4
Estudo de Caso

Com o propósito de melhor apresentar a arquitetura NeMaSA, assim como melhor ilustrar seu funcionamento, dois exemplos práticos de testes desenvolvidos sobre a arquitetura proposta serão apresentados neste capítulo. O primeiro teste é bem simples no que diz respeito à sua funcionalidade, mas ilustra bem a característica descentralizada da solução proposta. Além disto, este primeiro teste representa um tipo de teste comumente aplicado em redes reais devido à sua simplicidade. Por conta disto, sua apresentação é relevante para a presente discussão. O segundo teste é bem mais complexo, e ilustra a flexibilidade da solução proposta em cenários em que informações de topologia de rede são necessárias para o diagnóstico de uma falha.

4.1
Coleta de Estatísticas

Uma das atividades mais comuns no contexto de diagnóstico de falhas em redes de telecomunicações é o monitoramento de estatísticas coletadas por equipamentos de redes e mantidas nos mesmos. É comum haver em tais equipamentos contadores (ou acumuladores) para cada interface de rede (porta). Tais contadores mantêm diversos tipos de métricas tais como o número de *bytes* e pacotes trafegados, o número de erros detectados, o número de pacotes descartados, entre outros. Em alguns casos também é possível obter dados não diretamente relacionados ao tráfego de dados, mas sim relacionados ao funcionamento do equipamento propriamente dito, como por exemplo uso de CPU e memória. Tais informações podem ser utilizadas para a identificação de falhas ou outras situações anormais na rede. Um simples exemplo seria o monitoramento de erros detectados em uma interface de rede. Neste caso, uma taxa de erros acima de um determinado limite pode ser evidência de um problema grave tal como ruído na linha.

O primeiro teste selecionado para demonstrar o funcionamento da arquitetura NeMaSA é um teste de monitoramento de estatísticas de interfaces de rede. Tais estatísticas são diretamente coletadas nos equipamentos e comparadas a um valor limite escolhido pelo usuário. Caso o limite estabelecido seja ultrapassado, o teste pode executar um procedimento qualquer como o envio de um alerta aos operadores

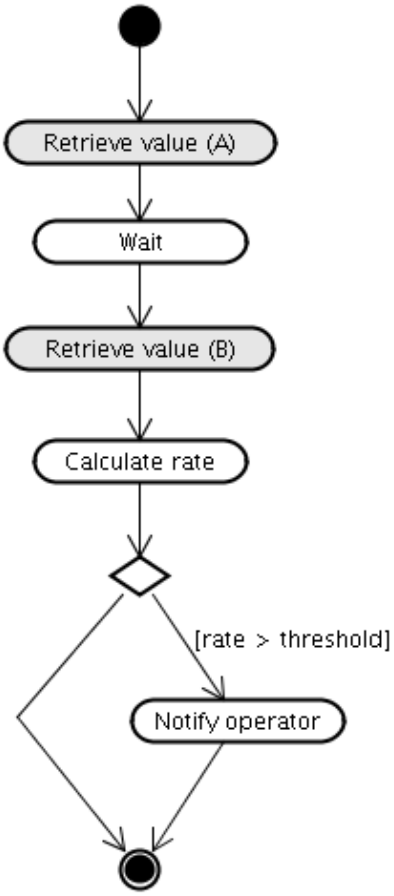

Figura 4.1: Representação gráfica do teste de erros de transmissão

de rede ou mesmo alguma ação corretiva como uma mudança de configuração do equipamento, por exemplo. A discussão seguinte ilustra como este teste pode ser modelado e implementado com a arquitetura NeMaSA.

Este simples tipo de teste necessita de apenas um tipo de tarefa. Esta tarefa é responsável por buscar no equipamento em questão o valor do contador que mantém a métrica de interesse. Neste caso, a métrica de interesse é o número de erros detectados em uma interface de rede (ou porta). Como esta tarefa não altera o estado do equipamento, ela pode ser classificada como um sensor. A lógica do teste é bem direta. O valor do contador é obtido em dois momentos distintos, separados por um intervalo de tempo pré-determinado. A taxa de erros é calculada e, caso seja maior que o limite definido pelo usuário, o teste termina com um diagnóstico de erro. Caso contrário, o diagnóstico indica uma situação normal. A Figura 4.1 ilustra graficamente a lógica do teste. Os elementos em tom escuro indicam a tarefa nos dois momentos descritos acima.

Para implementar este teste na arquitetura proposta, é necessário que se desenvolva duas classes principais: uma para representar as meta-informações sobre o teste, e outra para representar o algoritmo de diagnóstico do mesmo. As

Tabela 4.1: Meta-informações do teste de erros de transmissão

Meta-informação	Valor
Nome	"Transmission Errors Test"
Tipo de recurso de rede	Porta
Parâmetros de entrada	Valor limite (*threshold*) da taxa de erros; e porta (parâmetro implícito)

meta-informações necessárias são (i) o nome do teste, (ii) o tipo de recurso de rede sobre o qual este teste pode ser aplicado e (iii) o conjunto de parâmetros de entrada necessários para a sua execução. O parâmetro de entrada que define o recurso de rede a ser testado tem um tratamento especial e, por isso, não precisa ser declarado no conjunto de parâmetros de entrada do teste. De acordo com a API proposta, as meta-informações do teste devem ser definidas em uma classe que extenda a classe abstrata `Test`. O algoritmo do teste, por sua vez, deve ser representado em uma classe que implemente a interface `TestCommand`, definida pela API da arquitetura. Esta interface define apenas um método, que tem como parâmetro de entrada um `Test` e retorna um `TestResult`. Opcionalmente, pode-se construir uma única classe que extenda a classe `Test` e implemente a interface `TestCommand`, satisfazendo os dois requisitos acima apresentados.

As meta-informações do teste em questão estão expostas na Tabela 4.1. A Figura 4.2 exibe a listagem da classe `TransmissionErrorsTest` que representa o teste em questão. Esta classe contém tanto as meta-informações do teste quanto o seu algoritmo de diagnóstico.

Por fim, é também interessante discutir o comportamento da aplicação quando da execução deste teste. Em particular é interessante discutir o comportamento dos agentes e as trocas de mensagens entre os mesmos para a execução do teste em discussão. A execução deste teste se inicia com a solicitação por parte de um usuário do diagnóstico da taxa de erros de uma porta qualquer de sua escolha. Esta solicitação é feita através de uma interface gráfica ilustrada pela Figura 3.17. A partir desta solicitação o agente de usuário passa a agir no sistema em nome do usuário com o objetivo de que o teste seja executado. Sua primeira ação é solicitar ao agente DF um agente de teste que possa de fato executar o teste. Caso não haja um agente de testes disponível, o agente de usuário solicita ao agente de topologia responsável pela partição em que se encontra o nó a criação de um novo agente de teste. Uma vez definido o agente de teste responsável pela execução do teste em questão, o agente de usuário envia uma mensagem com a solicitação da execução do teste assim como os dados necessários, como as meta-informações do teste e os valores dos parâmetros de entrada. O agente de teste, então inicia o comportamento de execução de teste que chama o método `execute` da interface `TestCommand`

```
public class TransmissionErrorsTest extends Test implements TestCommand {

  public TransmissionErrorsTest() {
    super("Transmission Errors Test");
  }

  public ResourceType getApplicableResourceType() {
    return ResourceType.PORT;
  }

  protected TestParametersList initTestParametersList() {
    TestParametersList parameters = new TestParametersList();
    parameters.addTestParameter(new TestParameter("Error threshold"));
    return parameters;
  }

  public TestCommand getTestCommand() {
    return this;
  }

  public TestResult execute(Test test) {
    Port port = (Port) this.getResource();
    double threshold = Double.parseDouble(test.getTestParameters()
        .getTestParameter("Error threshold").getValue());

    RetrieveErrorCountFromPortTask task = new
        RetrieveErrorCountFromPortTask(port);

    double valueA = task.getErrorCount();

    try {
      Thread.sleep(10 * 1000);
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }

    double valueB = task.getErrorCount();

    double rate = (valueB - valueA) / 10.0;

    TestResult testResult = new TestResult();
    if (rate > threshold) {
      testResult.setMessage("Transmission errors are over threshold");
      testResult.setLevel(Level.ERROR);
    } else {
      testResult.setMessage("Transmission errors are below threshold");
      testResult.setLevel(Level.OK);
    }

    return testResult;
  }
}
```

Figura 4.2: Listagem da classe `TransmissionErrorsTest`

descrita acima. Quando este método acaba, o objeto `TestResult` é transmitido de volta ao agente de usuário através de uma mensagem. O agente de usuário, de posse deste resultado, exibe ao usuário as mensagens resultantes do teste em uma interface gráfica. Este processo está visualmente representado na Figura 3.18.

Um aspecto interessante da arquitetura proposta é evidenciado pelo exemplo acima exposto. Este aspecto é o mecanismo distribuído para a execução de testes. Conforme pode ser depreendido da discussão acima, o processo de execução de um teste pode ser dividido entre diversas entidades que estejam fisicamente distribuídas pela rede. Esta distribuição pode ser presenciada neste exemplo de teste em pelo menos dois momentos distintos. Primeiramente, não há restrições de onde o usuário e o agente de usuário devam estar para interagir com o sistema. Em segundo lugar, o agente de teste pode estar fisicamente localizado em uma máquina próxima ao nó sendo testado. Esta proximidade permite uma menor latência nas comunicações entre o agente de teste e o equipamento, assim como contribui para um menor tráfego pela rede de dados resultantes de solicitações de tarefas que fazem parte de algoritmos de testes. O único momento em que há um comportamento centralizado é quando um agente de usuário solicita a um agente de topologia a criação de um agente de teste. Porém, uma vez definido o agente de teste responsável pela execução de testes em um nó, toda a ação do sistema para a execução de testes ocorre de forma independente das entidades centralizadas, a saber, o agente central (e o repositório centralizado que gerencia) e os agentes de topologia.

4.2 Fonte de Sincronismo

O segundo exemplo prático de teste discutido neste capítulo é significativamente mais complexo que o primeiro. Enquanto o primeiro exemplo de teste serviu para ilustrar o mecanismo básico para a implementação de testes na arquitetura proposta, este segundo exemplo de teste ilustra a dinâmica de um teste em que informações de topologia da rede são necessárias para o diagnóstico de uma falha na rede.

Este segundo exemplo prático de teste, que faz uso de informações de topologia, tem como objetivo verificar o sincronismo do relógio (*clock*) de um equipamento de rede. Muitos equipamentos de telecomunicações de maior porte devem ajustar seu relógio interno de acordo com alguma fonte de sincronismo externa para garantir seu correto funcionamento [Bregni02]. Tal fonte de sincronismo geralmente é um receptor de GPS [GPS]. Também é possível, entretanto, que o sincronismo seja obtido de um outro nó da rede que esteja devidamente sincronizado. Para tanto, este deve estar direta ou indiretamente (através de outros nós) conectado a um receptor de GPS, o que acaba formando uma

"cadeia de sincronismo", que começa no nó em questão e vai até o receptor de GPS, passando por nós intermediários. O exemplo mais comum de falha neste contexto é a perda de conectividade com a fonte de sincronismo. O diagnóstico desta situação é relativamente complexo, pois ele exige a identificação da cadeia de sincronismo do nó em questão e de todos os nós a ela pertencentes. O que torna este problema ainda mais interessante é que um nó dificilmente possui uma referência direta ao nó seguinte da cadeia de sincronismo. Geralmente há apenas uma referência à porta que liga os dois nós. Para que se descubra o outro nó, é preciso identificar o cabo conectado a esta porta e, só então, o nó na outra ponta do cabo. Este é, portanto, um exemplo de teste em que a informação de topologia é essencial para a obtenção do diagnóstico de falhas.

Há pelo menos três condições de anormalidade que se deseja detectar em relação ao sincronismo de relógios de equipamentos de telecomunicações. Para uma melhor compreensão deste problema, entretanto, uma breve explanação sobre o comportamento de tais equipamentos se faz necessária. Geralmente, equipamentos de telecomunicações de grande porte são projetados de forma a serem capazes de se manterem corretamente sincronizados por um certo intervalo de tempo mesmo quando a cadeia de sincronismo é rompida. Esta situação é denominada *Hold Over*, e não caracteriza uma falha uma vez que o sincronismo ainda é mantido. Entretanto, tal situação exige alguma forma de intervenção para que uma situação de normalidade seja restabelecida. Caso esta condição seja mantida indefinidamente, sem que a conexão com uma fonte confiável de sincronismo seja restabelecida, o tempo máximo de sincronismo garantido pelo nó se esgota e o mesmo parte para um estado denominado *Free Run*. Nós nesta condição estão, para efeitos práticos, dessincronizados, o que pode acarretar falhas no tráfego de dados entre nós adjacentes. Tal condição sinaliza uma falha grave na rede que deve ser rapidamente corrigida. Uma terceira condição de anormalidade ocorre quando há a presença de laços (*loops*) na cadeia de sincronismo. Isto ocorre, por exemplo, se um nó A tem um nó B como sua fonte de sincronismo, o nó B depende de um nó C e, por fim, o nó C obtém de A seu sincronismo. Em casos como este, todos os nós podem ser considerados dessincronizados apesar de não estarem conscientemente no modo *Free Run*.

O algoritmo para a detecção de tal falha precisa percorrer a cadeia de sincronismo desde o nó original até encontrar uma fonte confiável, um nó em *Hold Over* ou um nó em *Free Run*. Estes casos indicariam, respectivamente, uma situação normal, um alerta e um erro. Além disto, este algoritmo também precisa manter uma lista de todos os nós da cadeia percorridos até então para poder detectar a existência de laços. Caso algum nó seja percorrido mais de uma vez, caracteriza-se a existência de um laço, que é um erro grave. A Figura 4.3 ilustra graficamente este

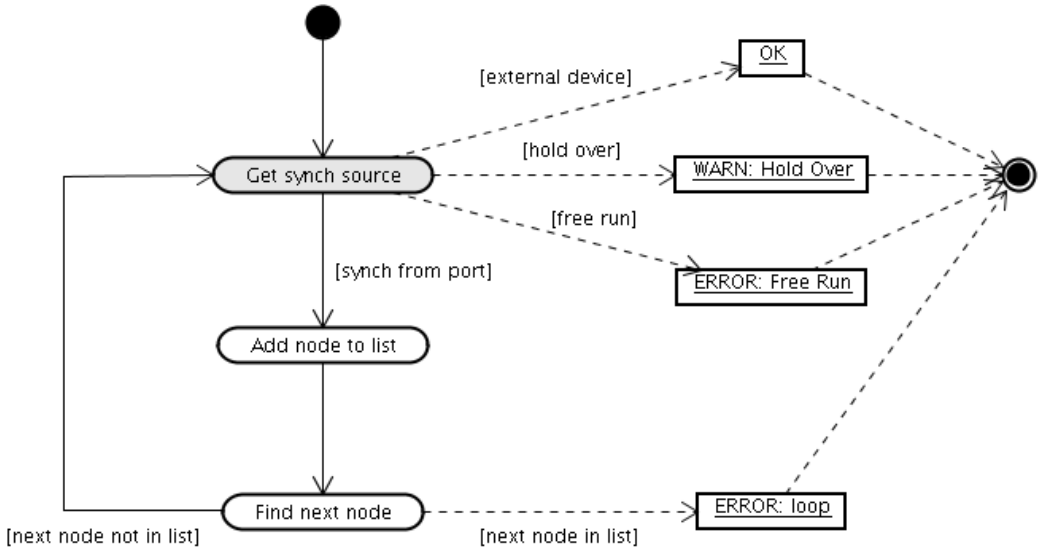

Figura 4.3: Representação gráfica do teste de sincronismo

algoritmo. Há uma tarefa (em tom escuro) que busca no equipamento a sua atual fonte de sincronismo. O resultado desta tarefa pode ser (i) sincronismo proveniente de dispositivo externo (GPS), (ii) nó em modo *Hold Over*, (iii) nó em modo *Free Run*, ou (iv) sincronismo proveniente de outro nó. Neste último caso, há a indicação da porta por onde vem o sincronismo. Sempre que esta quarta situação for encontrada, é necessário se descobrir o nó remoto ligado à porta em questão. Este é o nó seguinte na cadeia de sincronismo. Para que tal informação seja obtida, deve-se consultar uma base com a topologia da rede.

A Figura 4.4 exibe a listagem de como tal algoritmo pode ser implementado com a API fornecida pela arquitetura. A organização básica desta classe segue as mesmas características da classe do teste simples discutida anteriormente. Há algumas diferenças entre ambas, entretanto, e alguns comentários adicionais sobre esta implementação fazem-se necessários. Primeiramente, o tipo de recurso sobre o qual este teste atua é diferente. No caso anterior, o teste atuava sobre portas. Este atua sobre nós. Em segundo lugar, este teste não espera nenhum parâmetro de entrada além do recurso sobre o qual atua (nó). Por isto, ele retorna uma lista de parâmetros de entrada vazia. Em terceiro lugar, como este teste possui uma característica cíclica em seu comportamento, o algoritmo do teste é implementado através de chamadas recursivas ao método privado `executeTestOnNode`, que é invocado para cada nó pertencente à cadeia de sincronismo. Por fim, a característica determinante deste teste, que é a necessidade de consultas a informações de topologia é evidenciada pelo uso da interface `TopologyDao`. Uma implementação desta interface é passada pelo agente de teste para o objeto que implementa o teste. Nesta implementação, o agente de teste cria uma mensagem baseada no método invocado pelo teste assim como os parâmetros do método. Esta mensagem é enviada

```
public class SynchSanityTest extends Test implements TestCommand {

  private TopologyDao topologyDao;

  private Set<Node> visitedNodes;

  public SynchSanityTest() {
    super("Synchronization Sanity");
  }

  public void setTopologyDao(TopologyDao topologyDao) {
    this.topologyDao = topologyDao;
  }

  public ResourceType getApplicableResourceType() {
    return ResourceType.NODE;
  }

  protected TestParametersList initTestParametersList() {
    return new TestParametersList();
  }

  public TestCommand getTestCommand() {
    return this;
  }

  public TestResult execute(Test test) {
    Node node = (Node) test.getResource();
    visitedNodes = new HashSet<Node>();
    return executeTestOnNode(node);
  }

  private TestResult executeTestOnNode(Node node) {
    RetrieveSynchSourceTask task = new RetrieveSynchSourceTask(node);
    SynchSourceType synchSourceType = task.getSynchSourceType();

    switch (synchSourceType) {
    case EXTERNAL:
      return new TestResult("Node is correctly synchronized", Level.OK);

    case HOLD_OVER:
      return new TestResult("Node is in Hold Over mode", Level.WARNING);

    case FREE_RUN:
      return new TestResult("Node is in Free Run mode", Level.ERROR);

    case PORT:
      visitedNodes.add(node);

      Port localPort = task.getSynchSourcePort();
      Cable cable = topologyDao.getCableAtPort(localPort);
      Port remotePort = cable.getEndA();
      if (remotePort.equals(localPort)) {
        remotePort = cable.getEndB();
      }
      Node remoteNode = remotePort.getNode();

      if (visitedNodes.contains(remoteNode)) {
        return new TestResult("Loop found at synchronization chain",
            Level.ERROR);
      } else {
        return executeTestOnNode(remoteNode);
      }

    default:
      throw new RuntimeException("This should never happen");
    }
  }
}
```

Figura 4.4: Listagem da classe `SynchSanityTest`

pelo agente de teste para o agente de topologia correspondente. Assim que o agente de teste recebe a resposta do agente de topologia, esta mensagem é interpretada e passada de volta à execução do teste com o retorno do método invocado pelo teste. Este processo é ilustrado na Figura 3.15.

Como pode ser observado nos dois exemplos de testes apresentados neste capítulo, a arquitetura proposta permite que a implementação de testes seja bem simples, mesmo quando informações de topologia são necessárias para o diagnóstico de falhas. Há, entretanto, um problema com a forma com que o segundo teste foi implementado. Uma das vantagens da arquitetura proposta levantada na discussão do primeiro exemplo de teste foi que havia uma proximidade geográfica entre o agente de teste e os nós sobre os quais ele atuava. Da forma como este segundo teste foi implementado, esta proximidade geográfica não pode mais ser garantida, uma vez que o teste executa tarefas sobre todos os nós pertencentes à cadeia de sincronismo do nó sob diagnóstico, e não há garantias que estes nós estejam geograficamente próximos. Na prática esta proximidade é esperada, visto que não é comum criar cadeias de sincronismo de relógios com equipamentos muito distantes entre si. De qualquer maneira, a solução da arquitetura em seu estágio atual não é a melhor possível. Uma alternativa mais interessante seria deslocar a execução do teste para outro agente de teste mais próximo dos nós, mais este mecanismo não foi implementado na presente versão da arquitetura.