenabilidade é auto-explicativa: quão fácil é dar manutenção no sistema? *Bugs* são uma fonte importante de falhas e indisponibilidade e, a após a sua descoberta, é importante corrigir o sistema adequadamente. A *safety* se refere àqueles sistemas que só podem existir se forem comprovadamente, matematicamente tolerantes a falhas. Exemplos incluem sistemas de aeronaves ou usinas nucleares. A falha daqueles sistemas leva a consequências catastróficas, incluindo a perda de vidas humanas ou danos graves ao meio-ambiente.

Um outro atributo importante é a segurança (*security*), que é uma grande área da Ciência da Computação em si própria. A segurança também consiste de um conjunto de atributos. Curiosamente, um deles é justamente a disponibilidade (*availability*). A interseção entre as duas áreas - dependabilidade e segurança - é grande. A segurança, enquanto atributo de tolerância a falhas, incluir as propriedades clássicas, como confidencialidade, integridade, autenticidade entre outras. A confidencialidade se refere à capacidade do sistema de manter e comunicar informações de forma sigilosa, cujo conteúdo só é acessado por partes autorizadas. A integridade se refere à capacidade do sistema de evitar que os dados sejam alterados sem autorização. A autenticidade se refere à capacidade do sistema garantir a identidade de todas as partes, através de mecanismos de autenticação.

A disponibilidade (*availability*) de um sistema computacional reflete a probabilidade dele estar disponível quando você precisa dele. Trata-se de uma medida quantitativa, numérica. Para calcular a disponibilidade, o primeiro passo é observar o sistema. Esta observação pode ser feita

pelo período de tempo que você definir, que vamos chamar de *T_observação*. Por exemplo, *T_observação* pode ser 1 dia, 1 semana, 1 mês ou 1 ano ou qualquer outro intervalo de tempo que você definir. É necessário monitorar o funcionamento do sistema com cuidado, para não perder de vista situações que vão influenciar nos resultados. Durante este período de observação você vai marcar a data e hora exatas em que o sistema tenha ficado indisponível. É necessário acompanhar até o sistema voltar a ficar disponível, e marcar a data/hora exatas em que isso aconteceu. Passado o período de observação, você soma todos os intervalos de tempo em que o sistema ficou indisponível, subtrai do tempo total e obtém a quantidade de tempo em que ficou disponível, vamos chamar de *T_disponível*. Pronto, já podemos calcular a disponibilidade:

Disponibilidade (*Availability*) = *T_disponível/T_observação*

Por exemplo, considere que você ficou observando o sistema por 1 ano durante o qual o sistema esteve indisponível por 5 minutos. Convertendo um ano (não bissexto) para minutos: *T_observação* = 565 x 24 x 60 = 525.600 minutos. E considere que o sistema ficou indisponível por 5 minutos durante o ano temos que *T_disponível* = 525.595. A disponibilidade (availability) do sistema com 6 casas decimais é igual a:

Disponibilidade = *Availability* = 525.595/525.600 = 0,99999 = 99,999%

Como você pode observar esta disponibilidade é de 0,99999: são 5 noves depois da vírgula. E assim ela é chamada, "*five nines*". Se fossem seis noves, seria "*six nines*" e assim por diante. A disponibilidade de "cinco noves" é aquela esperada, por exemplo, das redes de telecomunicações comerciais. Neste caso são permitidos, no máximo, pouco mais de 5 minutos de indisponibilidade por ano (5,646 minutos para ser preciso).

Na verdade este exemplo que fizemos não leva em conta um ponto importante do cálculo da disponibilidade. O que se deseja é a *probabilidade* do sistema estar disponível quando você precisa dele. Assim, o cálculo da disponibilidade não pode ser feito usando uma única observação. Devem ser feitas múltiplas observações, até que se tenha uma média representativa do *T_disponível* dado o *T_observação*. A divisão então é feita utilizando a média calculada para o tempo disponível, dado o tempo de observação.

O outro atributo da dependabilidade que vamos calcular é a confiabilidade (*reliability*). A confiabilidade é semelhante à disponibilidade, mas com uma diferença importante. O que se mede é a probabilidade do serviço ser oferecido continuamente, sem qualquer falha que possa interrompê-lo. Uma outra forma de expressar o problema é: a confiabilidade de um sistema é a probabilidade

dele estar funcionando em um determinado instante de tempo. Para calcular a confiabilidade do sistema é necessário mais uma vez fazer o monitoramento de falhas e recuperações durante um intervalo *T_observação*. O primeiro ponto a calcular é: quantas falhas aconteceram durante este tempo? Vamos chamar o número de falhas de *Número_falhas*. Agora já podemos calcular um parâmetro que expressa a confiabilidade, chamado de MTBF (*Mean Time Between Failures*):

$$MTBF = T\_observação/Número\_falhas$$

Por exemplo, considere um sistema hipotético que foi observado por 10 minutos no qual aconteceram 5 falhas. Neste caso, o MTBF = 10/5 = 2 minutos. O MTBF é, portanto, uma medida da confiabilidade (*reliability*): espera-se que o sistema permaneça em média 2 minutos sem sofrer falhas. Mais uma vez vale lembrar que estas métricas se referem a probabilidades, assim os dados obtidos de 1 ou poucas amostras são insuficientes para serem considerados representativos.

2.4 Modelos de Falhas em Sistemas Distribuídos

Os sistemas computacionais sofrem falhas dos mais diversos tipos. Estes diferentes tipos vão ser definidos como os modelos de falhas descritos a seguir. Quando se constrói um sistema tolerante a falhas é essencial definir claramente qual modelo de falha é considerado -- ou seja, que tipo de falha o sistema tolera. Os modelos de falhas que vamos considerar a seguir são os mais comuns em sistemas distribuídos.

O tipo mais básico de falha é aquele conhecido como *por parada* ou, em inglês, *crash*. Um processo que sofre uma falha crash não produz qualquer resultado a nenhum estímulo. Não vamos sequer usar itálico para a palavra crash, que já foi virtualmente incorporda ao português, inclusive no verbo: crashou, crashado,... O processo que sofreu falha crash literalmente morreu. Uma das consequências é a perda do estado interno. Desta forma, se o processo recupera mais tarde, é como um processo inteiramente novo, que não tem qualquer informação sobre execuções anteriores.

Existem duas variações sobre o modelo de falhas crash que precisamos compreender com precisão. São os modelos *crash-recovery* e *fail-stop*, descritos a seguir. Quando você fala do modelo crash puro e simples, na verdade *não* está prescevendo a recuperação dos processos. No modelo crash, sim: processos podem recuperar (*recover*). O ponto é que eles perdem seu estado interno completamente e quando recuperam, se entram no sistema são como um *novo* processo que se une ao sistema em execução. Em geral, este tipo de situação é tratada nos sistemas de composição dinâmica pelo uso da primitiva *join()*. Desta forma, em sistemas estáticos, os processos no modelo crash que se recuperam em geral não fazem mais parte da composição do sistema.

No modelo *crash-recovery* a situação é diferente: os processos necessariamente têm memória secundária. Itens essenciais de informação do estado interno do processo são armazenados nesta memória não-volátil. Assim, quando o processo se recupera, ele consegue "naturalmente" voltar a executar o sistema. Vale a pena guardar o seguinte fato: o modelo *crash-recovery* implica no uso de memória secundária.

O modelo *fail-stop* é ainda outra variação muito utilizada para a construção de algoritmos e sistemas distribuídos. Este modelo precisa, na verdade, de mais que processos em estado falho ou correto. Quando um processo falha (por parada, isto é, falha crash) assume-se que todos os processos corretos ficam sabendo do evento. Como? Boa pergunta. De alguma forma -- não contemplada pelo modelo, que necessita evidentemente de monitoramento contínuo -- os processos corretos mantêm sempre a lista completa dos processos falhos.

Depois do modelo de falhas crash, vamos definir as falhas por *omissão.* Neste modelo, os processos não produzem o conjunto completo de mensagens que deveriam. Assim, em resposta a um estímulo, o processo produz como resposta um conjunto de mensagens que pode ser menor que aquele esperado tendo em vista sua especificação.

Por fim, chegamos ao modelo de falhas mais genérico: os falhas bizantinas. Neste caso, o comportamento do processo é arbitrário. Assim, um processo que sofre uma falha bizantina pode produzir qualquer saída para qualquer entrada. Justamente por esta liberdade de possibilidades, este modelo é aplicado para sistemas que sofrem intrusão: seu comportamento é imprevisível!

Existe uma variação do modelo de falhas bizantinas que é bastante importante, *as falhas por computação incorreta*. Neste tipo de falha, o *tipo* (sintaxe) do resultado é o mesmo esperado, mas o seu valor é diferente daquele que a especificação demanda de uma execução correta.

Existe ainda um modelo que aparece ocasionalmente, é o de falhas de *temporização*. Este modelo não pode ser aplicado, naturalmente, aos sistemas assíncronos. Uma falha de temporização implica que alguma ação tomada pelo sistema aconteceu mais cedo ou mais tarde do que deveria.

Os três modelos mais importantes são os das falhas crash, omissão e bizantina. É comum representar os três tipos aninhados, como ilustra a Figura 2.9. As falhas bizantinas são arbitrárias, englobando qualquer comportamento fora da especificação, inclusive omissão e crash. O modelo de falhas por omissão engloba os das falhas crash, na medida em que o processo que sofre uma falha por omissão produz um subconjunto das mensagens esperada -- o conjunto vazio é um tal subconjunto. Portanto, a falha crash é um tipo especial de falha por omissão.

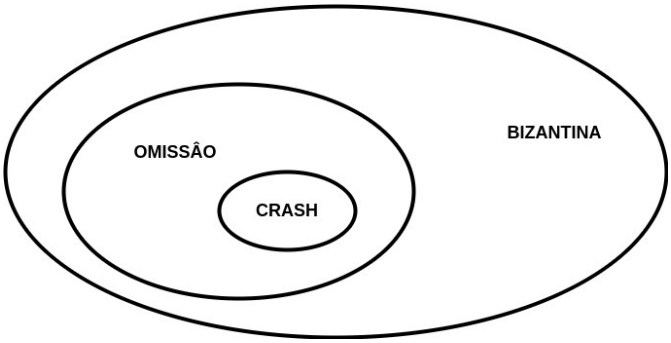

FIGURA 2.8: Falhas crash, por omissão em bizantina, um modelo “engloba” o outro.

Referências

Todos aqueles interessados na área de Tolerância a Falhas (*Dependability*) devem ler cuidadosamente o artigo seminal da área, que define a nomenclatura, publicado no Vol. 1, No. 1 da IEEE Transactions on Dependable and Secure Computing:

- A. Avizienis, J.-C. Laprie, B. Randell, C. 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.

Um livro clássico de Tolerância a Falhas é o do Pradham, que apesar de ter sido publicado em 1996, ainda pode ser considerado fonte de conteúdo importante, principalmente na discussão conceitual e histórica que traz sobre os diversos aspectos de Tolerância a Falhas:

- D. K. Pradhan (Editor), *Fault-Tolerant Computer System Design*, Prentice-Hall, 1996.

As Profas. Ingrid Jansch-Porto e Taisy Weber (UFRGS) publicaram já há algumas décadas artigos e apostilas com a nomenclatura da área em português, podemos destacar o minicurso do JAI a seguir:

- T. Weber, I. Jansch-Pôrto, R. Weber, "*Fundamentos de Tolerância a Falhas*", Minicurso Jornadas de Atualização em Informática (JAI), CSBC, 1990.

As falhas por computação incorreta foram originalmente propostas por Laranjeira, Malek e Jenevein:

- L. A. Laranjeira, M. Malek, R. M. Jenevein, "On Tolerating Faults in Naturally Redundant Algorithms", *The 10th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 118-127, 1991.

Este modelo é a base do diagnóstico baseado em comparações:

- Elias P. Duarte Jr., Roverli P. Ziwich, Luiz C. P. Albini, "A Survey of Comparison-Based System-Level Diagnosis," *ACM Computing Surveys*, Vol. 43, No. 3, pp. 1-56, April 2011.

O modelo de falhas fail-stop foi proposto por Schlichting e Schneider:

- R. D. Schlichting, F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," ACM Transactions on Computer Systems, Vol. 1, No. 3, pp. 222-238, 1983.