Universidade Federal de Santa Catarina — ine Departamento de Informática e Estatística

Escalonamento e Balanceamento de Carga em Ambientes Paralelos e Distribuídos

Prof. Mario Dantas

Ementa

- Introdução ao escalonamento e balanceamento de carga

- Escalonamento estático.

- Granularidade e particionamento de tarefas

Ementa

- Ferramentas de escalonamento
- Balanceamento de carga
- Mecanismos para migração de processos
- Índices de Carga

Bibliografia :

• *Scheduling and Load Balancing in Parallel and Distributed Systems*, Shirazi-Hurson, Kavi, IEEE Press, 1995.

•*Parallel Computing*: Theory and Practice, M. Quinn, McGraw-Hill, 1994.

Bibliografia :

• *High-Performance Cluster Computing* - Architectures and Systems, R. Buyya, Prentice-Hall, 1999.

• *Parallel Computer Architecture: A Hardware/ Software Approach*, D. Culler, Morgan Kaufmann, 1999.

Bibliografia :

- *The Grid Blueprint for a New Computing Infrastruture*, Ian Foster and Karl Kessemman, Morgan Kaufmann, 1999.

- *Grid Computing: Making The Global Infrastructure a Reality,* Fran Berman, Geoffrey Fox , Tony Hey , John Wiley & Sons, 2003, **ISBN** 0470853190.

Avaliação

Duas provas (P1 e P2) - pesos igual a 35 %

Trabalho (T) - peso igual a 30 %, será avaliado o conteúdo do material e o desempenho da apresentação do trabalho

Composição final da Menção - P1 + P2 + T

I - Programação Paralela

O emprego de paralelismo para resolver um problema pode ser feito em diferentes níveis.

Os níveis de paralelismo podem ser estabelecidos com base na quantidade de código que é passível de ser paralelizada.

Esta quantidade de código é conhecida como *grão*. O tamanho do *grão* define sua granularidade, que pode ser dividida em categorias como mostra a próxima tabela

Granularidade de Código e Níveis de Paralelismo

Granularidade	**Unidade de Código**	**Nível de Paralelismo**
Muito Fina	Instrução	Instrução
Fina	Loop/Bloco de Instrução	Dados
Média	Função	Controle
Grande	Processo ou Tarefa	Tarefa

Em uma mesma aplicação o paralelismo pode ser detectado em diferentes níveis, sendo possível explorar as abordagens aplicáveis a cada nível de forma complementar.

Dentro de um mesmo nível o tamanho do *grão* pode sofrer grande variação, pois as categorias de granularidade são apenas relativas.

O ajuste da performance de uma aplicação paralela é um problema que depende, entre outras coisas, de encontrar o tamanho de grão apropriado para a aplicação.

Em cada nível pode-se abordar o paralelismo de duas formas:

- uma automatizada, como é feito, a nível de instruções, nos microprocessadores superescalares;

- outra manual, como é feito, a nível de processos, ao empregar-se troca de mensagens.

Estas são as duas abordagens básicas de paralelismo:

Paralelismo implícito: a aplicação é paralelizada sem a intervenção do programador, que não especifica, e também não controla, a decomposição e o escalonamento das tarefas e a colocação dos dados.

Comumente suportado por hardware, por linguagens de programação paralelas e por compiladores paralelizadores;

- **Paralelismo explícito**: o programador é responsável pela maior parte do esforço de paralelização (decomposição e/ou escalonamento de tarefas, comunicação, etc.).

Essa abordagem se baseia na hipótese de que o programador é a pessoa que reúne as melhores condições para decidir como paralelizar sua aplicação.

Usualmente é dito que esta abordagem permite obter melhores resultados.

O desenvolvimento e a implementação de algoritmos paralelos, em uma arquitetura de memória distribuída, está normalmente associado ao emprego de paralelismo explícito.

1.1 - Projeto de Algoritmos Paralelos

Elaborar um algoritmo paralelo é uma atividade que pode ser desenvolvida de diferentes formas.

Foster, por exemplo, sugere uma metodologia de quatro passos para, a partir da especificação do problema, se obter um algoritmo paralelos que o resolva.

É importante frisar que, em grande parte dos casos, há mais de um algoritmo (ou pelo menos variações de um mesmo algoritmo) para resolver o problema, e que a melhor solução paralela pode ser bem diferente da correspondente seqüencial.

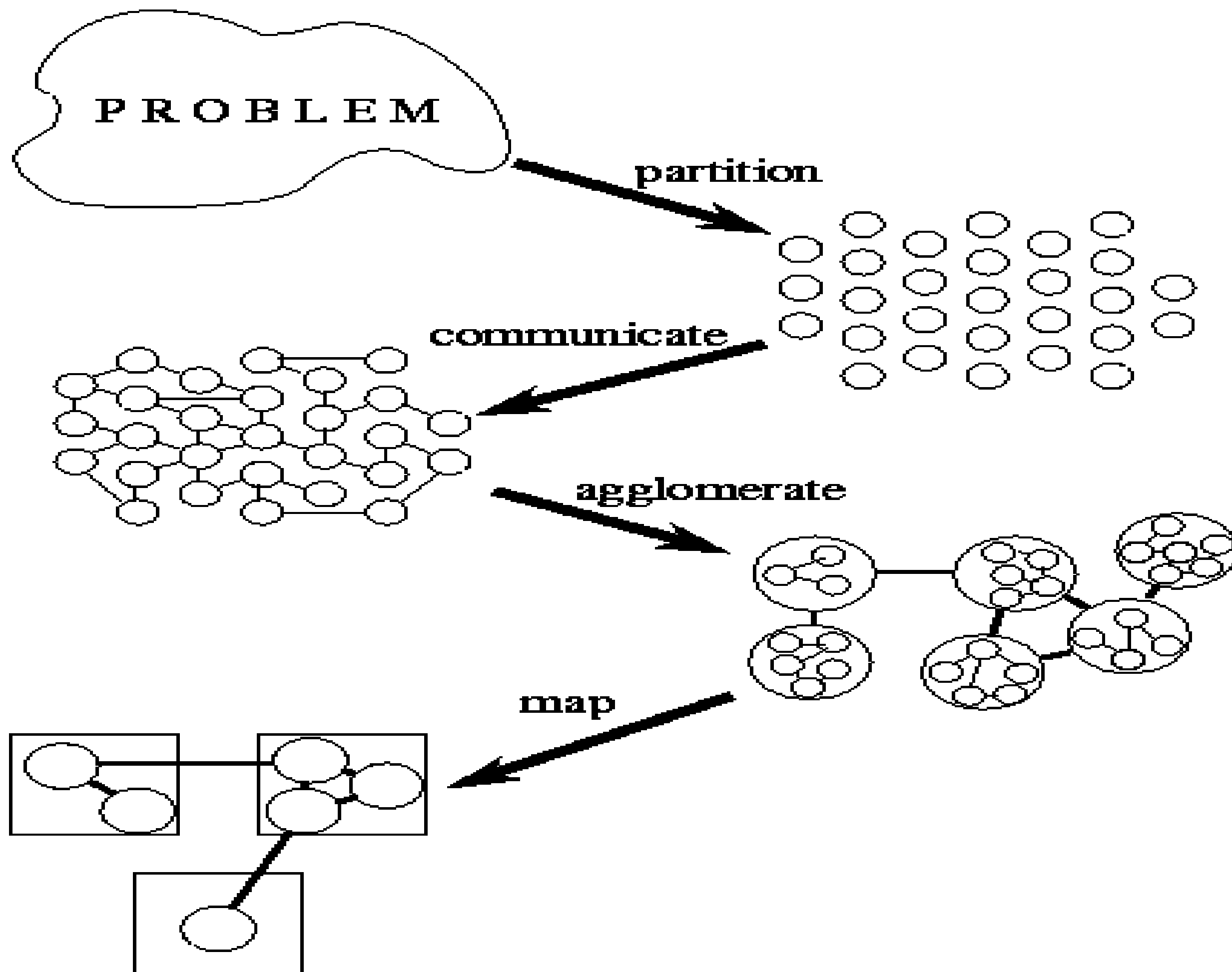
Os quatros passos para o projeto de algoritmos paralelos propostos por Foster são:

- particionamento;
- comunicação;
- aglomeração; e
- mapeamento.

Na próxima figura observamos a seqüência de passos sendo aplicada a um problema.

Nos itens a seguir serão abordados cada um dos passos.

Metodologia de Projeto de Programas Paralelos



1.1.1 - Particionamento (Partitioning)

O objetivo principal desta etapa é encontrar oportunidades de paralelização. Daí a principal atividade consiste em realizar a decomposição fina do problema, definindo as várias pequenas tarefas que o compõe.

São duas as técnicas principais empregadas com este fim :

- **Decomposição de Domínio (ou Dados):** nesta técnica, primeiro são particionados os dados associados ao problema, depois é feita a divisão do processamento, associando aos dados (divididos de forma disjunta) o conjunto de operações pertinentes.

Algumas operações poderão envolver dados de mais de uma tarefa, sendo que, neste caso, será necessário realizar comunicação entre as tarefas, a fim de estabelecer algum grau de sincronismo entre as mesmas;

Decomposição Funcional: neste caso, a ênfase é dada na divisão da computação envolvida no problema. Primeiro procura-se particionar o processamento em tarefas disjuntas, depois é feita a análise dos dados necessários a cada tarefa.

Caso os dados possam ser divididos em conjuntos disjuntos, a decomposição está completa. Caso contrário, pode-se tentar outro particionamento do processamento, replicação dos dados ou compartilhamento dos mesmos.

Caso a sobreposição de dados seja grande, pode ser mais indicado o emprego da decomposição de domínio.

1.1.2 - Comunicação (Communication)

O particionamento do problema é feito com a intenção de executar as tarefas definidas de forma concorrente e, se possível, de forma independente.

Contudo, a computação associada a uma tarefa tipicamente envolve dados associados a uma outra tarefa, o que requer o estabelecimento da comunicação entre as mesmas.

Esta fase tem por objetivo analisar o fluxo de informações e de coordenação entre as tarefas, definindo uma estrutura de comunicação.

A natureza do problema e o método de decomposição empregado serão determinantes na escolha do modelo de comunicação entre tarefas a ser utilizado.

Foster divide os modelos de comunicação em quatro categorias (ortogonais):

Local/Global: no modelo de comunicação local cada tarefa se comunica com um pequeno grupo de outras tarefas (suas "vizinhas"), enquanto que no modelo global as tarefas podem comunicar-se arbitrariamente;

- **Estruturado/Não estruturado:** no modelo estruturado as tarefas formam uma estrutura regular (e.g. árvore), já no modelo não estruturado elas formam grafos arbitrários;

- **Estático/Dinâmico:** no modelo estático a comunicação se dá sempre entre as mesmas tarefas, enquanto que no modelo dinâmico a comunicação não possui parceiros definidos, dependendo dos valores calculados em tempo de execução;

- **Síncrono/Assíncrono:** há, no modelo síncrono, uma coordenação entre as tarefas comunicantes, enquanto no modelo assíncrono não existe coordenação na comunicação.

1.1.3- Aglomeração (Agglomeration)

Neste passo, as tarefas e a estrutura de comunicação definida nos passos anteriores são avaliados em termos de custo de implementação e requisitos de performance. Se for necessário tarefas podem ser combinadas, ou aglomeradas, em tarefas maiores a fim de reduzir o custo de implementação ou aumentar a performance.

Pelo mesmo motivo pode-se efetuar a replicação de dados e/ou processamento.

Três objetivos, algumas vezes conflitantes, devem nortear as decisões por aglomeração e/ou replicação:

Aumento da Granularidade: redução dos custos de comunicação pelo aumento da granularidade do processamento e da comunicação;

Preservação da Flexibilidade: a granularidade deve ser controlada por um parâmetro em tempo de execução ou de compilação, a fim de que o número de tarefas possa ser adaptado ao número de elementos de processamento;

Redução dos custos de engenharia de software: deve se buscar o aproveitamento de código, sempre que possível, e a compatibilização com módulos já existentes.

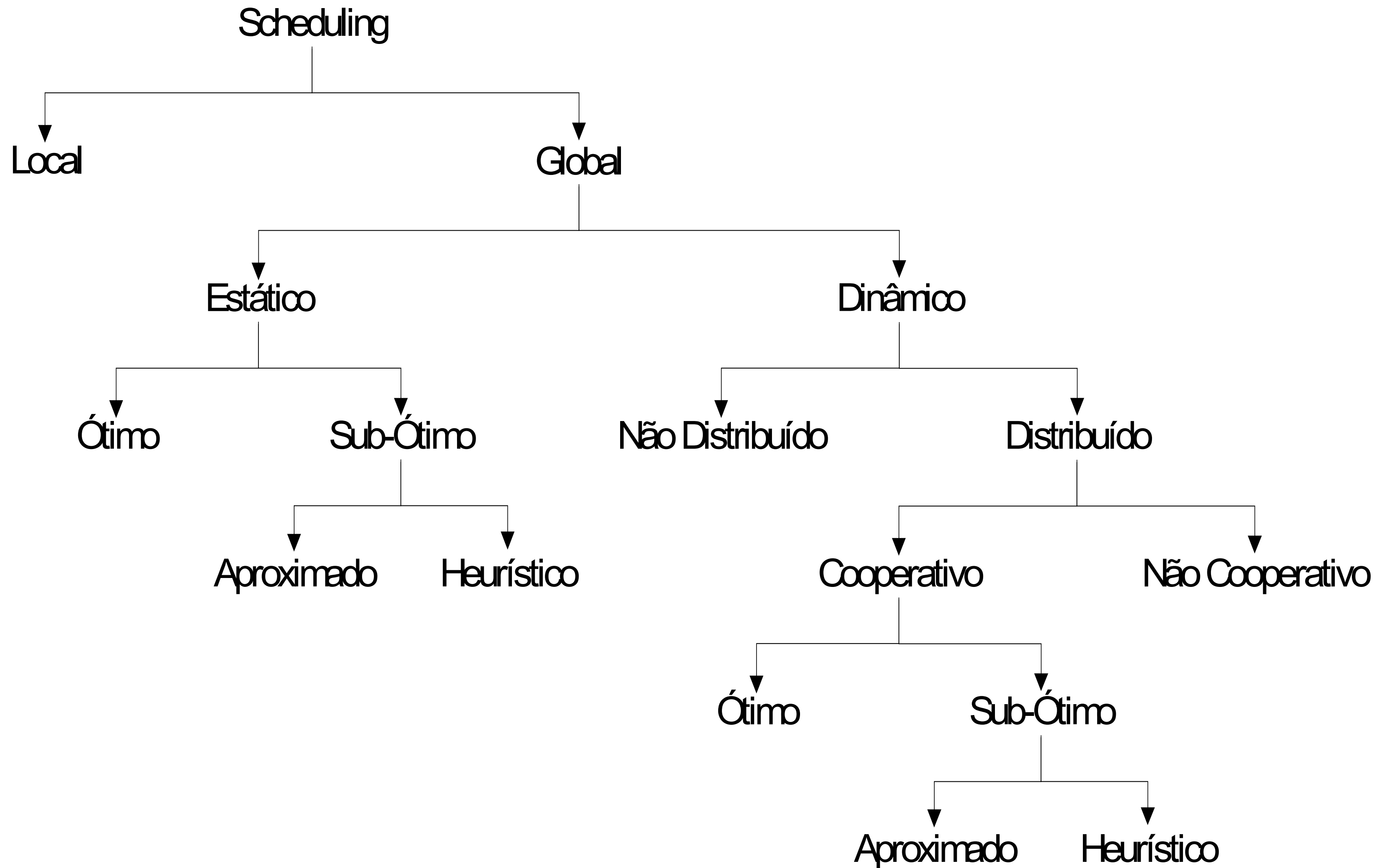
1.1.4 - Mapeamento (Mapping) ou Escalonamento

Neste passo faz-se a atribuição de tarefas a elementos de processamento, de forma a minimizar o tempo de execução.

Como objetivos intermediários, busca-se também maximizar a utilização dos recursos computacionais disponíveis (e.g. CPU) e minimizar os custos relativos à comunicação. Este é o problema que a literatura denomina de escalonamento (ou *scheduling*).

Os diferentes métodos de escalonamento são comumente classificados segundo a taxonomia apresentada por Casavant e Kuhl, que encontra-se representada na próxima figura.

Classificação dos métodos de escalonamento



Seguindo a taxomia apresentada, inicialmente os métodos de escalonamento (ou scheduling) são divididos em local e global.

Escalonamento local refere-se ao problema de atribuição das fatias de tempo *(time-slices)* de um processador aos processos. É aquele realizado normalmente pelo sistema operacional.

O *escalonamento global* refere-se ao problema de decidir à cerca de **onde** executar um processo sendo, portanto, seus métodos aplicáveis a sistemas distribuídos. Em face disto, veremos a seguir os dois grandes grupos nos quais se dividem os métodos de escalonamento globais.

Escalonamento Estático

Neste grupo, a atribuição de processos a processadores é realizada antes do início da execução do programa. Para isto é necessário que se tenha informações à cerca dos tempos de execução dos processos e dos elementos de processamento disponíveis, em tempo de compilação.

Uma vez realizado o escalonamento, ele não poderá ser alterado em tempo de execução, mesmo que ocorra defeito de algum EP (elemento de processamento) envolvido no escalonamento.

O escalonamento estático divide-se por sua vez em ótimo e sub-ótimo.

Em geral, obter escalonamentos ótimos é um problema NP-completo, somente sendo possível em casos restritos.

Por causa disto, normalmente se usam métodos sub-ótimos, que por sua vez se dividem em aproximados e heurísticos.

No escalonamento aproximado é feita uma busca no espaço de solução do tipo breadth-first (busca em amplitude) ou do tipo depth-first (busca em profundidade). Contudo, ao invés de percorrer todo o espaço de solução, o algoritmo pára quando encontra uma solução “boa” (ou aceitável). Nos métodos heurísticos, são usadas regras empíricas para orientar a busca por uma solução quase ótima.

Escalonamento Dinâmico

Os métodos deste grupo fazem frente a uma situação mais realista, onde muito pouco se sabe a priori à cerca das necessidades de recursos de um processo, ou do ambiente no qual o mesmo irá ser executado.

Escalonamento Dinâmico

No escalonamento dinâmico é realizada a (re)distribuição de processos a processadores durante a execução do programa, segundo algum critério.

Em geral os métodos deste grupo adotam como critério (ou política) o balanceamento da carga entre os EP, com o objetivo de melhorar o desempenho da aplicação.

São chamados, por isso, de métodos de balanceamento de carga (*Load Balancing*). Face à dificuldade em se estimar o tempo de execução de um processo e a natureza dinâmica dos recursos computacionais, estes métodos realizam a redistribuição de processos entre os EP, tirando trabalho dos que tenham ficado mais carregados e transferindo para os menos carregados.

Usualmente, ao se projetar um algoritmo de balanceamento de carga, definem-se três políticas:

- **Política de Informação:** especifica quais informações deverão ser passadas à cerca da carga dos EP. Diz ainda com que freqüência essas informações deverão ser atualizadas e para quem deverão ser enviadas;

- **Política de Transferência:** determina as condições sob as quais um processo deve ser transferido;

- **Política de Colocação:** identifica o EP para qual o processo deve ser transferido.

As decisões relativas ao balanceamento de carga podem ser tomadas de forma centralizada ou distribuída, ou ainda por uma combinação destas duas. Por exemplo, em um determinado algoritmo, a política de informação pode ser centralizada (um único EP recebe as informações de carga) enquanto que as de transferência e colocação podem ser distribuídas, sendo da responsabilidade de cada EP.

No caso do balanceamento de carga distribuído, caso os EP tomem suas decisões de forma totalmente independente diz-se que realizam escalonamento não cooperativo.

Um exemplo disto é o emprego de uma política de colocação aleatória.

No escalonamento cooperativo, ao contrário, o EP toma suas decisões de forma consoante com os demais EP, de forma a atingir um objetivo global para o sistema.

No caso da política de colocação, corresponde ao caso em que a escolha do EP é feita com base na informação de carga.

II – Paradigmas de Programação Paralela

Face ao tipo de paralelismo inerente ao problema e aos recursos computacionais disponíveis, os algoritmos paralelos desenvolvidos acabam por apresentar semelhanças em suas estruturas de controle. Com base nestas semelhanças surgiram os paradigmas (ou modelos) de programação paralela.

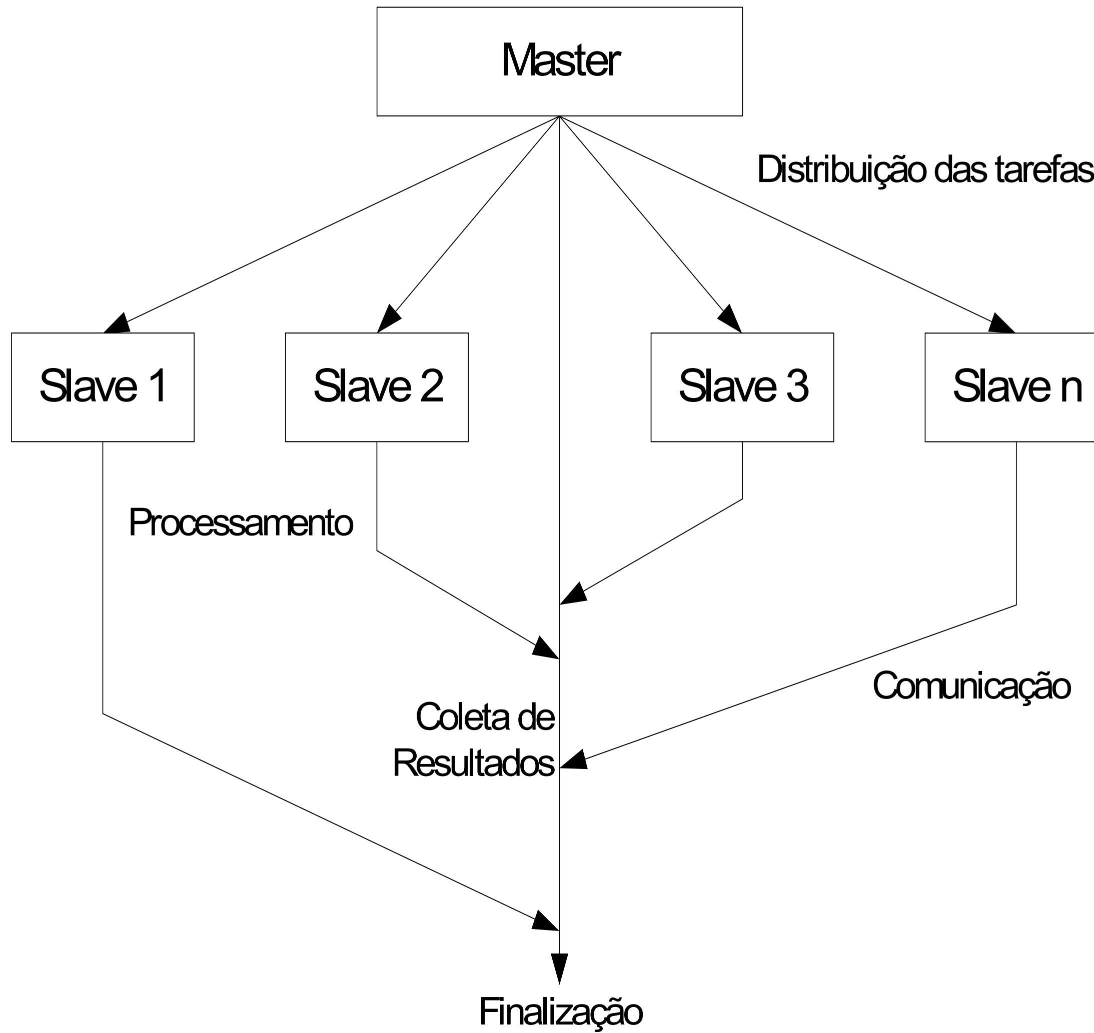
Podemos então definir paradigma, neste contexto, e segundo Hansen, como sendo uma classe de algoritmos que possuem a mesma estrutura de controle.

Na literatura, há várias propostas diferentes de classificação dos paradigmas.

Buyya e Silva analisam várias delas e os autores apresentam o que consideram ser um superconjunto de paradigmas, quais sejam:

- ***Master/Slave*** (ou Task-Farming): é o modelo das aplicações trivialmente paralelas, que corresponde à execução independente de componentes de um mesmo programa. Apresentamos na próxima figura

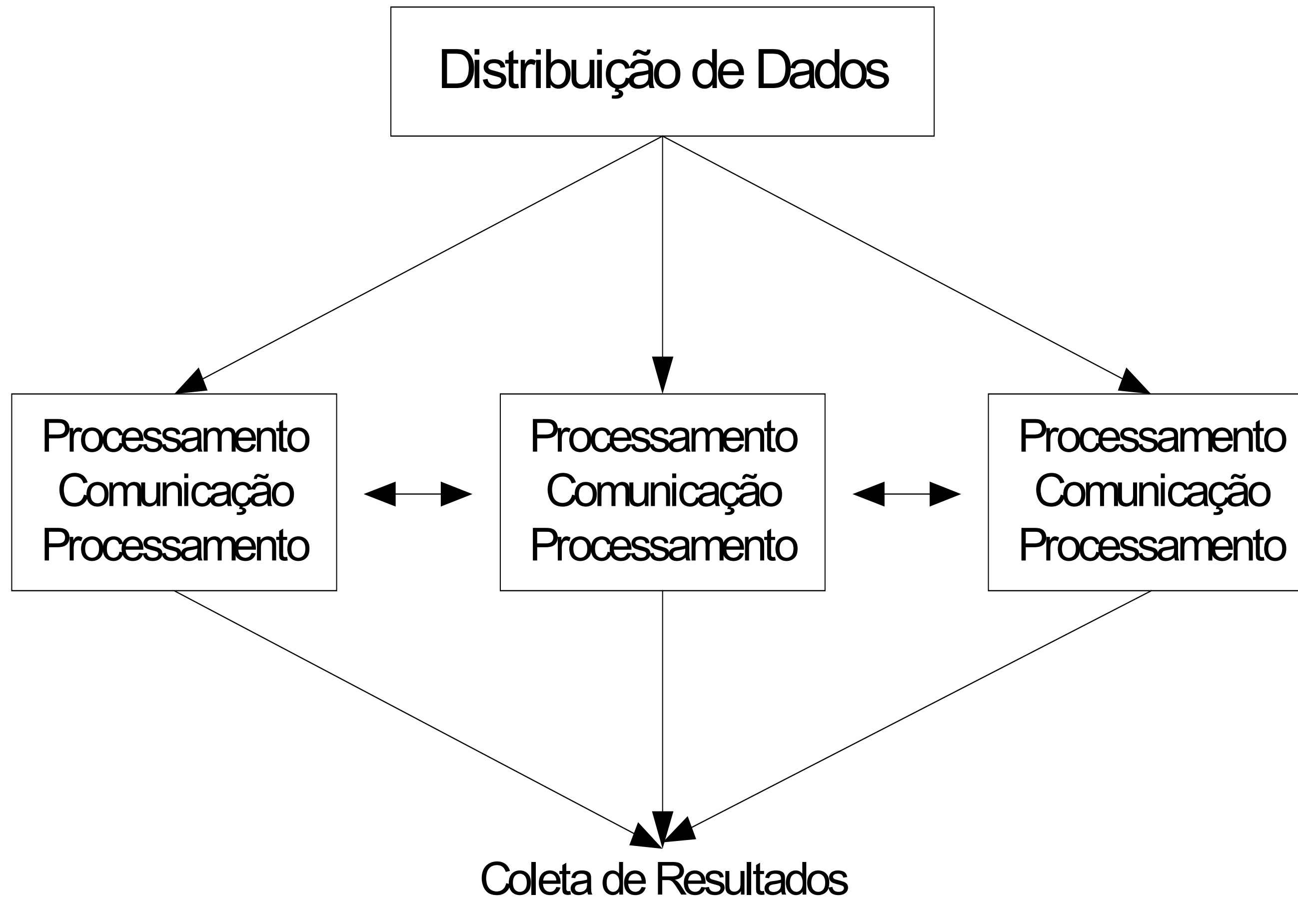
Estrutura do Modelo Master/Slave



Single-Program Multiple-Data (SPMD): segundo este paradigma, todos os processos executam o mesmo código, mas em partes diferentes dos dados.

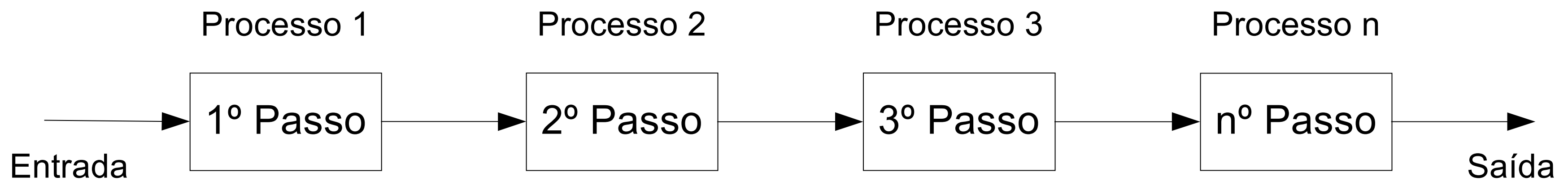
Os processos necessitam comunicar-se a fim de manter o sincronismo da execução, como na próxima figura

Estrutura Básica de um Programa SPMD.



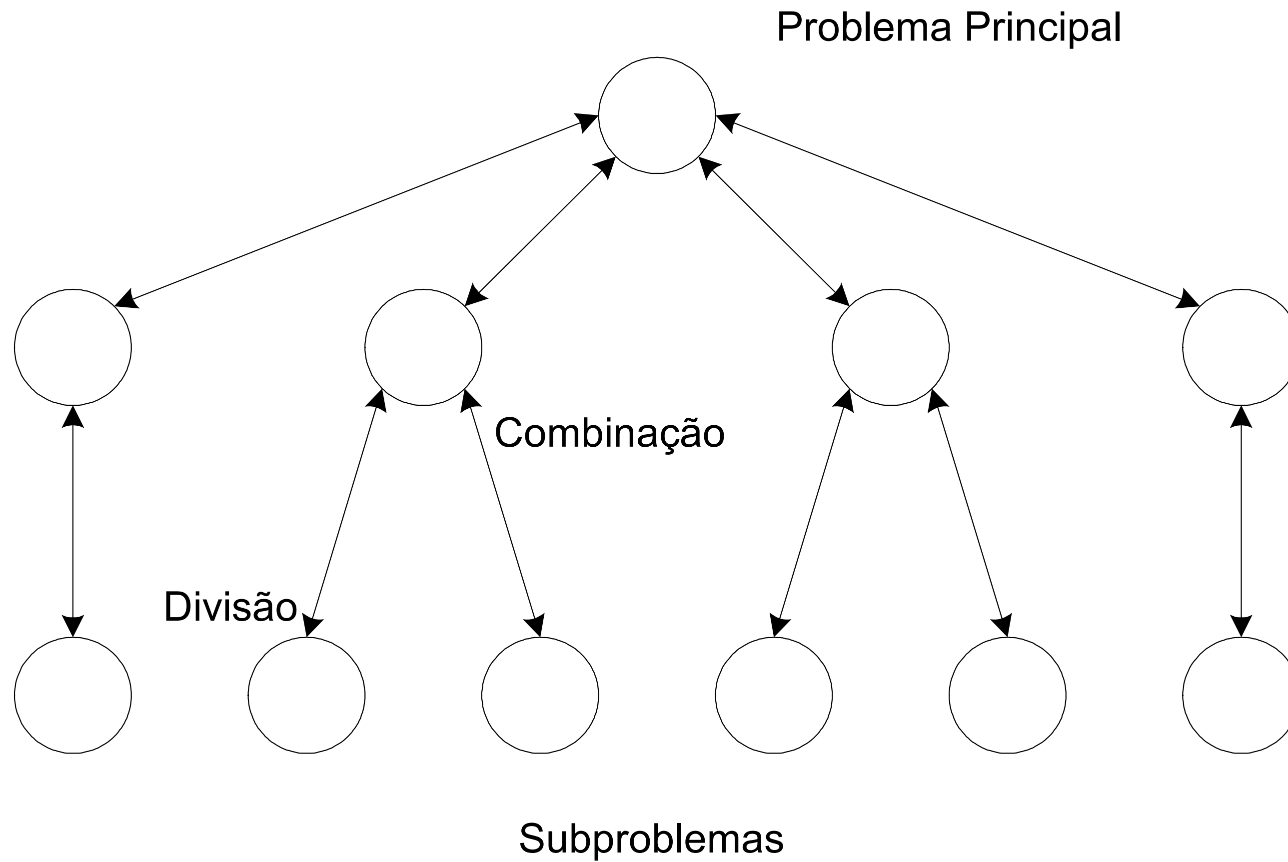
Pipelining de Dados: neste paradigma o problema é dividido em uma seqüência de passos. Cada passo vira um processo e é executado por um EP diferente. A próxima figura é um exemplo de ***pipelining de dados.***

Estrutura de um programa com Pipelining de Dados



- ***Dividir e Conquistar***: em termos gerais, neste modelo o problema é dividido em subproblemas, os quais são resolvidos independentemente e cujos resultados são então combinados para formar o resultado final. Enquanto que no paradigma Master/Slave a divisão do problema é feita somente no Master, neste paradigma o subproblema, uma vez atribuído, pode continuar a se dividir. A próxima figura exemplifica o modelo

Estrutura do Modelo Dividir e Conquistar.



- ***Paralelismo Especulativo***: empregado quando existem complexas dependências entre os dados, o que dificulta a utilização de outros paradigmas. Nestes casos o problema é dividido em partes pequenas e é utilizada especulação ou execução otimista a fim de facilitar o paralelismo.

Dependabilidade

Dependabilidade

O crescente uso de computadores na sociedade moderna acarreta, muitas das vezes, uma *dependência* intrínseca de certas atividades ao uso destes equipamentos. Como exemplo, podemos citar o controle de tráfego aéreo, o controle de vôo, o controle de reatores nucleares, de sistemas de suporte à vida e de sistemas de defesa.

Nestas áreas, a *falha* de algum sistema computacional pode acarretar, além de enormes prejuízos materiais, perda de vidas humanas. Em outras áreas, como o mercado de ações e o sistema bancário, o prejuízo envolvido é de natureza econômica.

Esta dependência nos impele à busca por sistemas computacionais mais confiáveis – o termo confiável, aqui, empregado no sentido mais amplo. Para que se possa expressar esta dependência de forma mais precisa e inequívoca, torna-se necessário à definição, ainda que informal, de alguns termos relacionados a este conceito.

Cabe salientar que na literatura as definições apresentadas não são unânimes e que a tradução dos termos empregados, do inglês para o português, dá margens ora à neologismos ora a sinonímias.

Desta forma, os termos em português serão apresentados acompanhados, por ocasião da sua definição, do original em inglês, a fim de manter clara a correlação com os vocábulos empregados originalmente.

O primeiro deles, que é exatamente o que exprime a relação de dependência, é o termo **dependabilidade** (do inglês *dependability*).

É uma propriedade dos sistemas computacionais que pode ser definida como a capacidade de prestar um serviço no qual se pode, justificadamente, confiar.

O **serviço** prestado por um sistema é o seu comportamento, tal qual percebido pelos usuários deste sistema. O **usuário** (do serviço) é um outro sistema (eventualmente o próprio homem) que interage com o primeiro através da **interface do serviço**. A **função** de um sistema traduz aquilo para o qual foi feito, e é descrita na **especificação do sistema.**

Diz-se que um **serviço** é **correto** quando implementa a especificação do sistema.

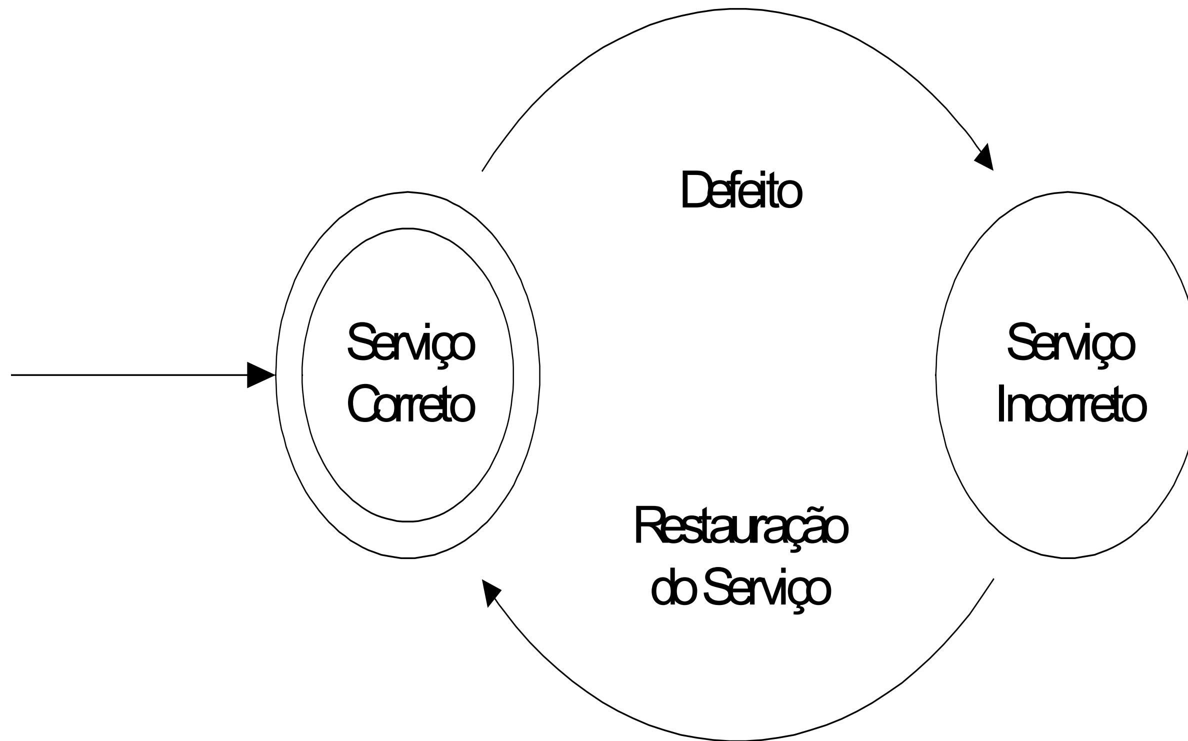
A caracterização da dependabilidade envolve ainda um conjunto de conceitos que alguns autores dividem em três grupos: os atributos, os meios (pelos quais será alcançada), e as ameaças.

Taxonomia da Dependabilidade

As Ameaças a Dependabilidade

Um sistema apresenta **defeito** quando não é capaz de prestar um serviço correto, ou seja, seu serviço se desvia da especificação do sistema. O defeito é o evento que causa a transição de estado do serviço de um sistema de correto para **serviço incorreto** (i.e., um que não implementa a especificação do sistema). A **restauração do serviço** é o evento que faz o serviço de um sistema retornar ao estado de serviço correto.

Transição de estados em um sistema devido à ocorrência de defeito.



O defeito só ocorre quando um erro existente no sistema alcança a interface do serviço e altera o serviço prestado. O **erro** é um *estado indesejado do sistema*, que *pode* vir a causar um defeito.

Desta forma, um sistema pode ter um ou mais erros e continuar apresentando serviço correto, ou seja, sem defeito. O tempo entre o surgimento de um erro e a manifestação do defeito é chamado de **latência de erro** e sua duração pode variar consideravelmente.

A **falha** é a causa do erro. Ela provoca no sistema uma transição de estado não planejada (indesejada) levando o mesmo para um *estado de erro*. Um sistema pode possuir uma ou mais falhas e não apresentar erros. Neste caso, a falha é chamada de **latente**.

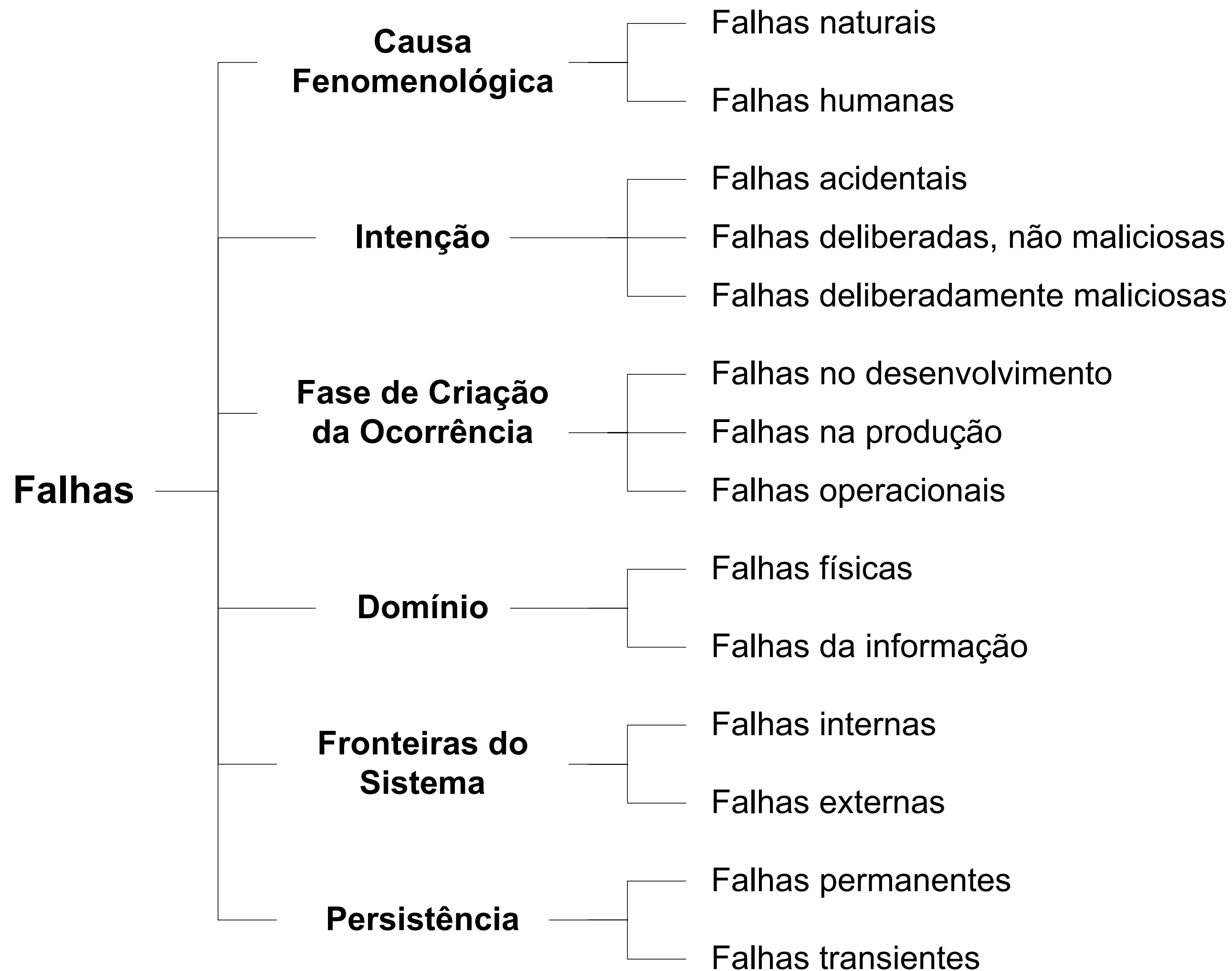
Quando ela efetivamente produz um erro passa a ser chamada de falha **ativa**. O tempo entre o surgimento da falha e sua ativação (produção do erro) é chamado de **latência de falha**.

Os tipos de falhas e as suas origens são bastante variados. Um curto circuito em uma porta lógica é uma falha. A conseqüência (alteração do resultado da operação lógica) é um erro, que poderá ou não vir a causar um defeito. Uma pessoa que ao operar um sistema realiza um procedimento inapropriado gera uma falha. O resultado de sua ação é o erro, que pode ser a alteração de um dado (informação).

Como último exemplo, uma perturbação eletromagnética (e.g., um raio) pode ser uma falha se possuir energia suficiente para, por exemplo, alterar (por indução) uma informação que esteja trafegando em um cabo metálico. Neste caso o erro é a informação distorcida que chegará para o destinatário

Avizienis apresenta um sistema de classificação defalhas bastante abrangente, que está reproduzido na próxima figura.

Classes de falhas



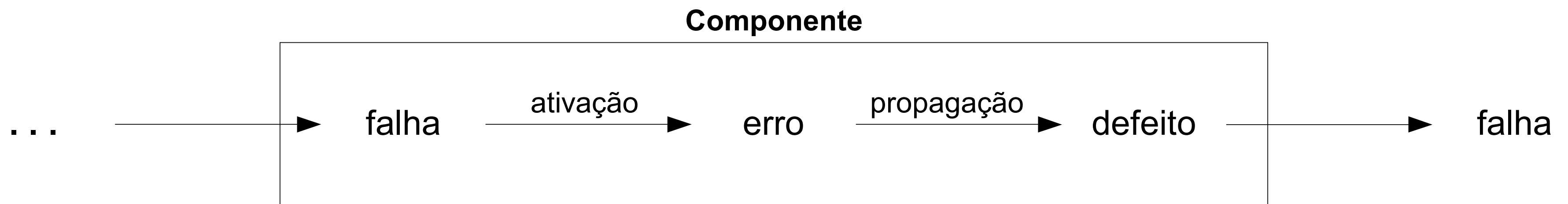
Os defeitos apresentados pelos sistemas, por sua vez, também podem ser de vários tipos diferentes. Alguns autores propõem taxonomias para classificar os defeitos, baseadas nas características dos comportamentos apresentados. São comumente chamadas de **modos de defeito** (*failure modes*) ou de modelos de defeito (*failure models*).

Modos de Defeito

O entendimento da relação de dependência entre falhas, erros e defeitos é a base para o que alguns pesquisadores chamam de **patologia da falha**.

Na próxima figura, observamos que a relação está aplicada a um componente. De fato, ela pode ser **utilizada recursivamente na análise de um sistema**, em diferentes níveis de abstração. Um componente de um sistema pode ser visto, ele próprio, como um (sub)sistema (composto, por sua vez, de outros componentes) que presta um serviço (ao sistema do qual é componente) que pode apresentar defeito. O defeito de um componente, desta forma, é considerado uma falha para o sistema que o engloba.

A cadeia de ameaças a dependabilidade



Os Atributos da Dependabilidade

Os atributos constituem-se em medidas da dependabilidade. Deve ficar claro, contudo, que face à natureza não determinística das circunstâncias consideradas, são associadas variáveis aleatórias a cada atributo e, portanto, as medidas obtidas são probabilidades.

Dependendo da aplicação envolvida, os atributos podem receber maior ou menor ênfase. Alguns podem mesmo chegar a ser dispensáveis. Outros, não listados aqui, podem vir a ser considerados. Os apresentados abaixo representam os mais comumente empregados

Confiabilidade

Probabilidade R(*t*) de um sistema apresentar serviço correto *continuadamente* durante um intervalo de tempo *t*, dado que o mesmo apresentava serviço correto em *t*=0. Em outras palavras, é a probabilidade do sistema não apresentar defeito durante o intervalo de tempo considerado.

Seja X o tempo para o sistema apresentar defeito, ou seja, a duração da “vida” de um sistema. X pode ser considerada como uma variável aleatória contínua, pois seu valor não pode ser previsto a partir de um modelo deteminístico. Na análise de sistemas computacionais, segundo alguns autores, a variável X possui **função de distribuição exponencial**. A probabilidade apresentada acima pode então ser expressa da seguinte forma:

Probabilidade R(t).

$$R(t) = P(X > t) = e^{-\lambda t}$$

O parâmetro λ é chamado de **taxa de defeito** do sistema e possui valor constante (devido à propriedade de ausência de memória da distribuição exponencial, que desconsidera o desgaste do sistema).

Outra forma muito comum de se expressar à confiabilidade de um sistema é pelo cálculo do seu **tempo médio para a ocorrência de defeito**, ou **MTTF** (*Mean Time To Fail*), que corresponde ao tempo de vida esperado do sistema:

Tempo médio para a ocorrência de defeito

$$MTTF = E[X] = \frac{1}{\lambda}$$

A expressão anterior fornece o MTTF de um sistema, trabalhando isoladamente.

De maneira geral, contudo, um sistema é composto por componentes que, por sua vez, podem ser considerados (sub)sistemas compostos por componentes e assim sucessivamente.

Desta forma, a confiabilidade de um sistema depende da confiabilidade de seus componentes. Vamos analisar, então, o MTTF de dois sistemas básicos a partir do MTTF de seus componentes.

Consideremos primeiramente um **sistema** formado por *n* componentes ligados **em série**. Neste caso, o sistema apresentará defeito caso algum de seus componentes apresente defeito. A probabilidade R(t) para este sistema fica da seguinte forma:

Probabilidade R(t) para um sistema em série

$$R(t) = R_1(t) \times R_2(t) \times \cdots \times R_n(t) = e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_n)t}$$

E desta maneira, seu **MTTF** é expresso pela equação abaixo:

$$MTTF_{série} = \frac{1}{\sum_{i=1}^{n} \lambda_i}$$

Tempo médio para a ocorrência de defeito em um sistema em série.

Da equação podemos concluir que a confiabilidade de um sistema em série é menor que a confiabilidade de qualquer um de seus componentes.

Consideremos agora um **sistema** formado por *n* componentes ligados **em paralelo** e que, desta forma, o sistema apresentará defeito somente quando todos os componentes apresentarem defeito. A probabilidade R(t) deste sistema é expressa da seguinte forma :

$$R(t) = 1 - \prod_{i=1}^{n} \left[1 - R_i(t)\right]$$

Probabilidade R(t) para um sistema em paralelo

Assumindo que todos os componentes possuam confiabilidade com distribuição exponencial e mesmo parâmetro λ, a fórmula acima fica equivalente a:

$$R(t) = 1 - \left[1 - e^{-\lambda t}\right]^n$$

Probabilidade R(t) para um sistema em paralelo, com distribuição exponencial e parâmetro λ.

Para este sistema, então, vamos obter o seu **MTTF** :

$$MTTF_{paralelo} = \frac{1}{\lambda}\sum_{i=1}^{n}\frac{1}{i}$$

Tempo médio para a ocorrência de defeito em um sistema em paralelo do tipo redundância ativa.

Da equação acima concluímos que a confiabilidade de um sistema em paralelo é maior que a de seus componentes.

Em tolerância a falhas, o emprego de componentes redundantes em paralelo, operando simultaneamente, com o intuito de aumentar a confiabilidade de um sistema, é conhecido como

redundância ativa (*active redundancy*).

Seu MTTF é o expresso pela fórmula anterior.

Outra forma de organização de um sistema, muito empregada em tolerância a falhas, é a de

redundância em reserva (*standby redundancy*)

onde apenas um dos componentes redundantes (chamado primário) fica operacional e todos os demais permanecem inativos. Quando o primário apresenta defeito, um dos componentes em reserva é ativado (instantaneamente) e passa a prestar o serviço no lugar daquele.

O MTTF deste tipo de sistema, para distribuições exponenciais, é expresso por

$$MTTF_{standby} = \frac{n}{\lambda}$$

Tempo médio para a ocorrência de defeito em um sistema em paralelo do tipo redundância em reserva.

Reparemos que a confiabilidade nos sistemas em paralelo é tanto maior quanto maior for o número de componentes redundantes.

Disponibilidade

É a probabilidade A(t) de um sistema apresentar serviço correto *num determinado instante* de tempo t. Definida assim, é chamada de **disponibilidade instantânea**

Contudo, na análise da disponibilidade, normalmente estamos interessados no comportamento de um sistema ao longo de períodos de tempo, no que se refere à alternância entre períodos de serviço correto e períodos de defeito (situação esta na qual consideraremos o sistema em reparo).

Em outras palavras, queremos saber a fração de tempo na qual um sistema deverá ter condições de apresentar serviço correto.

Neste sentido, definimos a **disponibilidade α** de um sistema como o limite de A(*t*) quando *t* tende a infinito (formalmente chamado de **limite da disponibilidade**).

$$\alpha = \lim_{t \to +\infty} A(t)$$

Disponibilidade de um sistema.

Utilizando o **tempo médio para a ocorrência de defeito** (**MTTF**), apresentado anteriormente, e o

tempo médio para reparo (**MTTR**, *Mean Time To Repair*),

podemos exprimir a disponibilidade α como sendo

$$\alpha = \frac{MTTF}{MTTF + MTTR}$$

Disponibilidade em função do MTTF e do MTTR

Segurança (contra catástrofes)

É a probabilidade $S_1(t)$ de um sistema não apresentar defeito que acarrete conseqüências catastróficas contra os usuários ou contra o meio ambiente, em um intervalo de tempo t.

Confidencialidade

Probabilidade C(t) de não ocorrer divulgação não autorizada de informação, em um intervalo de tempo t.

Integridade

Probabilidade I(*t*) de não ocorrer alterações impróprias de estado em um sistema, em um intervalo de tempo *t*.

Reparabilidade

Probabilidade M(*t*) de um sistema estar restaurado (retornar ao estado de serviço correto) no tempo t, dado que o mesmo apresentou defeito em *t*=0. Em outras palavras, é a capacidade que um sistema tem de passar por reparos e modificações.

Segurança (*security*)

Além dos atributos mencionados acima, citamos também o da segurança, que é um atributo obtido através da combinação de três outros:

a) disponibilidade, para usuários autorizados;
b) confidencialidade; e
c) integridade, com "impróprio" significando "não autorizado".

Segurança pode ser definida como a probabilidade $S_2(t)$ de que não ocorra acesso ou manipulação indevidos no estado do sistema, em um intervalo de tempo t.

Os Meios para se Alcançar a Dependabilidade

A fim de se desenvolver um sistema dependável, é necessário o emprego combinado de quatro técnicas:

- a prevenção de falhas (que permite prevenir a introdução ou ocorrência de falhas);

- a tolerância a falhas (que permite que seja apresentado serviço correto na presença de falhas);
- a remoção de falhas (que permite reduzir o número ou a severidade das falhas); e

- a previsão de falhas (que permite estimar a quantidade atual, a incidência futura, e as prováveis conseqüências das falhas).

Prevenção de Falhas

Na prevenção de falhas o objetivo é aumentar a confiabilidade dos sistemas através, basicamente, do emprego de **técnicas de controle da qualidade** nas etapas de projeto e desenvolvimento dos mesmos.

Uma vez que não há como, através da prevenção, eliminar todas as falhas possíveis (e.g., falhas de operação do sistema, fim da vida útil de componentes, etc.) e pelo fato desta técnica não fazer uso de redundância, é assumido que, eventualmente, irão ocorrer defeitos no sistema.

Desta forma, são previstos **procedimentos manuais de reparo** a fim de restaurar o sistema à condição de serviço correto.

Isto faz com que a prevenção de falhas, por si só, seja insuficiente para alguns sistemas atingirem a dependabilidade desejada devido, principalmente, às seguintes conseqüências: a interrupção do serviço (para reparos) pode estender-se por períodos de tempo inaceitáveis para os usuários do mesmo (e.g., sistemas de tempo real); para o reparo manual, há necessidade de se ter acesso ao sistema, o que nem sempre é possível (e.g., naves automatizadas de exploração do espaço); o reparo pode representar um custo elevado (e.g., sistema bancários, de defesa, de suporte à vida).

A fim de minimizar estas conseqüências pode-se complementar o uso da prevenção de falhas com o emprego da tolerância a falhas, que é uma técnica que faz uso da redundância para substituir o reparo manual do sistema pelo reparo e reconfiguração automatizados, o que aumenta a confiabilidade e a disponibilidade do sistema.

Tolerância a Falhas

Tolerância à falhas é a habilidade que um sistema tem de apresentar um comportamento muito bem definido na ocorrência de falhas ativas.

Este comportamento apresentado pode ser dividido em quatro grandes categorias ou formas de tolerância à falhas, de acordo com o atendimento ou não do que considera ser duas das propriedades comportamentais principais dos sistemas:

- a segurança contra catástrofes (safety); e
- a "vitalidade" (liveness), no sentido de permanecer ou não operacional.

Formas básicas de tolerância à falhas

Propriedades do Sistema	Operacionalidade garantida	Operacionalidade não garantida
Segurança garantida	Mascaramento	**Defeito seguro (*fail safe*)**
Segurança não garantida	**Sem mascaramento**	**Não tolerância**

A primeira forma de tolerância a falhas é a que realiza o **mascaramento** (empregado na acepção de encobrir ou ocultar) das mesmas.

Nesta o serviço apresentado pelo sistema não deverá ser modificado pela ocorrência de falhas, ou seja, o sistema como um todo não deverá apresentar defeito.

Sendo assim, o sistema deverá permanecer operacional e em um estado seguro (para os usuários e para o meio ambiente).

É a forma mais completa de tolerância a falhas, a mais desejada e, também, a de maior custo. Alguns autores ao definirem tolerância a falhas, apenas consideram esta forma.

Todas as demais formas modificam o serviço prestado pelo sistema na ocorrência de falhas.

A **não tolerância** a falhas é considerada uma forma extrema de tolerância. É a mais trivial, mais barata, mais fraca e, por isso mesmo, a mais indesejada.

Na ocorrência de falhas o sistema provavelmente apresentará defeito e, o mais grave, nada poderá ser afirmado quanto ao estado em que o sistema se encontrará após isso, podendo o mesmo ingressar em um estado não operacional ou mesmo em um estado inseguro.

Entre um e outro extremo, encontram-se as duas formas intermediárias de tolerância à falhas:

• a que garante que o sistema irá permanecer em um estado seguro, mas nada diz sobre o seu estado operacional, e por isso é chamada de **defeito seguro**;

• e a que garante que o sistema irá permanecer operacional, ainda que o mesmo ingresse, por causa da falha, em um estado inseguro, chamada de tolerância à falhas **sem mascaramento**

A primeira é sempre preferível à segunda, uma vez que segurança é muito mais importante que permanecer operacional.

Vamos ilustrar este ponto através de dois exemplos. O sistema de controle de um foguete, que o explode no ar caso se desvie da rota, é um exemplo de tolerância a falhas tipo defeito seguro, pois garante a segurança (da base de lançamento e da população, onde porventura os destroços poderiam cair) em detrimento da operacionalidade.

Já um sistema de controle de semáforos, em um cruzamento, que apresente perda de sincronismo, mas que permaneça funcionando, é um exemplo de tolerância a falhas tipo sem mascaramento, uma vez que a operacionalidade foi garantida (os semáforos continuam funcionando) ainda que com prejuízo da segurança (a perda de sincronismo pode causar uma situação em que a luz verde acenda para ambas as vias do cruzamento).

Pesquisadores indicam que existem três formas básicas de redundância :

- na estrutura,
- na informação e
- no tempo

que podem ser implementadas em hardware ou em software, conforme o caso, o que resulta num total de seis formas combinadas.

Classificação da redundância em sistemas tolerantes à falhas.

As duas formas de redundância na estrutura se baseiam na replicação de partes componentes de um sistema (ou até no sistema como um todo).

As réplicas podem ser idênticas (em sua constituição) ou não, mas sempre possuem a mesma função. Segundo alguns pesquisadores é a única forma de redundância capaz permitir a tolerância de falhas permanentes.

A utilização de duas ou mais versões de um mesmo algoritmo, programados de forma independente, para comparação das saídas e escolha (por maioria) do resultado correto – técnica conhecida como *N-version programming* – é um exemplo de redundância na estrutura baseada em software.

Sua contrapartida em hardware, onde ao invés de versões de um mesmo algoritmo temos cópias de um mesmo módulo de hardware, é chamada de *N-modular redundancy* (NMR).

Nem sempre é necessário duplicar um módulo inteiro para se obter tolerância à falhas. Às vezes, a utilização da redundância na informação pode ser suficiente. A técnica pela qual é mais comumente empregada se baseia na aplicação da teoria dos códigos corretores de erro, que acrescentam informação redundante ao conteúdo original de forma a permitir a detecção e a correção de erros. Como exemplos desta técnica podemos citar os códigos de paridade, de CRC (*Cyclic Redundancy Check*) e de *checksum* (*SUMmation CHECK*).

Por fim, pode-se ainda lançar mão da redundância no tempo, na qual a mesma atividade é repetida uma ou mais vezes.

As repetições baseiam-se no pressuposto de que a causa do problema é de natureza temporária.

E de fato, como já foi provado, a utilização da redundância no tempo mostra-se mais adequada para detectar erros que resultem da ocorrência de falhas transientes. Um exemplo desta forma de redundância é a retransmissão de mensagens que ocorre em protocolos de comunicação (como o TCP) quando o remetente da mensagem deixa de receber a confirmação do recebimento pelo destinatário.

Vimos até agora que no projeto de um sistema tolerante a falhas é necessário definir as falhas que serão toleradas e o comportamento desejado para o sistema na ocorrência delas.

Vimos também que tornar um sistema tolerante a falhas implica no emprego de pelo menos uma das formas de redundância apresentadas.

Agora veremos quais são as atividades normalmente desenvolvidas pelos sistemas na implementação da tolerância à falhas.

O ponto de partida é a **detecção de erros**. Isto é devido ao fato de que a ocorrência de falhas não pode ser observada diretamente e, sendo assim, tem de ser deduzida a partir da presença de erros [Jalote].

Como os erros são uma propriedade do estado do sistema, existem testes que podem ser conduzidos a fim de se provar a existência ou não deles.

Em conseqüência, um esquema de tolerância a falhas será tão eficaz quanto for o seu mecanismo de detecção de erros.

De acordo com [Jalote], o **teste ideal** para se detectar a presença de erros deve satisfazer três importantes propriedades: primeiro, o teste não deve ser influenciado pela implementação do sistema, devendo basear-se apenas na especificação do mesmo; segundo, deve ser completo e correto, devendo identificar todos os erros de fato existentes; por último, o teste deve possuir independência do sistema com relação à ocorrência de falhas, caso contrário a falha que acarreta um erro no sistema pode vir a causar um erro também no teste.

No mundo real, infelizmente, estas propriedades são muito difíceis de serem satisfeitas.

Desta forma, ao invés de testes ideais, normalmente são empregados **testes de aceitação**, que são aproximações dos testes ideais.

Neste caso, não há garantias de que os testes realizados vão ser capazes de detectar todos os erros existentes.

Depois de detectado o erro, é necessário realizar a **recuperação do sistema**, a fim de levá-lo de volta a um estado onde não haja erros detectados e nem falhas que possam ser ativadas, vindo a causar erros novamente. A recuperação, desta forma, consiste de duas ações: o tratamento dos erros e o tratamento das falhas.

O **tratamento dos erros** (*error handling*) visa colocar o sistema de novo em um estado correto. Ele pode atuar de duas formas: ***rollback***, que leva o sistema de volta a um estado anterior, identificado como correto, a partir de algum dos *checkpoints* (gravações periódicas do estado do sistema) que possua; ***rollforward***, que leva o sistema a um estado novo, no qual ainda não esteve, sem erros detectáveis.

Já o **tratamento das falhas** (*fault handling*) tem por objetivo impedir que as falhas por ventura localizadas venham a ativar-se novamente, causando erros outra vez. Para atingir seu objetivo, o tratamento das falhas se divide em quatro etapas: o **diagnóstico das falhas**, que localiza e identifica o tipo da(s) causa(s) do(s) erro(s); o **isolamento das falhas**, que realiza a exclusão (física ou lógica) dos componentes (diagnosticados com falhas) da participação no serviço do sistema; a **reconfiguração do sistema**, que realiza a ativação de um componente em reserva ou a redistribuição de tarefas entre os componentes restantes; e a **reinicialização do sistema**, que verifica, atualiza e grava a nova configuração do sistema.

Normalmente, após o tratamento das falhas, segue-se uma manutenção corretiva do sistema, a fim de remover as falhas encontradas e isoladas. Esta atividade não faz parte da tolerância à falhas, e distingue-se dela pela necessidade que tem da participação de um agente externo.

Mais uma vez ressaltamos a importância da escolha da hipótese de falha. Ela influenciará diretamente na escolha e implementação das técnicas para a detecção de erros, para o tratamento dos mesmos, e para o tratamento das falhas.

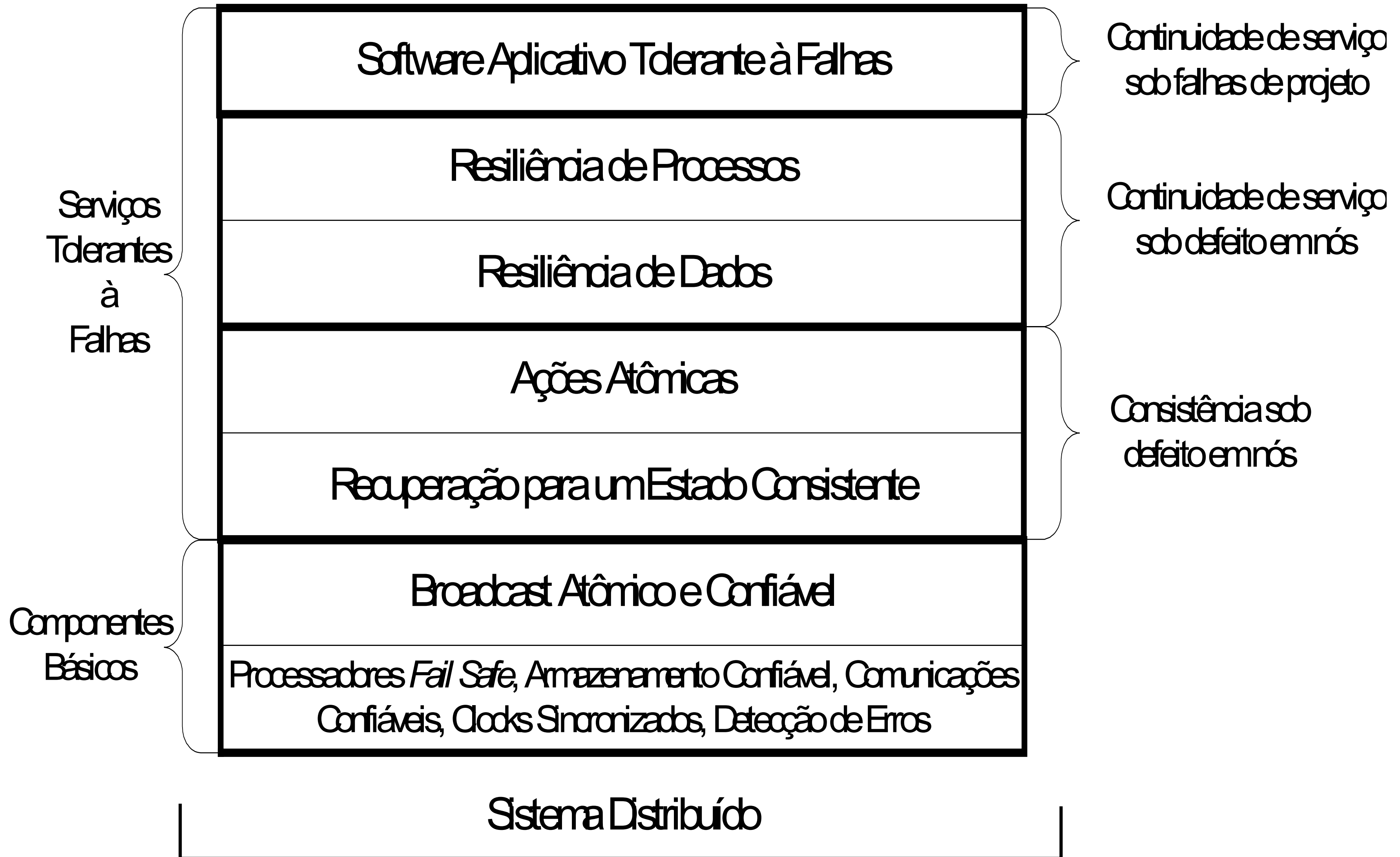
É importante destacar que a tolerância a falhas deve ser encarada como um conceito recursivo. É essencial que os mecanismos utilizados nesta atividade sejam, eles também, tolerantes à falhas. Ainda assim, infelizmente, caso as falhas ocorram de forma muito freqüente ou sejam muito graves, haverá sempre a possibilidade de que o sistema apresente defeito, conseqüência do fato de que a tolerância à falhas é limitada no espaço ou no tempo.

Finalizando este tópico, vamos verificar um exemplo de como o problema da tolerância a falhas pode ser abordado em sistemas distribuídos.

{Jalote] apresenta o sistema onde o mesmo é estruturado em diferentes níveis de abstração, cada um provendo diferentes serviços de tolerância à falhas.

Como apresentamos na próxima figura.

Níveis de tolerância à falhas em um sistema distribuído.



Os dois níveis mais baixos provêm os componentes básicos para a implementação de serviços tolerantes a falhas, mas que são, por si só, de limitado interesse.

Os próximos dois níveis ocupam-se de serviços que são necessários à garantia da consistência, quando da ocorrência de defeito em nós do sistema.

Apesar da ação atômica contribuir para a garantia da consistência, ela implica no cancelamento do que estava em andamento e tenha sido interrompido por algum defeito.

Para contornar esta situação, os dois níveis seguintes preocupam-se em garantir que uma vez iniciada uma ação, ela sempre termine, ainda que defeitos venham a ocorrer.

Por último, ainda que tudo o mais venha a funcionar corretamente, existe a possibilidade do software aplicativo conter falhas de projeto.

Desta forma, este último nível provê uma abstração na qual o software aplicativo é capaz de tolerar suas próprias falhas.

Remoção de Falhas

No caminho para se obter a dependabilidade, a remoção de falhas constitui-se em mais uma ferramenta valiosa, além da prevenção de falhas e da tolerância à falhas. Esta técnica pode ser aplicada tanto na fase de desenvolvimento como ao longo da vida operacional do sistema.

A **remoção de falhas na fase de desenvolvimento** é realizada em três etapas. A primeira delas é a **verificação**, onde é checado se o sistema atende a certas propriedades (denominadas condições de verificação).

Caso seja constatado que as condições não são atendidas, é feito o **diagnóstico** das falhas que acarretaram o não atendimento. Por último, é executada a **correção** do sistema, para eliminação das falhas diagnosticadas.

Em particular, a verificação das especificações do sistema é também chamada de **validação**.

As técnicas de verificação podem ser classificadas em estáticas e dinâmicas. A **verificação estática** checa o sistema sem chegar a colocá-lo em execução.

A **verificação dinâmica**, por outro lado, ativa o sistema para conferir as condições de verificação. Mas, para isto, ela necessita fornecer ao sistema as entradas que lhe forem necessárias, que podem ser de natureza fictícia, e daí gerar uma **execução simbólica**, ou podem ser reais, gerando então um **ensaio** (*testing*).

A verificação dos mecanismos de tolerância a falhas é um importante aspecto da remoção de falhas.

Na execução dos ensaios com o sistema, pelo uso de uma técnica conhecida como **injeção de falhas** (*fault injection*), pode-se criar falhas programadas a fim de colocar à prova a eficácia dos mesmos.

Já a **remoção de falhas ao longo da vida operacional do sistema** é realizada através de atividades de manutenção.

Para alguns pesquisadores são duas as atividades a serem consideradas:

- a **manutenção corretiva**, que visa remover falhas diagnosticadas no sistema (pela ação de tratamento das falhas, da tolerância à falhas, por exemplo); e

- a **manutenção preventiva**, que tem como objetivo descobrir e remover falhas latentes no sistema.

As atividades de manutenção podem ser realizadas com o sistema em funcionamento (*on-line*) ou podem requerer a parada do mesmo (*off-line*).

São consideradas atividades complementares às ações de tolerância a falhas, sem, contudo, as requerer.

Previsão de Falhas

O último meio utilizado para se alcançar a dependabilidade, a previsão de falhas, realiza uma avaliação do comportamento do sistema com relação à ocorrência e ativação de falhas. Esta avaliação possui dois aspectos: um, **qualitativo**, que busca identificar, classificar e ordenar por importância, as causas de defeito nos sistemas; outro, **quantitativo**, que busca mensurar, em termos probabilísticos, o atendimento aos atributos da dependabilidade. Tanto a avaliação qualitativa como a quantitativa preocupam-se em obter dados que validem as escolhas feitas na constituição da estrutura de dependabilidade do sistema, ou que subsidiem modificações na mesma para que a sua eficácia ou sua eficiência sejam melhoradas.

Relacionando Dependabilidade e Sobrevivência (*Survivability*) do Sistema

Este tópico tem a finalidade de fazer uma distinção entre os termos, que podem ser equivocadamente considerados sinônimos.

Survivability pode ser informalmente definida como uma propriedade que um sistema tem de continuar provendo serviço (possivelmente degradado ou modificado) sob determinadas condições hostis de operação.

Mais precisamente, um sistema é considerado sobrevivente se possuir:

- Uma declaração dos ambientes operacionais suportados pelo sistema;
- Um conjunto de especificações que definam os diferentes níveis de funcionalidades que o sistema é capaz de prover. Em alguns casos, pode ser aceitável a existência de um nível que apresente funcionalidade alguma;

– Uma priorização dos níveis, denotando a preferência de provisionamento das funcionalidades;

- Uma probabilidade associada a cada nível, que indique a chance daquele nível de serviço ser alcançado.

Portanto, das definições acima, sobressai claramente a origem do conceito, que vem do meio militar.

Survivability não é, desta forma, dependabilidade, mas ambas possuem muito em comum.

Resumo

Inicialmente falamos dos sistemas distribuídos, comentando suas arquiteturas e seus modelos de programação.

Depois comentamos sobre os ambientes de cluster e de grid, dois dos tipos mais comuns de sistemas distribuídos nos dias atuais.

Falamos a seguir das formas de programação utilizadas em ambientes distribuídos, onde pudemos ressaltar as dificuldades envolvidas e as características de cada uma delas.

Por último, falamos da dependabilidade e, dentre seus meios, detalhamos melhor a tolerância à falhas.