5
Fidedignidade

O objetivo desta seção é apresentar a terminologia comumente utilizada na literatura de tolerância a faltas. Esta introdução é importante para auxiliar na identificação dos aspectos de tolerância a faltas na abordagem apresentada nesta tese.

Um software é dito fidedigno quando se pode, justificavelmente, depender dele assumindo riscos de danos compatíveis com o serviço prestado pelo software (Avizienis, Laprie et al. 2004). Comumente fidedignidade é definida em termos probabilísticos, indicando a probabilidade de que o sistema esteja funcional em um determinado momento. Esta definição deriva conceitos e métricas tais como *mean-time-to-failure* (MTTF). É possível ainda identificar dois focos principais em fidedignidade: o projeto de sistemas fidedignos e a avaliação da fidedignidade. Esta separação é comumente classificada como os atributos de fidedignidade (perspectiva de avaliação) e os meios para alcançá-los (perspectiva de projeto).

Sob a perspectiva de avaliação, a fidedignidade pode ser decomposta nos seguintes atributos (Staa 2006):

Disponibilidade: estar pronto para prestar serviço correto sempre que se necessite do software.

Confiabilidade: habilidade de sempre prestar serviço correto. Conforme observado em (Boehm and Basili 2001), mais de 50% das falhas observadas em sistemas em uso devem-se a erros de operador. Ou seja, confiabilidade não é somente assegurar que o sistema esteja em correspondência exata com a sua especificação.

Segurança (safety): habilidade de evitar conseqüências catastróficas, ou de grande envergadura, afetando tanto os usuários como o ambiente.

Proteção: habilidade de evitar tentativas de agressão bem sucedidas.

Privacidade: habilidade de proteger dados e código contra acesso indevido.

Integridade: ausência de alterações não permitidas.

Robustez: habilidade de detectar, o mais cedo possível, eventuais falhas de modo que os danos (as correspondentes conseqüências) possam ser mantidas em um patamar aceitável.

Recuperabilidade: habilidade em ser rapidamente reposto em operação fidedigna após a ocorrência de uma falha.

Manutenibilidade: habilidade de ser modificado (evoluído) ou corrigido sem que novos problemas sejam inseridos e a um custo compatível com o tamanho da alteração.

Depurabilidade: facilidade de diagnosticar e eliminar possíveis causas de problemas a partir de relatos gerados.

Nota-se que todos os atributos de fidedignidade citados acima possuem uma relação direta com avaliação, ou seja, suas definições estão relacionadas com a medição em maior ou menor grau da presença de cada atributo. Por exemplo, a definição de disponibilidade é facilmente associada a uma métrica que indique o quão disponível o sistema esteve em um determinado intervalo de tempo.

Por outro lado, a perspectiva de projeto de sistemas fidedignos, frequentemente caracterizada como os meios para se alcançar melhores medidas para os atributos, se preocupa com métodos e técnicas que propiciam a construção de sistemas fidedignos. Sob esta ótica é possível destacar quatro categorias (Laprie and Randell 2004):

Prevenção de faltas: objetiva a prevenção da introdução de faltas no sistema. A maneira mais comum de se realizar a prevenção de faltas é através de métodos de controle da qualidade (Team 2006).

Tolerância a faltas: objetiva evitar falhas devido a presença de faltas.

Remoção de faltas: objetiva reduzir o número e a severidade das faltas.

Predição de faltas: objetiva estimar o número de faltas atuais, a evolução deste número de faltas e suas prováveis conseqüências.

A definição de faltas, falhas, erros encontrada na literatura é muitas vezes vaga e ambígua. Em (Cristian 1991) por exemplo, o autor reconhece que “o que uma pessoa chama de falha, uma segunda chama de falta e uma terceira chama de erro”. O entendimento preciso destas noções evita más interpretações e auxilia no entendimento do escopo das contribuições propostas nesta tese. Por esta razão e baseado no trabalho publicado por Gärtner (Gärtner 1999), a seguir será apresentada a definição destes conceitos.

Um sistema distribuído pode ser modelado como um conjunto finito de processos que se comunicam utilizando um subsistema de comunicação através da troca de mensagens compostas de um alfabeto finito de símbolos. As variáveis de cada processo definem um *estado local*. Cada processo executa um algoritmo local que resulta em uma seqüência de transições no seu estado local. Cada transição de estado define um evento, que pode ser um evento do tipo *send*, um evento do tipo *receive*, ou um evento *interno*.

O estado geral do sistema distribuído é chamado de *configuração*, e é composto pelo conjunto de todos os estados locais de todos os processos mais as mensagens em trânsito. Para auxiliar no entendimento destas definições, considere a listagem de Código 32. Este código descreve um algoritmo distribuído onde dois processos enviam mensagens alternadas um para o outro. O *estado local* do processo Ping é dado pelo valor das variáveis *z* e *ack*. O estado geral do sistema, ou *configuração*, é dado pelo valor de todas as variáveis (*z, ack* e *wait*) e pelo conjunto de mensagens que possam estar em trânsito (*a* ou *m*).

Os processos esperam que as condições de guarda (lado esquerdo do símbolo $\rightarrow$) sejam verdadeiras para que o lado direito seja executado acarretando na mudança de seu estado local e consequentemente na mudança da configuração do sistema.

```
process Ping
  var z: integer init 0
      ack: boolean init true
  begin
      ¬ack ∧ rcv(m) → ack := true; z := z + 1
      ack           → snd(a); ack := false
  end

process Pong
  var wait: boolean init true
  begin
      ¬wait         → snd(m); wait := true
      wait ∧ rcv(a) → wait := false
  end
```

Código 32 – Pseudocódigo do Sistema Distribuído Ping-Pong

A definição de falta utilizada neste trabalho é baseada na observação de que os sistemas mudam suas configurações baseadas em dois tipos de eventos: uma operação normal do sistema ou a ocorrência de faltas. Sendo assim, uma falta pode ser modelada como sendo uma mudança indesejada no estado de um processo. Como um exemplo, a listagem de Código 33 mostra o código do

processo Pong modificado para a inclusão de uma falta. A falta é incluída através da adição da variável *up* com o valor inicial *true* e da ação $\text{up} \rightarrow up := \text{false}$. Como conseqüência, no momento em que a ação de falta for executada, o processo Pong não enviará mais mensagens e não responderá ao processo Ping.

```
process Pong
  var wait: boolean init true
  error up: boolean init true
  begin
      {* normal program actions *}
      up ∧ ¬wait          → snd(m); wait := true
      up ∧ wait ∧ rcv(a)  → wait := false
      {* fault actions *}
      up                  → up := false
  end
```

Código 33 – Processo Pong com a Inclusão de uma Falta

A adição da variável *up* e da ação de falta transformou o sistema em um sistema com falta. A inclusão da falta pode levar ao sistema a não comportar de acordo com a sua especificação. Quando isto acontece, utiliza-se o termo falha. Ou seja, uma falha é definida como o fato do sistema não apresentar o comportamento esperado de acordo com a sua especificação.

Desta forma, as quatro principais técnicas para a construção de sistemas fidedignos (prevenção, tolerância, remoção e predição) se preocupam, principalmente, com a investigação das propriedades e técnicas para lidar com sistemas e faltas.

5.1.Detecção de Falhas

Frequentemente em sistemas distribuídos é preciso identificar quais os processos estão operacionais e quais não estão (*crashed*). Entretanto, as várias variáveis envolvidas neste tipo de sistema tais como latência de rede, perda de mensagens e heterogeneidade de processamento tornam esta tarefa não trivial. Os *detectores de falhas* (Tel 2000) são as entidades de software responsáveis por esta tarefa. A principal função de um detector é identificar um determinado conjunto de falhas e comunicar a aplicação sobre a detecção. Os detectores estão presentes em vários dos mais importantes algoritmos de tolerância a faltas em sistemas distribuídos tais como *Leader Election* e *Atomic Commitment* (Tel 2000). Em geral, os detectores são implementados como um conjunto de processos distribuídos que se comunicam para chegar a uma decisão sobre a ocorrência de

falha em um processo do sistema. Existem diferentes implementações de detectores conforme a aplicação, desta forma, os protocolos de comunicação, os tipos de falhas identificadas e os algoritmos para a tomada de decisão são exemplos de pontos de variação.

Uma das principais contribuições dos detectores acontece sob o prisma da Engenharia de Software. De fato, os detectores permitem que as aplicações deleguem os detalhes dos algoritmos de detecção de falhas para os detectores e permaneçam em um nível mais alto de abstração. Desta forma, de maneira análoga às linguagens de programação de alto nível em relação às linguagens de máquina, os detectores podem ser vistos como uma camada de abstração sobre os problemas de detecção de algumas classes de falhas melhorando, assim, o entendimento e a construção de aplicações mais complexas.