Universidade Federal da Paraíba
Centro de Ciências e Tecnologia
Departamento de Sistemas e Computação
Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Evolução de um *Framework* para a Construção de Aplicações de Gerência de Falhas em Redes de Computadores

Giovanni Almeida Santos

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba – Campus II, como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Redes de Computadores

Jacques Philippe Sauvé
(orientador)

Campina Grande, Paraíba, Brasil

Agosto de 2001

SANTOS, Giovanni Almeida

S237E

Evolução de um *Framework* para a Construção de Aplicações de Gerência de Falhas em Redes de Computadores

Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Brasil, agosto de 2001.

102p. Il.

Orientador: Jacques Philippe Sauvé

1. Redes de Computadores
2. Gerência de Redes
3. Evolução de *Frameworks*

CDU - 621.391

Resumo

Atualmente, devido ao fato de as redes de computadores estarem tornando-se indispensáveis para as instituições, é imprescindível que se faça um gerenciamento eficiente sobre tais redes, com o intuito de se minimizar a ocorrência de falhas. Por outro lado, com o crescente aumento no tamanho dessas redes, as soluções de gerência centralizada existentes não são escaláveis para gerenciar estas redes eficientemente. Este trabalho apresenta detalhes sobre a implementação e a validação do *framework* fwGF, um *framework* para a construção de aplicações de gerência de falhas. Também é efetuada a evolução deste, com base em um processo de evolução de frameworks, com o objetivo de prover as adaptações necessárias para que o fwGF possa dar suporte à construção de aplicações de gerência de falhas distribuídas.

Abstract

Nowadays, as computer networks become critical to enterprise, efficient network management is necessary, with a main goal of minimizing the occurrence of faults. On the other hand, due the increase in network size, centralized management applications are not scalable to manage these networks efficiently. This dissertation details a fault management framework (fwGF), its implementation and validation. We also discuss the evolution of the framework, based on a framework evolution process, to support distributed network management.

Agradecimentos

Quero deixar registrado meus sinceros agradecimentos ao meu orientador, Prof. Jacques Sauvé, pelo seu inestimável apoio e pela sua confiança no meu trabalho. Sua orientação foi fundamental durante todo esse tempo.

Ao Prof. Pedro Sérgio Nicolletti, sempre disposto a me atender. Aprendi muito com nossas conversas.

A todos da minha família, principalmente a minha esposa, Viviane, pelo apoio que sempre me deram.

A todos os meus amigos do DSC, alunos, funcionários e professores, pelo companheirismo que sempre tivemos.

"Dificuldades reais podem ser
resolvidas; apenas as imaginárias são
insuperáveis."

Theodore N. Vail

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

A utilização de redes de computadores nas organizações (acadêmicas, industriais, comerciais, etc.) tem crescido enormemente nos últimos tempos, principalmente a partir da popularização da Internet, sendo difícil, hoje, encontrar um departamento ou setor de uma organização que não faça uso de recursos ou serviços disponibilizados através de uma rede. Muitos dos sistemas utilizados pelas organizações estão disponibilizados através de suas redes, inclusive aqueles de missão-crítica, ou seja, aqueles que são fundamentais para a sobrevivência da organização, como por exemplo, serviços de vendas, de controle de produção e de estoque, de distribuição, de apoio à tomada de decisão, entre outros. Além disso, as redes de comunicação não mais englobam apenas usuários internos à organização, mas também usuários externos como clientes, parceiros, fornecedores, etc., que têm acesso a ela de forma remota.

Consequentemente, as redes de comunicação têm se tornado cada vez mais importantes para as organizações, sendo um fator indispensável para a sua sobrevivência. Qualquer falha em seu funcionamento pode comprometer a disponibilidade de algum serviço, podendo, então, causar enormes prejuízos.

Portanto, é necessário se fazer um gerenciamento eficiente dos recursos da rede no intuito de mantê-la funcionando de forma eficiente, garantindo, assim, a disponibilidade dos recursos e atendendo às necessidades dos usuários. Para isso, é preciso ter uma forma de se gerenciar a rede de forma a poder descobrir possíveis problemas e solucioná-los o quanto antes.

Ultimamente, tem-se procurado automatizar o máximo possível a tarefa de gerência de redes de computadores, pois o gerenciamento *ad hoc* (através de programas como *Ping* - teste de conectividade entre dois dispositivos - ou *Traceroute* - análise de rota entre dois dispositivos) não é mais viável. Isto se deve ao fato de que estas não são soluções escaláveis ou abrangentes em termos de detecção e solução de problemas (imagine gerenciar uma rede com milhares de equipamentos utilizando-se estes programas!).

Muitas aplicações de gerência têm surgido nos últimos anos, cada uma, geralmente, mais ligada a uma área específica da gerência de redes - configuração, desempenho, falhas, segurança e contabilidade. Assim, as aplicações devem configurar os equipamentos da rede, fazer análise de desempenho, detectar a ocorrência de falhas. Tudo para que a rede possa ser gerenciada da forma mais eficiente possível.

Entretanto, desenvolver aplicações destinadas ao gerenciamento de redes não é uma tarefa fácil. As plataformas de gerência, *APIs* (*Application Programming Interfaces*) e linguagens de programação disponíveis não oferecem as devidas abstrações para que um programador possa construir tais aplicações sem que tenha que se envolver com outros detalhes, como por exemplo, estruturas de dados complexas, características pertinentes ao protocolo de gerência utilizado, forma como as informações de gerência estão armazenadas e maneira de ter acesso a elas, etc. Tais aspectos estão ligados mais com programação do que propriamente com o domínio do problema.

Sendo assim, é importante o desenvolvimento de software reutilizável como forma de promover o aumento de produtividade e de qualidade na construção de tais aplicações (produtos). O objetivo é sempre desenvolver produtos de software com características genéricas e extensíveis para que possam ser reutilizadas, mediante as devidas extensões, na construção de outros produtos de software. Dessa forma, um programador não precisa iniciar a construção de uma aplicação sempre do "zero", mas pode reutilizar partes de software (construídos por ele ou por outros) e fazer as devidas adaptações de forma a atender às suas necessidades. Isto faz com que o tempo gasto para se desenvolver novos produtos de software diminua, pelo menos amortizando os custos ao longo do desenvolvimento de vários produtos. Além disso, os produtos resultantes são mais confiáveis, tendo em vista que as partes reutilizadas geralmente já se encontram testadas.

Uma das técnicas que tem sido bastante utilizada para a construção de software reutilizável é a técnica de *framework orientado a objeto* (*framework OO* [Roberts & Johnson, 1996], [Rogers, 1997], [D'Souza e Wills, 1999]). Um *framework OO*[1] é um sistema de software (ou parte dele) que é composto por um conjunto de classes (concretas e abstratas) que colaboram entre si, de forma a compor um projeto reutilizável para um domínio de problema específico [Rogers, 1997]. Sendo assim, um *framework* é uma aplicação quase completa, permitindo que o seu usuário complemente-o com as partes que estão faltando de acordo com a solução por ele projetada.

Frameworks fornecem um dos mais altos graus de reusabilidade, tendo em vista que eles capturam as funcionalidades comuns a várias aplicações de um mesmo domínio de problema, proporcionando, assim, a reutilização não apenas de código, mas também de análise e projeto, fases estas consideradas as mais complexas e que consomem mais tempo

[1] No restante deste documento, utilizar-se-á o termo *framework* no sentido de *framework OO*.

durante o processo de desenvolvimento de software. *Frameworks* fazem largo uso de padrões de projeto (*design patterns*) [Gamma *et al.*, 1995].

Entretanto, desenvolver *frameworks* não é uma tarefa simples [Roberts & Johnson, 1996]. Eles exigem do projetista um grande poder de abstração para efetuar generalizações sobre o domínio de problema ao qual elas se destinam. Normalmente, até o *framework* atingir a "maturidade" requer-se tempo e experiência por parte dos seus projetistas.

Contudo, um *framework* possui um ciclo de vida próprio, constituído de etapas que são desenvolvidas ao longo do tempo, promovendo, assim, sua evolução gradual [Roberts & Johnson, 1996]. É importante salientar que o ciclo de vida de um *framework* é completamente diferente do de outros tipos de produtos de software.

Com relação à utilização de *frameworks*, estes estão se tornando muito importantes no processo de desenvolvimento de software. Dentre as mais variadas aplicações de *framework* podem ser citadas: a construção de aplicações, o projeto de interfaces gráficas (*GUI – Graphical User Interface*), editoração gráfica, plataformas de gerência de rede, dentre outras.

Em [Freire, 2000], encontra-se um exemplo de utilização de ***F**rame**W**ork* para a construção de aplicações para **G**erência de **F**alhas em redes de computadores (fwGF). O fwGF tem como objetivo prover, ao programador deste tipo de aplicações (o usuário do fwGF), transparência de detalhes de mais baixo nível, ligados à programação, como por exemplo, estruturas de dados complexas, protocolos de gerência, tratamento de erros, etc. Dessa forma, o programador dispõe de abstrações mais adequadas para construir aplicações apenas envolvendo-se com o domínio do problema em si, sem se preocupar com os detalhes citados anteriormente.

Em [Freire, 2000] é feito um levantamento dos requisitos funcionais e não funcionais que uma ferramenta deve possuir para poder prover as devidas abstrações ao desenvolvedor. Portanto, com base nesse levantamento, a solução dada pelo fwGF é muito interessante, no sentido em que busca fornecer um nível de abstração mais adequado para o desenvolvimento de aplicações de gerência de falhas, já que torna transparente para o programador as peculiaridades de baixo nível e oferece as abstrações necessárias para que o mesmo possa envolver-se mais com o domínio do problema específico.

Entretanto, o fwGF é baseado no **paradigma de gerência centralizada** [Goldszmidt & Yemini, 1995b], ou seja, ele é baseado em uma arquitetura em duas camadas,

onde uma representa a estação de gerência (gerente) que coleta as informações e a outra os elementos gerenciados (agentes).

Devido ao crescimento no tamanho e na complexidade das redes de computadores, soluções baseadas neste paradigma têm enfrentado problemas de **escalabilidade**, ou seja, tornam-se ineficientes quando o gerenciamento é efetuado em uma rede com muitos dispositivos, por exemplo.

O problema com a gerência centralizada decorre do fato de que a estação de gerência (*NMS – Network Management Station*) centraliza toda a atividade de gerência, desde a requisição de dados dos agentes (elementos gerenciados), passando pelo processamento de tais dados para se detectar problemas na rede, até a geração das informações que serão repassadas ao administrador da rede. Dessa forma, muito tráfego dentro da rede é gerado devido às requisições de informações pela NMS aos agentes. Além disso, a NMS necessita de um alto poder de processamento para poder avaliar todas as informações que chegam até ela e, assim, poder efetuar um gerenciamento eficiente da rede.

Portanto, a necessidade de um gerenciamento eficaz para redes grandes e complexas tem feito com que algumas tecnologias tenham surgido ao longo dos últimos anos, de forma a se poder transferir parte da funcionalidade e do processamento realizado na estação de gerência para outras partes da rede, ficando cada uma destas responsável pelo gerenciamento de uma "parte" da rede.

Dessa forma, um novo paradigma tem surgido, o **paradigma de gerência distribuída** [Goldszmidt & Yemini, 1995b], no qual todas as tarefas não estão concentradas apenas no gerente, mas sim, são divididas entre gerente e agentes. Tais agentes, então, não mais funcionam apenas como coletores de dados, mas também possuem um conhecimento sobre atividades de gerenciamento através das quais eles podem gerenciar um dispositivo (ou um conjunto deles). Conforme encontrado em [Goldszmidt & Yemini, 1995b], isto significa levar as funções de gerenciamento até os dados ao invés de mover os dados até às funções.

Com isso, parte dos problemas surgidos em uma rede podem ser solucionados pelos próprios agentes, já que os mesmos são dotados de "inteligência" para tal. Além disso, ao invés de um gerente coletar "dados brutos", os agentes podem efetuar um processamento nos dados e passar ao gerente informações mais ricas semanticamente.

Tal paradigma, portanto, além de escalabilidade, oferece flexibilidade e robustez.

Conforme o exposto acima, dois problemas podem ser então identificados:

1. **o *framework* fwGF é promissor, mas é só uma proposta. Sendo assim, precisa-se ainda validá-lo e evolui-lo ao longo do ciclo de vida;**
2. **não é baseado no paradigma de gerência distribuída.**

Na seção a seguir, serão apresentados os objetivos deste trabalho.

1.1. Objetivos da Dissertação

Conforme visto acima, as plataformas, APIs e linguagens de programação não oferecem ao programador de aplicações de gerência abstrações adequadas para que o mesmo possa concentrar-se apenas no domínio do problema para o qual ele deseja desenvolver sua aplicação, fazendo com que ele tenha que obter um conhecimento mais aprofundado sobre aspectos de programação alheios ao domínio em si.

Em [Freire, 2000], há uma solução baseada em *frameworks* que esconde os detalhes de baixo nível associados ao desenvolvimento de aplicações de gerência de falhas de redes de computadores, proporcionando ao programador abstrações mais adequadas para o desenvolvimento de tais aplicações. Entretanto, assim como as outras soluções relacionadas em [Freire, 2000], ela se baseia no paradigma de gerência centralizada

Portanto, o objetivo principal do trabalho proposto neste documento é implementar e validar o fwGF e efetuar a **evolução** deste, com base no processo de evolução de *frameworks*, proposto em [Roberts & Johnson, 1996], de forma a prover as classes e funcionalidades necessárias para que tal solução possa dar suporte ao **paradigma de gerência distribuída** de redes num nível de abstração adequado.

Em linhas gerais, este processo de evolução pode ser caracterizado pelas seguintes etapas:

- **Implementar/Verificar o *framework* especificado em [Freire, 2000]**: como o *framework* fwGF encontra-se apenas com seus componentes especificados, o trabalho iniciar-se-á na implementação destes. A verificação do fwGF se dará através da construção de testes de unidades;

- **Validar o *framework***: após a etapa de implementação, será realizada a etapa de validação do fwGF. Nesta fase, o objetivo principal é determinar se o *framework* implementado é um "software válido", ou seja, especificada uma aplicação de gerência de falhas, se é possível que, com a utilização deste *framework,* se possa construir tal aplicação de forma que os seus requisitos originais sejam satisfeitos;
- **Propor uma adaptação ao *framework* para a construção de aplicações para gerência de falhas distribuída**: após o término da etapa anterior, serão efetuadas as adaptações necessárias ao fwGF para que o mesmo dê suporte à construção de **aplicações de gerência de falhas distribuída**.

1.2. Escopo e Relevância

Com base nos objetivos propostos, neste trabalho é apresentada a evolução do *framework* fwGF com base no processo de evolução descrito em [Roberts & Johnson, 1996]. Como já dito anteriormente, desenvolver um *framework* não é uma tarefa simples. Entretanto, um *framework* possui um ciclo de vida próprio ao longo do qual o mesmo passa por várias etapas, evoluindo à medida em que as mesmas ocorrem.

No que diz respeito à validação do fwGF, optou-se por escolher um subconjunto abrangente dentro do espaço de aplicações possíveis de gerência de falhas em redes de computadores. Dessa forma, o fwGF será utilizado para a construção das aplicações desse subconjunto e, ao final, serão deduzidas as conclusões sobre a validade deste *framework*, apontando-se os seus pontos positivos e negativos.

Com relação à adaptação do fwGF para prover suporte à gerência de falhas distribuída, os requisitos funcionais e não-funcionais descritos em [Freire, 2000] continuam sendo seguidos, de forma a sempre prover abstrações de alto nível para o usuário (programador de aplicações) do fwGF.

A importância deste trabalho está em se prover o desenvolvimento de um *framework* que dê suporte à gerência de falhas distribuída, a partir do qual possam ser desenvolvidas aplicações de gerência escaláveis (que podem ser utilizadas para se gerenciar grandes redes) e robustas (cujas falhas ocorridas em qualquer estação de gerência não afetem o processo de gerenciamento da rede).

1.3. Estrutura da Dissertação

No capítulo 2, é apresentada uma breve descrição do fwGF e de suas características fundamentais, a fim de que o leitor possa se familiarizar com o mesmo.

No capítulo 3, são apresentados os detalhes relativos à etapa de implementação do fwGF. Neste capítulo não é mostrada a implementação de cada componente do fwGF, mas sim, são mostrados aspectos considerados relevantes como a aplicação de padrões de projeto, a construção de testes de unidade, entre outros.

No capítulo 4, é apresentada a etapa de validação do fwGF, no qual é especificado um conjunto de aplicações de gerência de falhas a serem construídas com este *framework* Para cada aplicação, será descrita seus objetivos, os componentes do fwGF a serem instanciados e se a construção foi bem sucedida ou não. Ao final desse capítulo, serão efetuadas as conclusões sobre a validade do fwGF.

No capítulo 5, serão apresentados os detalhes referentes à etapa de evolução do fwGF, a qual é baseada no processo descrito em [Roberts & Johnson, 1996] e tem por objetivo acrescentar ao fwGF as características necessárias para que o mesmo possa ser utilizado na construção de aplicações que dêem suporte à gerência de redes distribuída.

Por fim, no capítulo 6, serão apresentadas as conclusões e as sugestões para futuros trabalhos.

2. Descrição do Framework

Antes de apresentar a evolução do *framework* especificado em [Freire, 2000], convém fazer aqui uma breve exposição do mesmo e de suas características fundamentais, a fim de que o leitor possa se familiarizar com o assunto. Aqueles que já conhecem o *framework* podem passar à leitura dos capítulos seguintes. Adotar-se-á aqui a abreviação fwGF para representá-lo.

O fwGF é um *framework* composto por um conjunto de componentes de software [D'Souza & Wills, 1999] que cooperam entre si constituindo um projeto de software reutilizável aplicado a uma classe específica de problemas, neste caso, à gerência de falhas. Seu objetivo principal é fornecer ao programador de aplicações (o usuário do fwGF) abstrações de alto nível para que o mesmo não tenha que se preocupar com detalhes ligados à programação tais como a manipulação de estruturas de dados utilizadas nos protocolos de gerência e o tratamento de erros de comunicação, entre outros.

Para se construir uma aplicação com o fwGF, primeiramente, é importante definir os seus requisitos de maneira a se determinar quais os componentes que serão utilizados. Em seguida, deve-se selecionar os componentes necessários e conectá-los entre si para "montar" a aplicação. Como cada componente provê uma funcionalidade específica, a funcionalidade da aplicação é dada pelo conjunto de componentes em cooperação.

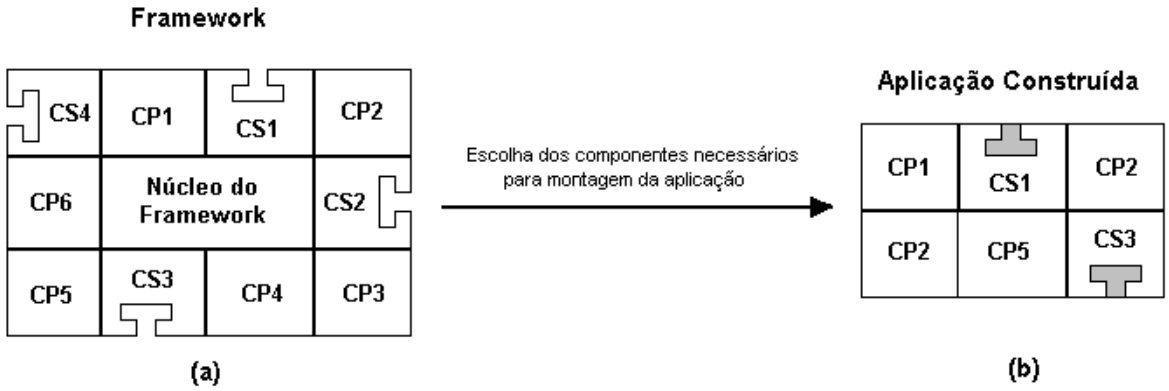


Figura 2.1. Representação dos componentes do *framework*

Na **Figura 2.1 (a)**, pode-se observar a estrutura de um *framework*, o qual possui componentes prontos (CPi) e semi-prontos (CSi), sendo que estes últimos, para serem utilizados, precisam ter sua funcionalidade complementada. Tal complementação deve ser

provida pelo programador de aplicações e é específico de cada aplicação, dependendo dos requisitos da mesma.

Os componentes se *plugam* ainda em um outro pedaço de software, aqui chamado de núcleo do *framework*. Este núcleo, necessariamente, não é um componente (embora também possa ser), podendo também ser composto por classes e interfaces que podem ser utilizadas pelo programador de aplicações no desenvolvimento de novos componentes.

Na **Figura 2.1 (b)** são mostrados os componentes instanciados na montagem de uma aplicação. Observe que foram utilizados aqui componentes semi-prontos do *framework*, os quais tiveram sua funcionalidade complementada de forma a se atender aos requisitos da aplicação. É importante frisar que nem todos os componentes que constituem o *framework* necessariamente precisam participar da construção de uma aplicação.

Nas seções a seguir, são apresentados os componentes que constituem o fwGF. Porém, antes desta descrição, convém apresentar ao leitor, de forma suscinta e de alto nível, como uma aplicação pode ser construída com a utilização de componentes do fwGF.

Considere que se deseja construir uma aplicação de gerência para se verificar se um equipamento da rede (um roteador, por exemplo) está ou não operacional (*up*). Esta é uma aplicação bastante importante tendo em vista que, nos últimos anos, o número de serviços disponibilizados através das redes de computadores vem crescendo muito, o que implica que se deve ter um gerenciamento eficiente sobre estas de forma a minimizar a ocorrência de falhas e manter a disponibilidade de tais serviços durante o máximo de tempo possível.

Portanto, a aplicação acima citada deve monitorar o referido equipamento de tempos em tempos de forma a se verificar o seu estado. Então, no caso de uma falha neste dispositivo ocorrer, vindo a afetar o seu funcionamento, o responsável por tal equipamento deve ser notificado com a máxima urgência, a fim de que esta falha possa ser corrigida o mais rápido possível para que o equipamento possa voltar a funcionar. Sendo assim, esta aplicação pode ser construída, selecionando-se componentes do *framework* e interligando-os de maneira que o conjunto formado por estes proveja a funcionalidade requerida para tal aplicação.

Para se construir esta aplicação, portanto, deve-se utilizar um componente para representar o equipamento a ser gerenciado, aqui chamado de *Roteador*, um para indicar qual a informação que caracteriza o estado do equipamento, chamado *Informação Requisitada*, um *Monitor*, responsável por monitorar o equipamento, coletando a intervalos de tempos regulares, o valor da informação requisitada., um *Detector de Falhas*, que recebe a informação coletada pelo monitor e verifica se o valor da informação coletada indica uma

falha no equipamento e um componente para tratamento de falhas (*Tratador de Falhas*), o qual é encarregado de avisar ao responsável pela administração da rede, caso tenha ocorrido uma falha no equipamento, para que o mesmo tome as devidas providências para corrigi-la. Na realidade, para que esta aplicação fosse mais eficiente ainda, ela deveria prover mecanismos para corrigir a falha automaticamente (ou pelo menos tentar). Entretanto, aqui vamos considerar apenas que o componente de tratamento da falha notifica ao administrador através do envio de uma mensagem de correio eletrônico para o mesmo. Na **Figura 2.2** pode-se observar a comunicação entre os componentes através do repasse de eventos (*pipeline* de eventos) entre os mesmos, ocorrido ao longo do ciclo de vida da aplicação.

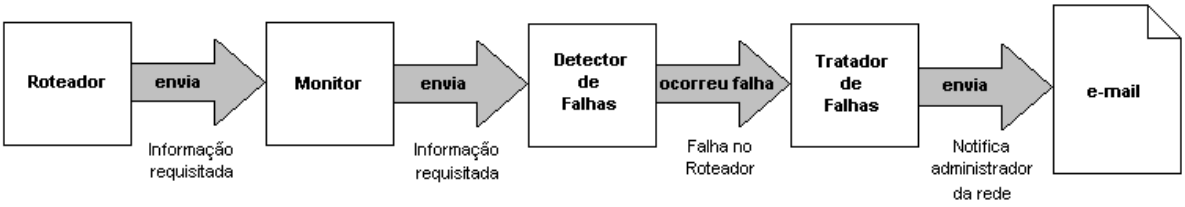


Figura 2.2. Aplicação de monitoração de um roteador: Comunicação entre os componentes

2.1. Classes de Componentes do fwGF

Os componentes que formam o fwGF podem ser divididos em quatro classes:

- *Descrição da Rede Gerenciada*: determina quais dispositivos serão monitorados e as informações de gerência a serem requisitadas dos mesmos;
- *Monitoramento da Rede*: estabelece parâmetros de monitoramento dos dispositivos, como por exemplo, o intervalo de tempo entre as requisições de informações;
- *Identificação de Faltas*: determina os mecanismos utilizados para a detecção de faltas com base nas informações obtidas;
- *Tratamento de Faltas*: a partir das faltas identificadas, os componentes desta classe podem efetuar um tratamento sobre as mesmas, como por exemplo, *correlação de eventos* (algoritmos para correlação podem ser encontrados em [Meira & Nogueira, 1997]), geração de alarmes, etc.

Nas próximas seções, passamos a descrever os componentes do *framework* de acordo com as classes descritas acima.

2.2. Descrição da Rede Gerenciada

Para se descrever a rede gerenciada, o fwGF provê quatro componentes básicos:

- `ElementoGerenciado`: este componente representa o dispositivo a ser gerenciado. Para cada dispositivo uma aplicação de gerência deve instanciar um componente deste tipo;
- `InfoGerencia`: representa uma unidade de informação (ou variável) de gerência a ser coletada do dispositivo gerenciado;
- `GrupoInfoGerencia`: para otimizar a coleta de informações dos dispositivos, cada requisição pode transportar um conjunto de unidades de informação (`InfoGerencia`), as quais devem ser agrupadas neste componente.
- `ConfiguradorDeRequisição`: utilizado para se configurar os parâmetros de uma requisição de informações a um dispositivo gerenciado, como por exemplo, *timeout*, número de retransmissões, etc.;

2.3. Monitoramento da Rede

Para se monitorar periodicamente a rede gerenciada, o *framework* dispõe dos seguintes componentes:

- `Monitor`: para cada equipamento deve-se associar um monitor e informar seu período de monitoração bem como o grupo de informações a ser coletado. Como cada monitor deve rodar em uma *thread* separada, pode-se monitorar os equipamentos da rede concorrentemente;
- `MonitorEvent`: este componente representa um evento que é gerado quando um componente de monitoração requisita informações a um dispositivo gerenciado. Estes eventos são utilizados na comunicação entre componentes de acordo com o padrão de projeto *Observer* [Gamma *et al*, 1995] visto mais adiante;
- `MonitorListener`: representa a interface[2] que todo componente interessado em receber eventos de um `Monitor` deve implementar;

[2] No contexto deste documento, refere-se a uma interface na linguagem Java. É conceitualmente idêntico a um Tipo Abstrato de Dado.

- `ReceptorDeTraps`: utilizado para a monitoração assíncrona da rede. Ele recebe os *traps* gerados na rede e envia eventos para os consumidores (*listeners*) interessados (um *trap* é uma notificação de um agente para uma estação de gerência sobre a ocorrência de alguma situação significativa [Stallings, 1996]. Por sua vez, um agente é um software que provê acesso aos dispositivos da rede e permite que os mesmos sejam gerenciados [Black, 1998]);
- `TrapEvent`: representa um evento que é gerado quando um *trap* é recebido por um `ReceptorDeTraps`;
- `TrapListener`: representa a interface que todo componente interessado em receber eventos de um `ReceptorDeTraps` deve implementar;

2.4. Identificação de Falhas

Com relação à identificação de faltas, os componentes abaixo são fornecidos:

- `GeradorDeEventoFalha`: a partir dos valores das informações de gerência coletadas dos dispositivos, um componente deste tipo pode analisar tais valores (de acordo com os critérios estabelecidos pelo programador da aplicação) e concluir se houve ou não a ocorrência de faltas.
- `GeradorDeEventoFalhaComHisterese`: este é um componente mais específico e que é baseado no mecanismo de histerese, cuja função é limitar a geração de eventos. Através deste mecanismo pode-se configurar dois valores extremos (limiares) para uma variável de gerência. Quando o valor da variável cruza um destes limiares, um evento (`EventoFalhaEvent`) é gerado, sendo que outro evento somente é gerado quando o valor da variável cruza o limiar oposto (**Figura 2.3**);
- `EventoFalhaEvent`: representa um evento que é gerado quando da identificação de uma falta e é utilizado na comunicação entre componentes;
- `EventoFalhaListener`: representa a interface que todo componente interessado em receber eventos de um `GeradorDeEventoFalha` deve implementar;

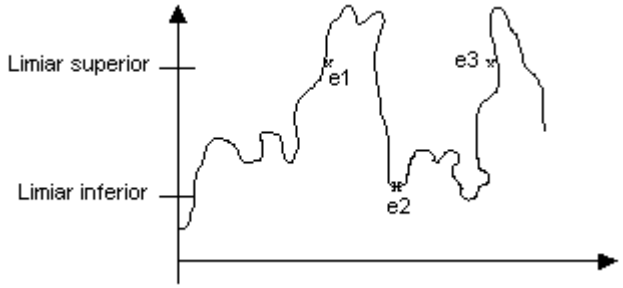


Figura 2.3. Mecanismo de histerese

2.5. Tratamento de Falhas

Para o tratamento de faltas o fwGF fornece os seguintes componentes:

- `CorrelatorDeEventoFalha`: utilizado para se efetuar correlação de eventos. Por exemplo, a partir dos eventos gerados decorrentes de faltas identificadas na rede, pode-se interpretá-los e substituí-los por um novo evento mais rico semanticamente. Este componente herda as características de GeradorDeEventoFalha;
- `CorrelatorDeEventoFalhaCompressão`: implementa o algoritmo de compressão, que consiste em se identificar a ocorrência de eventos equivalentes em um dado intervalo de tempo, substituindo-os por apenas um único com a indicação da quantidade de eventos ocorridos;
- `CorrelatorDeEventoFalhaSupressão`: implementa o algoritmo de supressão, que consiste em se gerar um evento quando o número de ocorrências de um mesmo evento, em um dado intervalo de tempo, atingir um limiar estabelecido;
- `CorrelatorDeEventoFalhaCMP`: é utilizado para um controle de manutenção programada (CMP), como por exemplo, para se evitar que eventos sejam gerados desnecessariamente, em um intervalo de tempo, quando os dispositivos gerenciados estiverem desligados por conta de manutenção.

Vale salientar aqui que alguns desses componentes básicos não podem ser usados diretamente (instanciados)[3] por uma aplicação de gerência, pois possuem características que dependem de um protocolo de gerência. Por exemplo, o componente

[3] Os componentes básicos são implementados como interfaces ou classes abstratas na linguagem Java.

`ElementoGerenciado` não pode ser instanciado por uma aplicação para representar uma entidade gerenciada, mas é permitido instanciar um componente que o estenda fornecendo as características específicas do protocolo desejado.

Portanto, a partir destes componentes básicos, pode-se também criar novos outros, como por exemplo, aqueles que implementam as características particulares de um protocolo de gerência específico. É o caso justamente dos componentes referentes ao protocolo de gerência SNMP (*Simple Network Management Protocol*) que são providos pelo *framework* (`ConfiguradorDeRequisicaoSnmp`, `ElementoGerenciadoSnmp`, etc.). Estes herdam as características dos componentes básicos e implementam as características peculiares ao SNMP. Dessa forma, pode-se ampliar ainda mais o conjunto de componentes que constitui o *framework*.

A **Tabela 2.1** apresenta um resumo dos componentes que são fornecidos pelo fwGF.

2.6. A Comunicação entre os Componentes

Conforme já citado anteriormente, a comunicação entre componentes no fwGF é feita de acordo com o descrito no padrão *Observer* [Gamma *et al.*, 1995]. Segundo este padrão, um componente B (consumidor ou *listener*), interessado em receber eventos gerados por um outro A (fonte), cadastra-se neste. Então, toda vez que A gerar um evento, este é enviado para B (**Figura 2.4**).

Dessa forma, por exemplo, um gerador de eventos do tipo `GeradorDeEventoFalha` pode então se cadastrar junto a um `Monitor` para receber eventos do tipo `MonitorEvent`. A partir deste evento, o gerador pode analisar os valores coletados das variáveis de gerência e verificar se houve ou não a ocorrência de uma falta (**Figura 2.5**). De maneira análoga, um componente do tipo `CorrelatorDeEventoFalha` pode-se cadastrar em um `GeradorDeEventoFalha` para receber eventos do tipo `EventoFalhaEvent` (**Figura 2.5**).

Retomando o exemplo mostrado no início deste capítulo, onde fora construída uma aplicação para se verificar o estado de funcionamento de um roteador, convém, agora, apresentar a construção desta mesma aplicação utilizando-se dos componentes do fwGF descritos acima. Vale salientar que, como é necessário se utilizar um protocolo de gerência para se poder gerenciar uma rede, então, os componentes utilizados aqui serão os referentes ao protocolo SNMP.

Componente	Descrição
ConfiguradorDeRequisicao	Configura parâmetros de uma requisição.
ConfiguradorDeRequisicao-Snmp	Configura os parâmetros de uma requisição SNMP.
ElementoGerenciado	Representa uma entidade gerenciada.
ElementoGerenciadoSnmp	Representa uma entidade SNMP gerenciada.
GrupoInfoGerencia	Representa um conjunto de unidades (variáveis) de informação.
GrupoInfoGerenciaSnmp	Representa um grupo de variáveis MIB [Rose, 1996] (InfoGerenciaSnmp) utilizado pela aplicação.
InfoGerencia	Representa uma unidade (variável) de informação.
InfoGerenciaSnmp	Representa as variáveis MIB.
Monitor	Efetua monitoração síncrona, controlando a frequência com que um ElementoGerenciadoSnmp requisita informações e gerando eventos que contêm o resultado da monitoração realizada.
GeradorDeEventoFalha	Identifica faltas (gera eventos).
GeradorDeEventoFalha-ComHisterese	Gera eventos com base no mecanismo de histerese.
GeradorDeEventoFalhaLimiar	Gera eventos com base em limiares informados.
ReceptorDeTraps	Efetuar monitoração assíncrona.
ReceptorDeTrapsSnmp	Responsável por monitoração assíncrona em uma rede SNMP. Recebe notificações (*traps*) geradas pelos agentes SNMP da rede [Rose, 1996].
GeradorDeEventoFalhaTrap	Gera eventos com base nos traps recebidos da rede.
CorrelatorDeEventoFalha	Correlaciona eventos.
CorrelatorDeEventoFalha-Compressao	Faz correlação de eventos com base no algoritmo de compressão.
CorrelatorDeEventoFalha-Supressao	Faz correlação de eventos com base no algoritmo de supressão.
CorrelatorDeEventoFalhaCMP	Utilizado para correlacionar eventos quando se deseja efetuar um Controle de Manutenção Programada.

Tabela 2.1. Resumo dos Componentes fornecidos pelo fwGF.

Para se descrever a rede gerenciada, deve-se utilizar o componente ElementoGerenciadoSnmp para representar o equipamento a ser gerenciado. Este

componente é equivalente ao *Roteador* utilizado anteriormente. Com relação à *Informação Requisitada*, deve-se utilizar os componentes `InfoGerenciaSnmp`, que representam a informação de gerência e GrupoInfoGerenciaSnmp, ao qual o anterior é adicionado. O *Monitor* utilizado no exemplo inicial é aqui substituído pelo também denominado `Monitor`, cuja função é requisitar ao `ElementoGerenciadoSnmp`, a intervalos de tempo regulares, o `GrupoInfoGerencia` e, consequentemente, obter o valor de todas as unidades de informação de gerência contidas nele. Além disso ele gera um evento, representado pelo componente `MonitorEvent`, que é repassado aos componentes nele cadastrados.

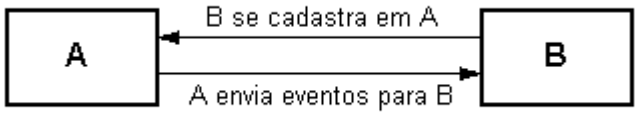


Figura 2.4. Representação do padrão *Observer*

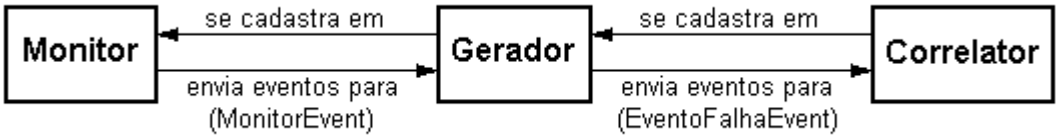


Figura 2.5. Padrão *Observer* no *framework*.

Para se detectar falhas, deve-se criar um novo componente, que pode ser estendido do `GeradorDeEventoFalhaComHisterese`, denominado `GeradorDeEventoFalhaStatusEquipamento`, o qual é associado ao `Monitor` para receber o evento gerado, ou seja, o valor da informação requisitada e analisar a ocorrência de falha no roteador. Em caso afirmativo, este componente gera um evento do tipo `EventoFalhaEvent` que é repassado para todos os interessados em recebê-lo.

Com relação ao tratamento de falhas, pode-se criar um novo componente a partir do `GeradorDeAlarmes` que implementa as características necessárias para se enviar uma mensagem de correio eletrônico para que o responsável pela administração do roteador possa ser notificado. Este componente pode ser chamado de `NotificaçãoDeAlarmesPorEmail` e deve ser associado ao `GeradorDeEventoFalhaStatusEquipamento` para poder receber os eventos por ele gerados.

Efetuada esta nova montagem da aplicação de exemplo, o novo "pipeline" referente à comunicação entre os componentes que a constituem apresenta-se como mostrado na **Figura 2.6**.

Figura 2.6. Aplicação de monitoração de um roteador construída com o fwGF: Comunicação entre os componentes

Portanto, como pode ser observado acima, para se construir uma aplicação com o fwGF, o programador deve utilizar os componentes necessários a ela e, se for preciso, ele também pode estender a funcionalidade dos componentes já existentes no *framework*, construindo os seus próprios.

Apesar de alguns componentes do fwGF já estarem definidos, o trabalho desenvolvido em [Freire, 2000] é ainda apenas uma especificação. No capítulo a seguir são mostrados os principais detalhes referentes à etapa de implementação deste *framework*.

3. Implementação do Framework

O objetivo deste capítulo é apresentar ao leitor detalhes relativos à etapa de implementação do *framework* especificado em [Freire, 2000], aqui referenciado como fwGF.

Ao longo desta etapa, foram implementados os componentes que compõem a especificação original do *framework*. Além disso, alguns novos componentes, bem como atributos e métodos, tiveram que ser adicionados ao *framework* com o intuito de torná-lo mais completo e/ou facilitar o seu desenvolvimento.

Com relação à linguagem de programação utilizada para se implementar o *framework*, Java foi a escolhida. Tal escolha se deve, principalmente, ao fato dela atender ao requisito de portabilidade estabelecido para o *framework* em [Freire, 2000], além de permitir a implementação de componentes com maior facilidade devido à funcionalidade de reflexão[4] (*Reflection*) [Eckel, 2000] oferecida pela linguagem.

É importante frisar que o leitor não encontrará neste capítulo uma descrição sobre a implementação de cada componente, o que tornaria a leitura exaustiva e de pouco valor agregado. Na realidade, apenas serão apresentados os aspectos da implementação considerados relevantes.

Portanto, segundo estes aspectos considerados de destaque, a estrutura deste capítulo encontra-se dividida nas seguintes seções: na **seção 3.1**, falar-se-á da criação de novas interfaces no *framework*, destacando-se a sua importância; na **seção 3.2**, serão tratados os aspectos que envolvem a requisição de informações dos dispositivos gerenciados de uma rede através do protocolo SNMP; a **seção 3.3** destaca a adaptação efetuada ao componente `InfoGerenciaSnmp` para o cálculo de taxas; na **seção 3.4** será feita uma descrição sobre o componente `Monitor`, bem como será mostrada uma análise de desempenho sobre aplicações construídas com o *framework*; a **seção 3.5** apresenta os aspectos relacionados com a implementação dos componentes para a recepção de *traps*; a **seção 3.6** trata dos testes de unidade aplicados ao fwGF para se verificar a corretude de seu código; e para finalizar este capítulo, a **seção 3.7** apresenta a linguagem de *script* BML (*Bean Markup Language*), a qual é utilizada para se construir aplicações com os componentes do fwGF e executá-las.

[4] Mecanismo que permite a descoberta de informações (métodos, propriedades, etc.) sobre classes em tempo de execução. Esta característica é bastante utilizada por ferramentas gráficas de desenvolvimento (*RAD – Rapid Application Development*) e para se poder criar e executar objetos distribuídos em máquinas remotas em uma rede (*RMI – Remote Method Invocation*).

3.1. Programando para Interfaces

Na especificação original do *framework*, alguns dos seus componentes herdam de classes abstratas. Entretanto, estas classes não são "puramente abstratas", isto é, elas já provêem implementação para alguns de seus métodos. Este tipo de herança é chamado de herança de classe ou de implementação.

A herança de implementação em um projeto de software é bastante interessante na construção de uma hierarquia de classes pois permite o reuso de código já escrito, reaproveitando a funcionalidade nele presente, o que caracteriza o *polimorfismo*. Mas, este tipo de herança também apresenta algumas desvantagens referentes à extensibilidade de um software. A maior delas é o maior acoplamento entre classes. Por exemplo, se uma classe A referencia um objeto O através de uma classe abstrata B, então A só pode ser usada com instâncias ou suhclasses de B.

Considere como exemplo a classe abstrata `GeradorDeEventoFalha` contida no fwGF. Para que uma classe qualquer possa usar um objeto desse gerador, ela tem que referenciar subclasses deste. Por outro lado, se `GeradorDeEventoFalha` fosse uma interface, qualquer classe que implemente a interface poderá ser usada o que contribui para um desacoplamento entre as classes.

A herança baseada em interface, portanto, agrega o conceito de tipo ao projeto de software. Dessa forma, tanto uma classe pode ter vários tipos como classes diferentes (pertencentes a hierarquias distintas) podem ter o mesmo tipo.

Sendo assim, considerando-se o exemplo citado acima, se `GeradorDeEventoFalha` for transformado em uma interface, então qualquer classe pode exercer o papel de um gerador de eventos, desde que forneça a implementação para os métodos daquela interface (**Figura 3.1**).

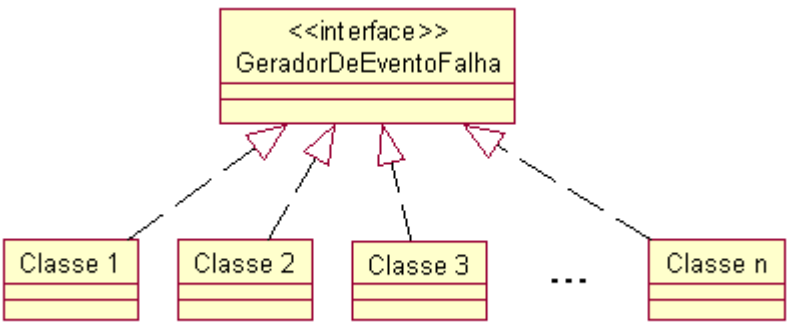


Figura 3.1. Herança de interface com o GeradorDeEventoFalha

De maneira análoga, qualquer componente, independente de estrutura hierárquica, pode receber eventos do tipo `MonitorEvent` (gerado por monitores) ou `EventoFalhaEvent` (gerado por geradores de eventos) bastando para isso implementar a interface `MonitorListerner` ou `EventoFalhaListener`, respectivamente. Um exemplo disso são os correlatores de evento do fwGF que, além de implementarem a interface `CorrelatorDeEventoFalha`, ainda implementam os métodos de `GeradorDeEventoFalha` e de `EventoFalhaListener`. Dessa forma, um correlator desempenha o papel tanto de gerador como de consumidor de eventos (inclusive ambos ao mesmo tempo).

Portanto, como o leitor pode observar, a herança de interface produz um projeto de software com um maior poder de extensibilidade. Além disso, ganha-se muito em flexibilidade. Por exemplo, quando um `Monitor` envia um evento para os *listeners* nele cadastrados, é necessário fazer uma chamada ao método `processaDadosMonitorados(MonitorEvent me)` de cada *listener* para que o evento enviado seja devidamente processado. Então, como o `Monitor` trata todos os *listeners* como sendo do tipo `MonitorListener`, a chamada do método é associada dinamicamente à implementação correspondente.

Com base então nas vantagens de se construir um projeto de software baseado em interfaces, foram adicionadas interfaces para os componentes do fwGF e classes (abstratas) adaptadoras para prover uma implementação *default* para os métodos destas interfaces. Uma classe adaptadora serve para facilitar a implementação de classes que querem implementar uma certa interface mas que querem um comportamento *default* para a maioria dos métodos da interface (**Figura 3.2**). Dessa forma, faz-se com que a classe de implementação sendo escrita herde da classe adaptadora e faça o *override* somente dos métodos onde haja um comportamento diferente do *default*. Portanto, pode-se criar novos componentes tanto através da extensão dessas classes como pela implementação direta da interface definida.

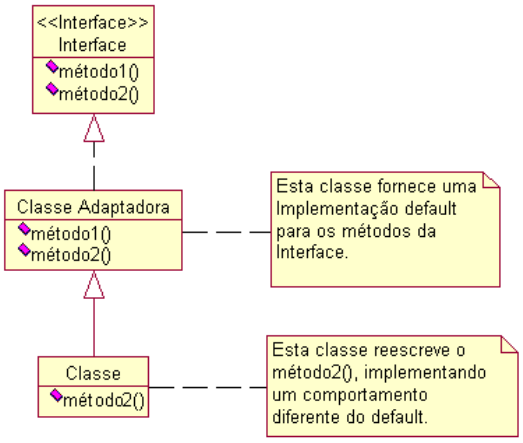


Figura 3.2. Representação de uma Requisição SNMP no fwGF

A **Tabela 3.1** apresenta as interfaces e classes adaptadoras adicionadas ao fwGF durante a fase de implementação.

Classe original	Interface	Classe Adaptadora
ConfiguradorDeRequisicao	ConfiguradorDeRequisicao	ConfiguradorDeRequisicaoAdapter
GrupoInfoGerencia	GrupoInfoGerencia	GrupoInfoGerenciaAdapter
GeradorDeEventoFalha	GeradorDeEventoFalha	GeradorDeEventoFalhaAdapter
Monitor	MonitorInterface	MonitorAdapter
ReceptorDeTraps	ReceptorDeTraps	ReceptorDeTrapsAdapter

Tabela 3.1. Interfaces e classes adicionadas ao *framework*.

3.2. Requisições SNMP

O fwGF, apesar de ser independente de protocolo de gerência, fornece suporte *default* ao SNMP (*Simple Network Management Protocol*). Portanto, ele possui em sua estrutura componentes que estendem os componentes básicos e que encapsulam as características referentes a este protocolo.

Para se representar as entidades gerenciadas da rede, o fwGF fornece o componente `ElementoGerenciado`. No caso específico do SNMP, o `ElementoGerenciadoSnmp` é quem fornece a implementação necessária para se representar os agentes SNMP. O método `getGrupoInfoGerencia(GrupoInfoGerencia gig)` deste componente contém o código responsável para se efetuar uma requisição ao agente SNMP para a coleta dos valores das variáveis contidas no `GrupoInfoGerencia`. Este é justamente o método chamado

pelo `Monitor` a intervalos de tempo regulares para se efetuar uma monitoração síncrona sobre a entidade, para o qual ele passa o grupo de informações de gerência desejado e, após finalizada a requisição, este mesmo grupo é retornado ao `Monitor` contendo os valores obtidos para as variáveis nele contidas.

Para a implementação deste método, optou-se por não implementar toda a sua funcionalidade a partir do "zero", mas sim reutilizar algo que já tivesse sido desenvolvido. Para isso, se está fazendo uso de um pacote denominado *AdventNet* [AdventNet, 2001]. Este pacote é constituído por um conjunto de componentes para a criação de aplicações de gerência de redes através do protocolo SNMP. No `ElementoGerenciadoSnmp`, se está utilizando o componente `SnmpTarget` deste pacote.

O `SnmpTarget` pode ser usado em aplicações, *applets* ou por outros componentes. Ele é utilizado quando se precisa efetuar requisições SNMP e esperar por suas respectivas respostas. A utilização deste oferece algumas vantagens, como por exemplo, a não necessidade de se preocupar com a abertura e o fechamento de sessões SNMP durante uma requisição. Sendo assim, o `ElementoGerenciadoSnmp` delega todo o processo relativo ao gerenciamento de requisições SNMP para o `SnmpTarget` (**Figura 3.3**).

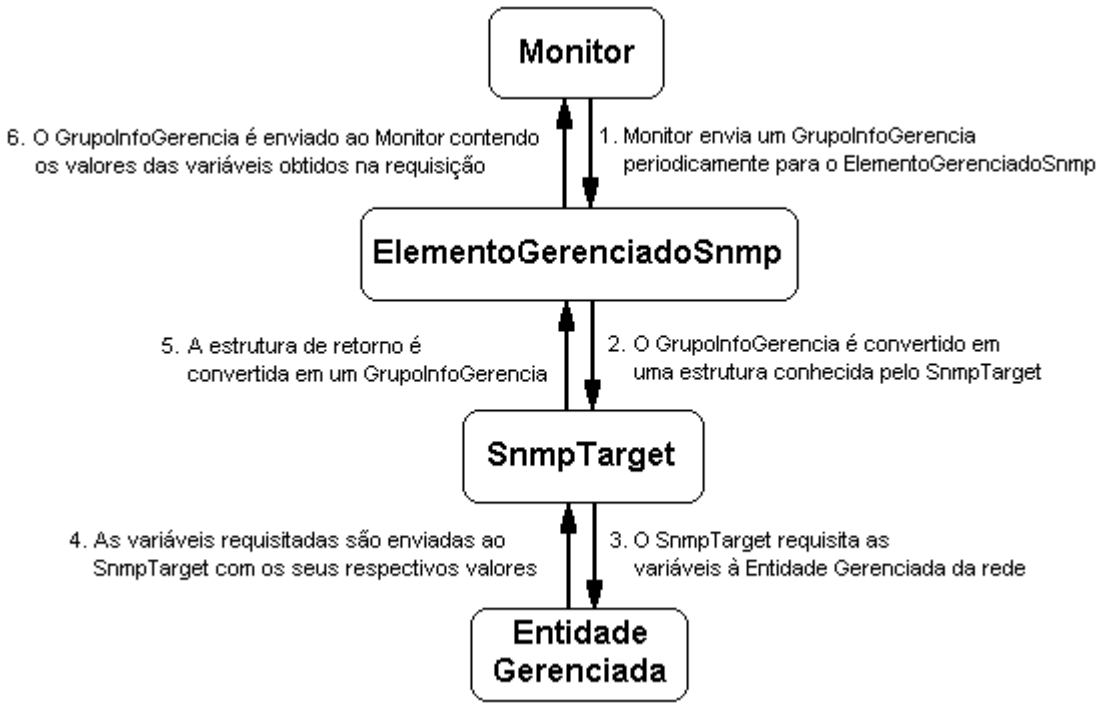


Figura 3.3. Representação de uma Requisição SNMP no fwGF

É importante salientar, que o fwGF é independente do protocolo de gerência. Ele oferece suporte *default* ao protocolo SNMP. Porém, para que o mesmo possa construir aplicações utilizando outro protocolo, basta apenas prover os componentes necessários (com as características peculiares ao protocolo) através da extensão das interfaces existentes no *framework.*

A **Figura 3.4** mostra o código-fonte do método `getGrupoInfoGerencia(GrupoInfoGerencia gig)` do `ElementoGerenciaSnmp`. Como pode ser visto nesta figura, inicialmente uma instância do SnmpTarget é criada. Em seguida, são configurados os parâmetros necessários para que este possa efetuar a requisição SNMP (endereço IP da entidade gerenciada, o nome da comunidade, o número de retransmissões, o *timeout* e a versão do protocolo SNMP utilizada na rede). Por conseguinte, é informado o conjunto de variáveis a ser coletado da entidade. Este conjunto é obtido através da chamada ao método `getSnmpOIDList()` do `GrupoInfoGerenciaSnmp` (que implementa a interface `GrupoInfoGerencia`, provendo algumas características peculiares ao SNMP). Este método retorna um *array* de `SnmpOID` (um outro componente do pacote *AdventNet*) o qual é passado como parâmetro para o `SnmpTarget`. A **Figura 3.4** mostra os componentes envolvidos na requisição.

O método do `SnmpTarget` aqui utilizado para fazer a requisição à entidade gerenciada é o `snmpGetVariables()`, o qual retorna um *array* de `SnmpVar` (este componente do *AdventNet* representa uma variável MIB). Se o resultado da requisição é diferente de nulo, significa que os valores das variáveis foram obtidas com sucesso. Por outro lado, se o valor de retorno da requisição for nulo, isso significa que algum erro aconteceu durante a requisição, como por exemplo, uma falha na comunicação com o agente SNMP da entidade gerenciada, uma variável informada e que não encontra-se na MIB (*Management Information Base*) desta entidade, entre outros.

3.3. Cálculo de Taxas

No fwGF, o componente `InfoGerencia` representa uma unidade de informação de gerência presente em um `GrupoInfoGerencia`. No caso específico do SNMP, `InfoGerenciaSnmp` estende as características básicas de `InfoGerencia` e provê as peculiaridades deste protocolo, representando cada variável MIB presente em um `GrupoInfoGerenciaSnmp`.

Na especificação original do *framework*, o `InfoGerenciaSnmp` armazena o valor de uma variável MIB obtido durante uma requisição. Entretanto, para algumas variáveis, seus valores pontuais em um dado instante de tempo não têm nenhum significado semântico. Estas variáveis são as do tipo *contador* (tipos `Counter` e `Counter64` no SNMP). Para que variáveis deste tipo tenham significado relevante, é preciso que seja calculada a relação entre dois de seus valores obtidos e o intervalo de tempo transcorrido entre as duas requisições.

Considere como exemplo a variável `ifInErrors` do grupo `interfaces` da `MIB-2` (um conjunto de variáveis de gerência padrão de todo agente SNMP), cujo valor indica a quantidade de erros de entrada ocorridos em uma interface desde que o agente SNMP entrou no ar. Um valor individual desta variável não possui nenhum significado útil. Entretanto, quando se usa essa variável para se calcular a taxa de erros em uma interface em um dado intervalo de tempo, então, tem-se um resultado semanticamente importante, pois pode-se deduzir inúmeras conclusões a partir dele, como por exemplo, a ocorrência de falhas na respectiva interface.

Entretanto, na especificação original do fwGF, `InfoGerenciaSnmp` não possui uma forma de se armazenar os valores obtidos em duas requisições consecutivas à variável. Portanto, para que seja possível se calcular taxas, `InfoGerenciaSnmp` sofreu uma adaptação para que pudessem ser mantidos os valores obtidos nas duas últimas requisições à variável MIB, bem como os respectivos valores da variável `sysUpTime` do grupo `system`, também pertencente à `MIB-2`, a qual armazena a quantidade de tempo transcorrida desde o momento em que o agente SNMP entrou no ar. Dessa forma, quando se faz uma chamada ao método `getString()` de uma instância de `InfoGerenciaSnmp`, por exemplo, é verificado se a variável é do tipo `Counter` ou `Counter64`. Em caso afirmativo, é retornado o valor da taxa, se não, é retornado o valor obtido na requisição mais recente.

```
public GrupoInfoGerencia getGrupoInfoGerencia(GrupoInfoGerencia gig) {
  if (gig != null){
    // Instancia um SnmpTarget para efetuar busca no elemento gerenciado
    SnmpTarget target = new SnmpTarget();

    // Informar o endereço IP do elemento gerenciado
    target.setTargetHost(endIp);

    // Setar parâmetros de configuração da requisição SNMP.
    target.setCommunity(
      ((ConfiguradorDeRequisicaoSnmp)getConfiguradorDeRequisicao()).getComunidade()
    );
    target.setRetries(
      ((ConfiguradorDeRequisicaoSnmp)getConfiguradorDeRequisicao()).getNumeroRetransmissoes()
    );
    target.setTimeout(
      ((ConfiguradorDeRequisicaoSnmp)getConfiguradorDeRequisicao()).getTimeout()
    );
    target.setSnmpVersion(
      ((ConfiguradorDeRequisicaoSnmp)getConfiguradorDeRequisicao()).getVersao()
    );

    // Faz a requisição SNMP de todas as variáveis contidas
    // em GrupoInfoGerencia. Aqui utilizamos o método snmpGetList (SnmpOID[])
    // de SnmpTarget. Este método solicita ao agente as informações contidas
    // no array de SnmpOID e retorna um array de SnmpVar, o qual contém os
    // respectivos valores das variáveis.
    target.setSnmpOIDList(((GrupoInfoGerenciaSnmp)gig).getSnmpOIDList());
    SnmpVar result[] = target.snmpGetVariables();

    // Atualizar o GrupoInfoGerencia com os resultados obtidos na
    // requisição SNMP
    ((GrupoInfoGerenciaSnmp)gig).setOidListResult(result);
    // Verifica se a requisição foi feita com sucesso
    if (result != null){
      // Setar o status deste ElementoGerenciado para OK
      this.status = this.OK;
    } else {
      // Setar o status deste ElementoGerenciado para NOK
      this.status = this.NOK;
    }
  }
  return gig;
};
```

Figura 3.4. Método getGrupoInfoGerencia(GrupoInfoGerencia gig) de ElementoGerenciadoSnmp

3.4. Uso do Padrão *Adapter* na Recepção de *Traps*

Para a monitoração assíncrona da rede, isto é, para a recepção das notificações (*traps*) recebidas dos elementos gerenciados, o fwGF provê um componente chamado `ReceptorDeTraps`. Para prover as características específicas do protocolo de gerência SNMP, o *framework* conta com o componente `ReceptorDeTrapsSnmp`.

Da mesma forma que para se fazer requisições a uma entidade gerenciada, para a recepção dos *traps* da rede se está fazendo uso do pacote *AdventNet*. O componente deste pacote utilizado para a recepção de *traps* é o `SnmpTrapReceiver`, o qual fica "ouvindo" na porta configurada para a recepção de *traps* da rede (a porta UDP 162 é a *default*) e, toda vez que um *trap* é recebido, ele repassa as informações contidas neste *trap* (gera um evento `com.adventnet.snmp.beans.TrapEvent`)[5] para os consumidores cadastrados (`com.adventnet.snmp.beans.TrapListener`).

Entretanto, para se poder utilizar o `SnmpTrapReceiver`, é necessário então que o `ReceptorDeTrapsSnmp` seja transformado em um consumidor (ou seja, implemente `com.adventnet.snmp.beans.TrapListener`), para que assim ele possa receber os eventos gerados por aquele e possa convertê-los em eventos do tipo `TrapEvent` do *framework*.

Sendo assim, foi aplicado o padrão de projeto (*design pattern*) *Adapter* [Gamma *et al*, 1995] ao `ReceptorDeTrapsSnmp` (**Figura 3.7**).

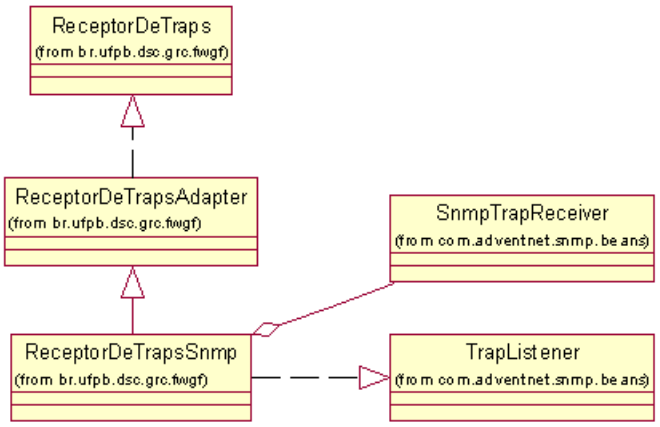


Figura 3.7. Uso do padrão de projeto *Adapter* no ReceptorDeTrapsSnmp

[5] Devido ao fato do pacote AdventNet e do fwGF possuírem componentes chamados TrapEvent e TrapListener, para se evitar equívocos, quando o texto fizer referência aos componentes do AdventNet, eles virão acompanhados da designação deste pacote (com.adventnet.snmp.beans).

Portanto, este componente passa agora a implementar a interface `com.adventnet.snmp.beans.TrapListener`. Além disso, o `ReceptorDeTrapsSnmp` passou a ser composto por um `SnmpTrapReceiver`. Cada vez que uma instância do primeiro é criada, automaticamente ela é cadastrada no `SnmpTrapReceiver`. Cada vez que este último recebe um *trap* vindo da rede, ele gera um evento para os `ReceptorDeTrapsSnmp` nele cadastrados. Estes, então, enviam eventos para os seus próprios consumidores, como por exemplo, o `GeradorDeEventoFalhaTrap`, que verifica se os *traps* recebidos caracterizam uma situação de falha na rede.

A **Tabela 3.2** apresenta um resumo da etapa de implementação do fwGF.

Descrição	Quantidade
Componentes implementados	19
Linhas de código	5080
Interfaces	12
Classes abstratas	11
Classes concretas	27

Tabela 3.2. Resumo de Implementação do fwGF.

3.5. Testes de Unidade do *Framework*

Como forma de se verificar a corretude de um software, é imprescindível que se estabeleça uma política de testes. Testar um software tem por objetivo assegurar a qualidade final do produto reduzindo defeitos que possivelmente o mesmo possa conter [Beck & Gamma, 1998].

Testes devem ser feitos em todas as fases do desenvolvimento de um produto. Além disso devem ser testadas pequenas partes de cada vez, ou seja, à medida em que se está desenvolvendo o produto, deve-se construir e executar os testes paralelamente, corrigindo os problemas encontrados e evitando que os mesmos se propaguem para as fases subseqüentes do desenvolvimento [Beck & Gamma, 1998]. Efetuar testes apenas quando o produto está finalizado não é uma boa prática, tendo em vista que isso aumenta consideravelmente o tempo gasto para depurá-lo.

Existem vários tipos de testes que podem ser utilizados para a verificação de um software como por exemplo testes de unidade, de integração, de sistema, funcionais, de regressão, entre outros [Kaner, 1999] [Dustin *et al.*, 1999].

Paralelamente à implementação do fwGF, foram construídos testes de unidade. Estes têm o objetivo de exercitar detalhadamente os possíveis caminhos de execução de uma unidade de código, nesse caso, uma classe (ou um grupo delas, quando fortemente acopladas), testando o comportamento de suas interfaces, estruturas de dados e condições limites, sempre com o intuito de detectar problemas e corrigi-los para, por fim, assegurar a integridade da unidade em teste.

Para automatizar a execução dos testes de unidade, foi utilizado um *framework* de testes chamado *JUnit* [Beck & Gamma, 1998]. Para se utilizar este *framework*, deve-se informar o nome da classe de teste a ser executada pelo mesmo. Após a execução, a relação dos métodos onde ocorreram erros (caso haja algum) é apresentada ao usuário. A partir daí o usuário pode fazer as devidas alterações nas classes originais como forma de corrigir os erros detectados.

A principal vantagem de se construir testes de unidades e de se automatizar o processo de execução dos mesmos através do JUnit é que as classes de teste construídas podem ser executadas sempre que for necessário. Por exemplo, mudanças no código de um software devido a manutenções periódicas no mesmo podem afetar outras partes do software provocando erros que antes não ocorriam. Portanto, é altamente recomendado que a cada mudança se executem todos os testes de unidade elaborados para se verificar se tais alterações não acarretam em problemas ao produto final. Além disso, testes de unidade fazem parte do software e se tornam, portanto, patrimônio da empresa que desenvolveu o software.

Uma outra vantagem bastante importante dos teste de unidade é que os mesmo podem ser utilizados também como forma de documentação do software. Quando não se tem certeza como determinada parte de um software funciona ou deve ser utilizado, pode-se consultar os testes de unidade. O código dos próprios testes deve ser suficiente para tirar as dúvidas.

Com relação aos testes construídos para o fwGF, um problema detectado em algumas de suas classes foi o acesso a objetos não inicializados (objetos nulos). Dessa forma, com a execução dos testes foi possível descobrir e solucionar tal problema ainda durante a fase de implementação do fwGF, evitando assim que tais erros ocorressem durante a execução

das aplicações construídas com o *framework*, provocando exceções do tipo `java.lang.NullPointerException` [Sun, 1999b].

É importante frisar aqui, que para se executar os testes de unidade construídos para o fwGF não é necessário a existência de uma rede de computadores. Todos os valores de limiares são previamente estabelecidos e a geração de eventos e de valores das variáveis de gerência são gerados randomicamente.

A **Tabela 3.3** apresenta um resumo dos testes efetuados para o fwGF.

Descrição	Quantidade
Classes de Teste	14
Métodos de Teste	120
Asserts (métodos que testam uma condição em um método de teste)	267

Tabela 3.3. Estatísticas de Testes de Unidade do fwGF.

3.6. Configurando e Instanciando Componentes com BML

Para se construir uma aplicação com o fwGF, como o mesmo depende da composição de componentes, é preciso especificar que composições devem ser efetuadas além de se configurar as propriedades parametrizadas de cada componente. É importante destacar que uma propriedade é um atributo que pode ser descoberto por uma ferramenta externa de composição e seu valor obtido e/ou alterado por ela. Além disso, um componente pode ter outros atributos que não foram "expostos" e, portanto, não são propriedades.

Devido ao fato do fwGF ainda não possuir uma ferramenta visual através da qual se possa efetuar a composição e a configuração das propriedades dos componentes de uma aplicação, uma ferramenta externa de composição foi então pesquisada como meio para suprir esta necessidade.

Portanto, optou-se por utilizar BML (*Bean Markup Language*) [Kesselman & Duftler, 1999], uma linguagem baseada em XML (*Extended Markup Language*) utilizada para a configuração e instanciação de componentes Java.

A linguagem BML possui elementos que podem ser utilizados para a instanciação de componentes, para se ter acesso a componentes já instanciados, para configuração de componentes através de suas propriedades, para se efetuar a comunicação entre *beans* através de eventos, assim como para se fazer chamada de métodos.

Para se configurar os componentes, deve-se escrever um *script BML*, que após ser processado, resultará na aplicação configurada. Este *script BML* pode conter apenas um *bean* ou então um conjunto de componentes conectados entre si formando uma aplicação completa [Johnson, 1999].

Nas subseções a seguir, serão mostrados detalhes referentes à linguagem BML. Inicialmente, serão descritas as *tags* que constituem a linguagem e serão mostrados exemplos sobre como se instanciar componentes, acessar componentes previamente instanciados, setar propriedades, fazer chamada a métodos, entre outras operações. Em seguida, serão descritos os utilitários usados para se executar scripts BML, *BML player* e *BML compiler*, fornecidos com a linguagem. Estas subseções não são de leitura obrigatória para aqueles que já conhecem a linguagem BML; o leitor pode sentir-se à vontade para passar à leitura dos demais capítulos deste documento.

3.6.1. A linguagem BML

BML possui um conjunto de elementos (*tags*) que suporta a configuração de componentes e que expressa as operações básicas sobre estas, tais como instanciação, inter-conexão (*event-binding*) e chamada de métodos. A **Tabela 3.4** apresenta uma breve descrição dos elementos utilizados para se construir os scripts BML.

Elemento	Descrição
<bean>	Instanciar um novo componente ou ter acesso a um já instanciado
<args>	Especificar os argumentos de um construtor
<string>	Criar uma string
<property>	Alterar ou obter o valor de uma propriedade de um componente
<field>	Alterar ou obter o valor de um campo de um componente
<event-binding>	Associar um evento de um componente para outro
<call-method>	Efetuar uma chamada a um método de um componente
<cast>	Conversão de tipos
<add>	Criar uma hierarquia de componentes, adicionando uns aos outros
<script>	Definir uma sequencia de declarações em BML (ou em uma outra linguagem suportada, como por exemplo Javascript)

Tabela 3.4. Elementos da linguagem BML.

Apesar de não ser necessário para alguns *parsers XML*, um arquivo de script BML deve conter como primeira linha o seguinte elemento `<?xml version="1.0"?>`. Ele indica ao *parser* qual a versão do XML na qual o script é baseado. Após esse elemento vem o código BML responsável pela configuração dos componentes.

Nas subseções a seguir, serão apresentados alguns dos elementos acima citados. Além disso serão mostrados trechos de scripts BML para exemplificar as operações mais comuns sobre componentes, como a instanciação, a chamada de métodos, a mudança de propriedades, a associação de eventos e a conversão de tipos.

3.6.1.1. Instanciando os Componentes

Conforme já citado anteriormente, um script BML é composto por um componente ou um conjunto deles conectados entre si. Portanto, para se representar estes componentes dentro do script, deve-se utilizar o elemento BML `<bean>`. Este elemento é utilizado tanto na instanciação de componentes quanto no acesso àqueles previamente instanciados (mostrado na próxima seção). A **Figura 3.8** mostra a sintaxe desse elemento[6].

```
<bean class="nome da classe" id="nome de registro">
    ... elementos de configuração do bean ...
</bean>
```

Figura 3.8. Sintaxe do elemento <bean>: instanciação

Para se instanciar o componente deve-se informar no campo class o nome da classe a ser instanciada. Se o elemento `<args>` não for informado, o construtor utilizado na instanciação será o construtor sem argumentos. Se <args> estiver presente, o construtor chamado será aquele cuja assinatura contiver uma lista de parâmetros cujos tipos coincidam com os da lista informada.

Um identificador (*id*) pode ser, opcionalmente, associado à instância para que a mesma seja registrada no espaço de nomes do BML (*object registry*), com esse *id* servindo como chave para que se possa ter acesso a essa instância em outros locais do script.

A **Figura 3.9** mostra um trecho de um script utilizado para se instanciar um componente do tipo ElementoGerenciadoSnmp no fwGF. Neste caso, estar-se-á configurando um `ElementoGerenciadoSnmp` cujo identificador é *leprecom*.

[6] Alguns dos elementos aqui mostrados apresentam mais de uma sintaxe, embora com pequenas distinções entre si. Entretanto, nem todas elas são destacadas neste documento. Para se conhecer todas elas deve-se consultar [Weerawarana e Duftler, 1999].

3.6.1.2. Tendo Acesso a um Componente Instanciado

Após ter sido instanciado, um componente pode ser acessado em outra parte do script BML também através do uso do elemento <bean>. A sintaxe utilizada nesse caso é a seguinte:

```
<bean class="br.ufpb.dsc.grc.fwGF.ElementoGerenciadoSnmp" id="leprecom">
      <property name="nome" value="150.165.75.152"/>
      <property name="endIp" value="150.165.75.152"/>
      <property name="nomeResponsavel" value="Pedro Sérgio Nicolletti"/>
      <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
      <property name="configuradorDeRequisicao">
            <bean source="configurador1"/>
      </property>
</bean>
```

Figura 3.9. Instanciação de um componente do tipo ElementoGerenciadoSnmp

```
<bean source="nome do registro">
    ... elementos de configuração do bean ...
</bean>
```

Figura 3.10. Sintaxe do elemento <bean>: acesso

O *nome de registro* é o identificador que foi utilizado para se registrar o componente no *object registry*. Na **Figura 3.11** é apresentado um trecho de um script BML que mostra o acesso à instância do componente ElementoGerenciadoSnmp mostrada na **Figura 3.8**. Neste trecho, o ElementoGerenciadoSnmp registrado como *leprecom* está sendo atribuído à propriedade chamada *eg* do componente Monitor (propriedades são o tema da seção a seguir).

```
<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_leprecom">
      ...
      <property name="eg">
            <bean source="leprecom"/>
      </property>
      ...
</bean>
```

Figura 3.11. Acesso a um componente previamente instanciado

3.6.1.3. Alterando Propriedades dos Componentes

O elemento `<property>` é utilizado quando se deseja obter ou alterar o valor de uma propriedade de um componente. Para se alterar o valor de uma propriedade, pode-se aplicar uma das duas sintaxes mostradas abaixo. A **Figura 3.12** mostra a sintaxe utilizada quando se deseja atribuir um valor imediato a uma propriedade. Tal sintaxe apresenta os seguintes campos:

- *target*: deve-se utilizar este campo quando se deseja atribuir o valor a uma propriedade pertencente a um outro componente que não o *default* (contém o elemento `<property>`);
- *index*: quando presente, indica que a propriedade é indexada (um *array*, por exemplo) e o n-ésimo elemento da propriedade será então alterado;
- *id*: quando está presente, indica o nome utilizado para se registrar a propriedade no *object registry*;
- *name*: nome da propriedade;
- *value*: valor a ser atribuído à propriedade.

```
<property [target="bean alternativo"] name="nome da propriedade"
          [index="índice"] [id="nome de registro"] value="valor"/>
```

Figura 3.12. Sintaxe do elemento <property> para atribuição de um valor imediato

A **Figura 3.13** apresenta a sintaxe utilizada para se atribuir um valor indireto à propriedade. O termo indireto significa que o valor a ser atribuído à propriedade deve ser obtido de um outro local, como por exemplo, uma propriedade de um outro componente, o retorno de um método, etc.

```
<property [target="bean alternativo"] name="nome da propriedade"
          [index="índice"] [id="nome de registro"]>
    ...um bean, string, campo, propriedade, chamada de método ou script...
</property>
```

Figura 3.13. Sintaxe do elemento <property> para atribuição de um valor indireto

Na **Figura 3.9** pode-se observar exemplos de atribuição de valor imediato na instanciação do `ElementoGerenciadoSnmp` nas propriedades *nome*, *endIp*, *nomeResponsavel* e *contatoResponsavel*. Por outro lado, a propriedade

configuradorDeRequisicao presente no componente desta mesma figura é um exemplo de atribuição de um valor indireto, o qual é representado por um componente do tipo `ConfiguradorDeRequisicao`, previamente instanciado, com o *id configurador1*.

Com relação à obtenção do valor de uma propriedade, a **Figura 3.14** mostra a sintaxe utilizada para esta operação.

```
<property [target="bean alternativo"] name="nome da propriedade"
          [index="índice"] [id="nome de registro"]/>
```

Figura 3.14. Sintaxe do elemento <property> para se obter o valor da propriedade

3.6.1.4. Chamando um Método

Uma outra operação muito importante sobre componentes é a *chamada a métodos*. Isso é o que faz com que BML possa não somente especificar as composições de componentes para a aplicação mas executá-la também. Para se representar esta operação dentro de um script BML deve-se utilizar o elemento chamado `<call-method>`. Este apresenta a sintaxe mostrada na **Figura 3.15**.

```
<call-method [target="bean alternativo"] name="nome do método"
          [id="nome de registro"]>
    ...zero ou mais argumentos que podem ser beans, strings, campos,
    propriedades, chamada a métodos ou scripts...
</call-method>
```

Figura 3.15. Sintaxe do elemento <property> para se obter o valor da propriedade

Se o campo *target* estiver presente, a chamada ao método é efetuada sobre o componente indicado nessa propriedade. Este campo é bastante utilizado, por exemplo, nas chamadas a métodos estáticos (*static methods*), onde *target* representa o nome da classe desejada. O *id*, por sua vez, representa o nome utilizado para se registrar esta chamada no *object registry*.

Dentro de um `<call-method>` podem ser relacionados zero ou mais elementos, os quais são avaliados sequencialmente e resultarão na lista de argumentos passada ao método. O método chamado, então, será aquele cuja assinatura coincidir com os tipos dos argumentos passados, de acordo com a lógica de resolução de métodos utilizada em Java.

A **Figura 3.16** mostra um exemplo do uso do elemento `<call-method>`. Neste exemplo, o método `addInfoGerencia` do componente `GrupoInfoGerenciaSnmp` é

chamado passando-se como argumento um componente do tipo `InfoGerencia`, previamente instanciado, registrado no *object registry* com o nome *sysUpTime_leprecom*.

```
<bean class="br.ufpb.dsc.grc.fwGF.GrupoInfoGerenciaSnmp" id="gig_leprecom">
      ...
      <call-method name="addInfoGerencia">
            <bean source="sysUpTime_leprecom"/>
      </call-method>
      ...
</bean>
```

Figura 3.16. Exemplo de uma chamada a método usando o elemento `<call-method>`

3.6.1.5. Convertendo Tipos

O elemento `<cast>` é utilizado para se converter um tipo de dado em outro. A **Figura 3.17** apresenta a sintaxe desse elemento.

```
<cast class="classe destino" [value="string a ser convertida">
    ... bean, string, campo, propriedade, chamada a método ou script...
</cast>
```

Figura 3.17. Sintaxe do elemento <cast> para conversão de tipos

Se o campo *value* estiver presente, então a string será convertida para a classe destino. Por outro lado, se o campo *value* não for informado, então isso equivale à conversão (classe destino)null, ou seja, será feita a conversão do valor nulo (*null*) para o seu respectivo valor na classe destino.

A **Figura 3.18** mostra alguns exemplos do uso de `<cast>`.

```
<cast class="int" value="100"/>
<cast class="int"><string value="100"/></cast>
<cast class="int"><string>20</string></cast>
```

Figura 3.18. Exemplos de conversão de tipos usando o elemento `<cast>`

3.6.1.6. Trabalhando com Eventos

BML provê suporte ao modelo de eventos de componentes Java [Eckel, 2000] através do uso do elemento `<event-binding>`. De acordo com esse modelo, considerando que *XEvent* representa um tipo de evento, então, a todo componente (fonte) que gere eventos desse tipo, pode-se associar a ele outros componentes (consumidores ou *listeners*) que implementem a interface *XListener*, interessados em receber e tratar os eventos gerados. Para

que um componente possa gerar eventos para os seus *listeners* é necessário que o mesmo implemente os métodos *addXListener(XListener)* e *removeXListener(XListener)*, os quais são utilizados para cadastrar e remover os componentes interessados em receber eventos, respectivamente. A interface *XListener*, por sua vez, possui os métodos que devem ser chamados quando da geração dos eventos, para que estes possam então ser enviados aos *listeners* cadastrados.

A **Figura 3.19** mostra a sintaxe utilizada para se cadastrar um *listener* em um outro componente gerador de eventos. De acordo com essa sintaxe, um componente (*listener*) será cadastrado em um outro (fonte) dado pelo campo *target*. Todo evento gerado pelo componente fonte é então enviado ao *listener* cadastrado. Se o campo *target* não for informado, então o listener será cadastrado no componente fonte *default* (o que contém o `<event-binding>`).

```
<event-binding [target="bean alternativo" name="nome do evento">
    <bean .../>
</event-binding>
```

Figura 3.19. Sintaxe do elemento <event-binding>

A **Figura 3.20** mostra um exemplo do uso desse elemento no qual dois componentes do tipo GeradorDeEventoFalha, previamente instanciados (*status_leprecom* e *status27_leprecom*), são associados ao componente Monitor. Dessa forma, toda vez que o Monitor gerar um evento, este será então enviado aos dois *listeners* cadastrados.

Entretanto, além de se poder associar um componente como listener de outro através do elemento `<event-binding>`, pode-se também escrever um `<script>` que é então executado toda vez que um evento é gerado.

3.6.2. *BML player*

O *BML player* lê um arquivo contendo um *script BML*, instancia e conecta dinamicamente os componentes presentes neste script e executa a estrutura resultante. O *BML compiler*, por sua vez, lê o arquivo de *script BML* e o transforma em um código Java que pode ser compilado e executado ou como uma aplicação ou como um novo componente que pode ser embutido em uma outra aplicação.

```
<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_leprecom">
      ...
      <event-binding name="dispara">
            <bean source="status_leprecom"/>
      </event-binding >
      <event-binding name="dispara">
            <bean source="status27_leprecom"/>
      </event-binding >
      ...
</bean>
```

Figura 3.20. Exemplo do uso do elemento <event-binding>

Conforme mostrado na **Figura 3.21**, o *BML player* utiliza-se de um *parser XML* para transformar o script BML em uma estrutura de componentes chamada árvore DOM (*Document Object Model tree*). Com esta estrutura, o *BML player* pode instanciar e conectar os componentes nela contidos. Como resultado final tem-se uma aplicação em execução.

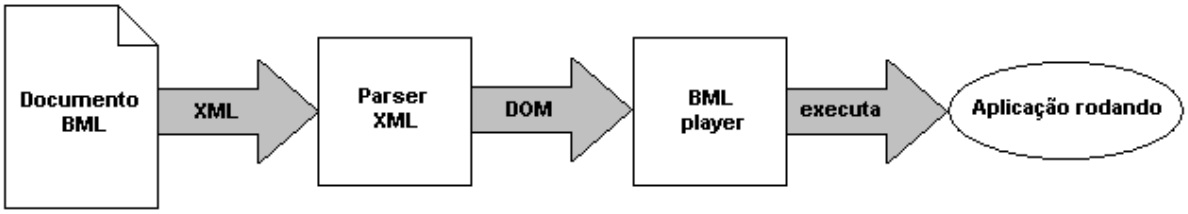


Figura 3.21. Executando uma aplicação com o BML player

3.6.3. *BML compiler*

O *BML compiler* também utiliza um *parser XML* para ler o arquivo de script BML e transformá-lo em uma árvore DOM. Entretanto, ao invés de instanciar os componentes como faz o *BML player*, ele gera um código Java que pode ser compilado e executado em uma Máquina Virtual Java (*Java Virtual Machine - JVM*) como uma aplicação comum, conforme mostrado na **Figura 3.22**.

A vantagem do *compiler* em relação ao *player* é que este provoca um *overhead* no tempo gasto para instanciar os componentes, principalmente quando se tem um grande número deles. Este *overhead* é decorrente do fato de o *player* utilizar reflexão (*reflection*), característica que permite se descobrir os métodos de uma classe sem precisar conhecer o seu tipo ou a hierarquia a que pertence [Eckel, 2000], o que influencia na performance da

aplicação, segundo alguns programadores [Johnson, 1999]. Tal *overhead* não acontece quando se usa o *BML compiler* pois o mesmo gera um código Java sem reflexão (*reflection-free*).

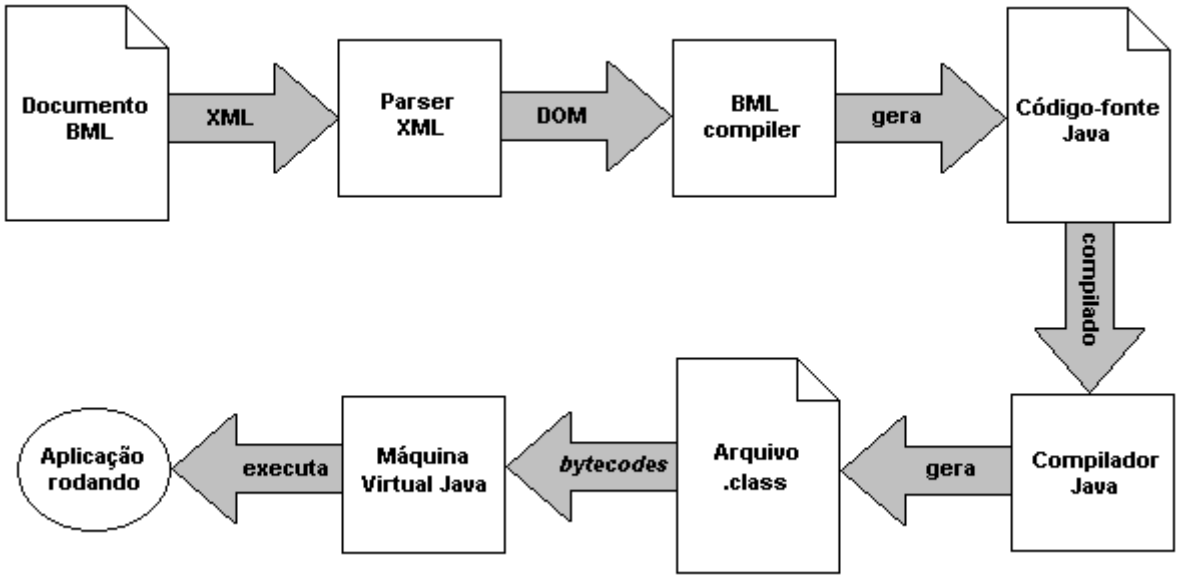


Figura 3.22. Executando uma aplicação com o BML compiler

No **capítulo 4**, que trata da validação do fwGF, serão mostrados exemplos de aplicações construídas com este *framework* e montadas com o uso da linguagem BML.

4. Validação do Framework

Neste capítulo são apresentados os detalhes referentes à fase de **validação do *framework***. Nesta fase, o objetivo principal é determinar se o *framework* implementado é um "software válido", ou seja, especificada uma aplicação de gerência de faltas, se é possível que, com a utilização do *framework*, se possa construir tal aplicação de forma que os seus requisitos originais sejam satisfeitos.

Entretanto, devido à natureza do domínio do problema em questão (a construção de aplicações para gerência de faltas), aplicar um *método formal* para a validação do *framework* se torna inviável tendo em vista sua complexidade e por não fazer parte do objetivo deste trabalho.

Além disso, a dificuldade de se aplicar um método formal, nesse caso, se deve ao fato do *espaço de aplicações de gerência de faltas possíveis* ser considerado *open ended*, ou seja, não apresentar "limites" bem definidos, conforme mostrado na **Figura 4.1**. Validar o fwGF, significa então caracterizar onde está a elipse desta figura.

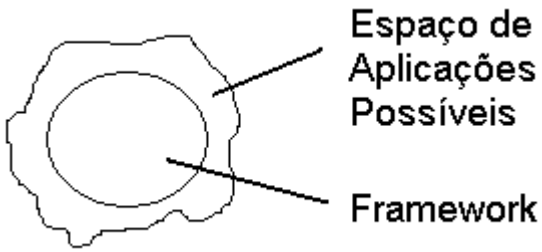


Figura 4.1. Espaços de Aplicações de Gerência de Faltas Possíveis e do *Framework*

Portanto, para validar o *framework*, optou-se por utilizar um *método não formal*. Dessa forma, algumas aplicações do espaço de possibilidades serão selecionadas e, com base na construção dessas aplicações, as conclusões sobre a validade do *framework* deverão ser obtidas.

Sendo assim, considerando-se o espaço de possibilidades representado como na **Figura 4.1**, pontos (ou seja, aplicações) dentro deste espaço serão escolhidos (ver **Figura 4.2**) de maneira tal que se possa obter um subconjunto de aplicações abrangente o bastante para se poder efetuar a validação do *framework*. Além disso, tão importantes quanto as aplicações que possam ser construídas com o *framework*, são aquelas que não possam ser implementadas com a atual configuração de componentes que constitui o mesmo, tendo em vista que estas aplicações colocarão em evidência as atuais limitações do *framework*.

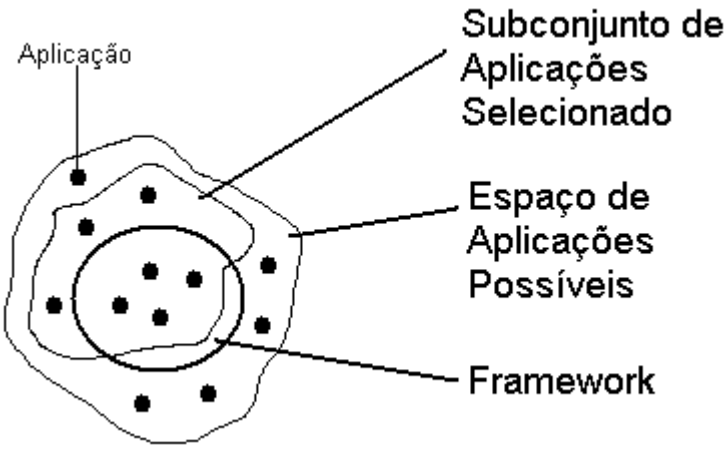


Figura 4.2. Representação do Subconjunto de Aplicações Selecionado para Validação do *Framework*.

Considerando-se que funções de gerência de faltas devem constar em um sistema de gerência de redes de computadores de maneira que este seja o mais abrangente possível, um subconjunto de aplicações capaz de fornecer suporte a tais funções foi selecionado. As aplicações são as seguintes:

- *Conectividade dos Dispositivos da Rede*: monitora o funcionamento dos dispositivos de uma rede, bem como suas interfaces de conexão;
- *Monitoramento de Erros em Interfaces de Comunicação*: monitora a taxa de erros de entrada ou saída na interfaces dos dispositivos da rede;
- *Monitoramento de Erros de Tamanho de Quadros e de Colisões*: monitora a taxa de erros no tamanho dos quadros e erros de colisão de quadros no meio físico;
- *Monitoramento de Tráfego em Interfaces de Comunicação*: monitora a taxa de tráfego de entrada e/ou saída nas interfaces de comunicação dos dispositivos;
- *Monitoramento de Saturação de Conexões*: monitora a quantidade de conexões estabelecidas (por exemplo, conexões TCP) com dispositivos (em geral , servidores) da rede;
- *Monitoramento Assíncrono da Rede*: monitora os eventos assíncronos (*traps*) gerados pelos dispositivos da rede decorrentes de situações especiais, como por exemplo, a queda de um enlace de comunicação;
- *Tempo de Resposta*: monitora o tempo transcorrido para que um dado (por exemplo, um pacote) trafegue pela rede de um dispositivo para outro;

- *Configuração de Alarmes na MIB RMON* (Detalhes sobre esta MIB podem ser encontrados em [Stallings, 1996]): configura limiares para o monitoramento de variáveis MIB e a geração de alarmes na MIB RMON;
- *Descobrimento e Estatísticas de Tráfego de Hosts na MIB RMON*: mantém uma tabela com os hosts descobertos através da análise dos dados que passam pelas interfaces de comunicação dos dispositivos gerenciados, bem como os valores de tráfego gerados por tais hosts, gerando alarmes quando o tráfego gerado por algum destes hosts atingir valores limites.

As aplicações acima citadas são entendidas como fazendo parte de um subconjunto que cobre bem o espaço para validação do *framework*. Na seção a seguir, estas aplicações são expostas mais detalhadamente.

Entretanto, antes de prosseguir com o detalhamento das aplicações, é importante que se faça uma descrição sobre a rede de computadores utilizada como estudo de caso para a construção das mesmas: a rede do Departamento de Sistemas e Computação (DSC) da Universidade Federal da Paraíba.

Como pode ser observado pela **Figura 4.3**, a rede do DSC é composta por quatro comutadores que interligam os diversos setores e laboratórios do departamento. Todos os comutadores e servidores dessa rede possuem agentes SNMP instalados que implementam a MIB padrão, MIB-2 [Rose & McCloghrie, 1995]. Entretanto, apenas os comutadores Leprecom e Central suportam a MIB RMON.

Portanto, as aplicações a serem construídas levam em consideração a configuração de rede apresentada acima. Contudo, não se faz necessário que cada aplicação deva gerenciar todos os dispositivos integrantes desta rede, tendo em vista que o gerenciamento pode ser feito de forma parcial, isto é, sobre apenas um subconjunto de dispositivos, desde que este seja suficiente para atender aos requisitos da aplicação.

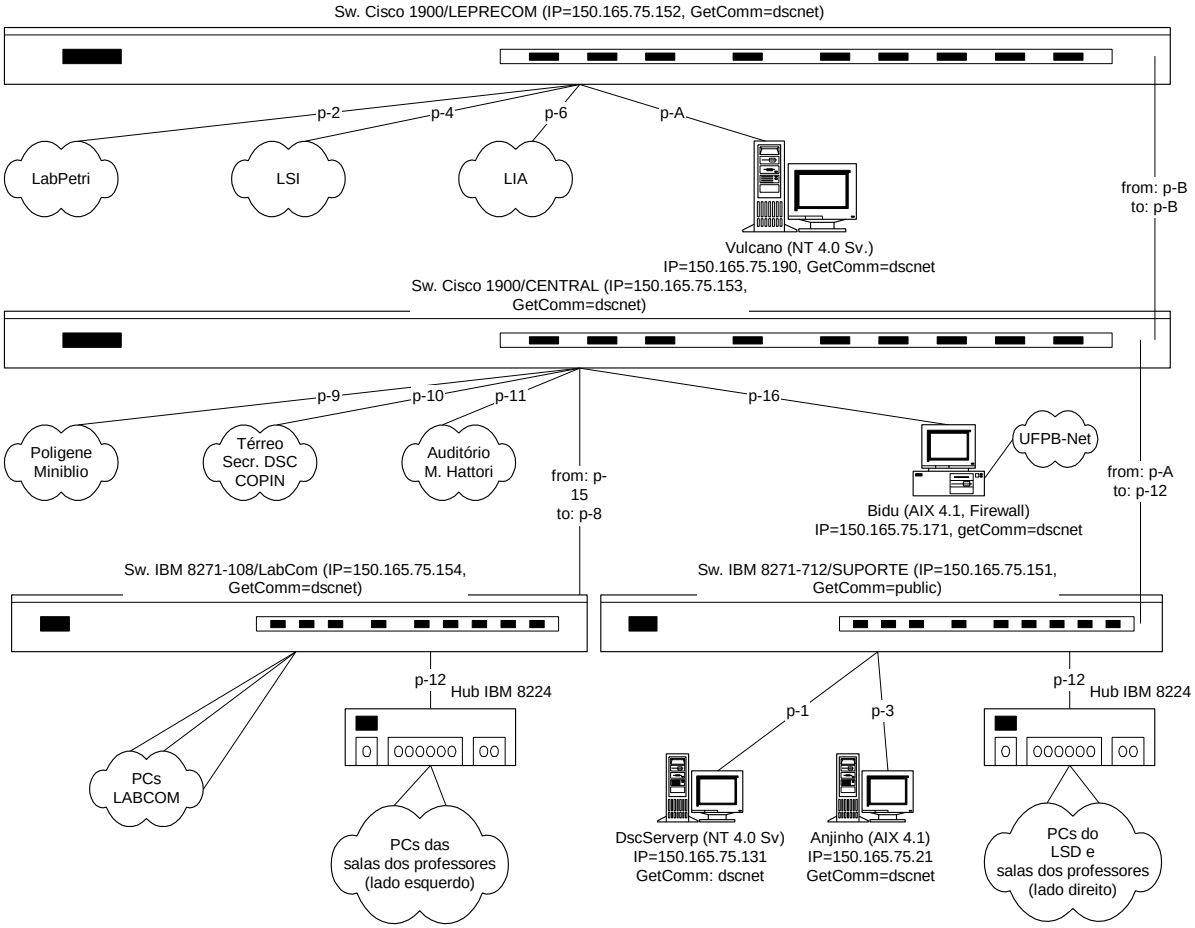


Figura 4.3. Topologia da Rede do DSC.

Antes de iniciar a aprensentação das aplicações selecionadas, vale salientar que, para representar cada componente instanciado de uma aplicação convencionou-se utilizar a seguinte representação:

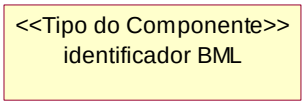


Figura 4.4. Representação de um componente.

onde "Tipo do Componente" representa o componente a ser instanciado e "identificador BML" é um valor no espaço de nomes utilizado para se instanciar um componente no script BML (*Bean Markup Language*)[7] [Kesselman & Duftler, 1999], em outras palavras, o valor do campo *id*.

[7] Veja também o capítulo Implementação do *Framework*.

Além disso, é importante frisar que, nos diagramas UML (*Unified Modeling Language*) [Rumbaugh *et al.*, 1999] utilizados para a representação dos componentes instanciados em uma aplicação, a relação entre dois componentes (A e B, por exemplo) pode ser representada conforme uma das três maneiras mostradas na **Figura 4.5**.

As relações apresentadas na **Figura 4.5** podem ser sintetizadas da seguinte forma:

(a) – Relação de *agregação* onde o componente B (cliente) é composto por A (fornecedor). Nesta relação, A pode também compor outros componentes, ou seja, A pode ser compartilhado. Representada por um "losango não cheio" do lado do cliente;

(b) – Esta também é uma relação de *agregação*, onde B (cliente) é composto por A (fornecedor), porém, A somente compõe B, não sendo compartilhado com nenhum outro. Representada por um "losango cheio" do lado do cliente;

(c) – Relação de associação entre dois componentes (A e B). É utilizada geralmente para representar uma conexão semântica entre os componentes. Utiliza-se esta relação, por exemplo, quando A é *listener* de B (padrão *Observer* [Gamma *et al*, 1995]), ou seja, quando A está "cadastrado" em B para receber eventos gerados por este.

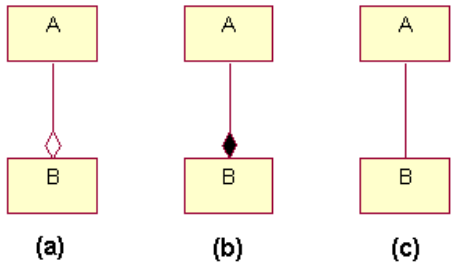


Figura 4.5. Representação de relações entre componentes.

Caso o leitor ainda não esteja familiarizado com a linguagem UML, em [Fowler & Scott, 1998] e [Rumbaugh *et al.*, 1999] pode-se obter mais informações sobre a mesma.

4.1. Aplicações de Gerência de Faltas Selecionadas para Validação do *Framework*

4.1.1. Aplicações Bem Sucedidas

4.1.1.1. Conectividade dos Dispositivos da Rede

O objetivo da aplicação aqui construída é de se verificar se os dispositivos gerenciados de uma rede estão funcionando. É importante salientar que não há nenhuma variável que indique se um dispositivo está ou não funcionando. Na realidade, o que realmente é feito é uma requisição de uma variável qualquer ao dispositivo (geralmente `system.sysUpTime`, no caso do SNMP) e, caso seja obtido um valor não nulo para essa variável, considera-se que o dispositivo está funcionando. Vale salientar ainda que esta requisição apenas indica se o serviço SNMP está ou não no ar e, portanto, caso a variável requisitada tenha valor nulo necessariamente não significa que o dispositivo não esteja funcionando. Contudo, na construção dessa aplicação considerou-se que se o serviço SNMP não está no ar, o dispositivo não está funcionando.

Além de se verificar se os dispositivos estão funcionando, nesta aplicação também é verificado o funcionamento das interfaces de comunicação que os conectam uns aos outros, pois a indicação de que um dispositivo está fora do ar não necessariamente indica que este não está funcionando, mas também pode ser devido a uma falha no enlace de comunicação através do qual se tem acesso àquele dispositivo.

Esta é uma aplicação bastante importante no gerenciamento de faltas, tendo em vista que o administrador da rede pode ter uma visão do funcionamento da rede gerenciada e ser informado sobre os dispositivos que não estão funcionando, tomando as devidas providências para solucionar o problema.

Nesta aplicação, apenas se está efetuando um gerenciamento parcial da rede do DSC, já que estão sendo monitorados somente os comutadores *leprecom* e *central* (**Figura 4.3**).

Configuração e instanciação dos componentes desta aplicação

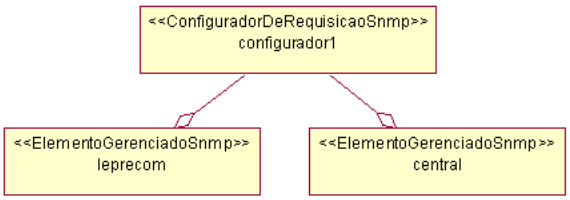


Figura 4.6. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Conectividade.

```
<?xml version="1.0"?>

<!--
connectivity_dsc.bml
Autor: Giovanni Almeida Santos

Objetivo da aplicação:
Esta aplicação tem como objetivo monitorar o estado dos dispositivos gerenciados de uma rede
bem como as interfaces que interconectam os mesmos. Para tal, estamos monitorando a variável
sysUpTime da MIB-II dos dispositivos e a ifOperStatus das interfaces.

Equipamentos monitorados:
Comutadores Cisco 1900 do LEPRECOM e Central.
-->


<script>

        <!-- Configuradores de Requisição -->

        <bean class="br.ufpb.dsc.grc.fwGF.ConfiguradorDeRequisicaoSnmp" id="configurador1">
                <property name="comunidade" value="dscnet"/>
                <property name="timeout" value="2"/>
                <property name="numeroRetransmissoes" value="0"/>
        </bean>
        ...
```

Figura 4.7. Trecho do script BML para configuração do componente ConfiguradorDeRequisiçãoSnmp

1. Um `ConfiguradorDeRequisicaoSnmp` (identificado por *configurador1*), conforme mostrado na **Figura 4.6**, para configurar os parâmetros de requisição para o protocolo

SNMP. Na **Figura 4.7**[8] é apresentado o trecho do script BML referente à configuração deste componente.

2. Dois componentes `ElementoGerenciadoSnmp` (*leprecom* e *central*) representando os dispositivos gerenciados da rede (**Figura 4.6**). Na **Figura 4.8** é apresentado o trecho do script BML referente à configuração destes.

3. Um componente `MibsSnmp` (**Figura 4.9**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado. O trecho do script BML para a configuração deste componente é mostrado na **Figura 4.10**. Para se informar o arquivo MIB a ser carregado deve-se chamar o método `loadMibModule` deste componente.

4. Quatro unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.9**), para a obtenção dos valores das variáveis MIB desejadas. Neste caso, as variáveis a ser coletadas são a `sysUpTime` de cada dispositivo e a `ifOperStatus` das interfaces desejadas. Na **Figura 4.11** é mostrado o trecho BML referente à configuração destes componentes[9].

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.9**) para agrupar várias unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas. O trecho BML referente à configuração destes componentes é mostrado na **Figura 4.12**.

6. Um componente para a geração de alarmes `GeradorDeAlarmesStatus` (**Figura 4.13**) que, ao receber um evento, imprime na tela uma mensagem de notificação para o responsável pela rede. Como este recebe eventos de um `GeradorDeEventoFalhaStatusEquipamento` ou de um `GeradorDeEventoFalhaStatusEnlace`, as mensagens impressas na tela indicam o estado dos dispositivos e das interfaces, isto é, se estes entraram ou saíram do ar. O trecho BML da configuração deste componente é mostrado na **Figura 4.14**.

[8] Os valores das propriedades que dizem respeito a tempo (timeout, periodo, etc.), apresentados nos scripts deste capítulo, são dados em segundos.

[9] Por default, quando um *oid* (*object identifier*) é informado sem um ponto inicial, considera-se que tal *oid* faz parte da MIB-2, o qual possui o prefixo padrão .1.3.6.1.2.1.

```
...
<!-- Elementos Gerenciados -->

<bean class="br.ufpb.dsc.grc.fwGF.ElementoGerenciadoSnmp" id="leprecom">
        <property name="nome" value="150.165.75.152"/>
        <property name="endIp" value="150.165.75.152"/>
        <property name="nomeResponsavel" value="Pedro Sérgio Nicolletti"/>
        <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
        <property name="configuradorDeRequisicao">
                <bean source="configurador1"/>
        </property>
</bean>

<bean class="br.ufpb.dsc.grc.fwGF.ElementoGerenciadoSnmp" id="central">
        <property name="nome" value="150.165.75.153"/>
        <property name="endIp" value="150.165.75.153"/>
        <property name="nomeResponsavel" value="Pedro Sérgio Nicolletti"/>
        <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
        <property name="configuradorDeRequisicao">
                <bean source="configurador1"/>
        </property>
</bean>
...
```

Figura 4.8. Trecho do script BML para configuração dos componentes ElementoGerenciadoSnmp

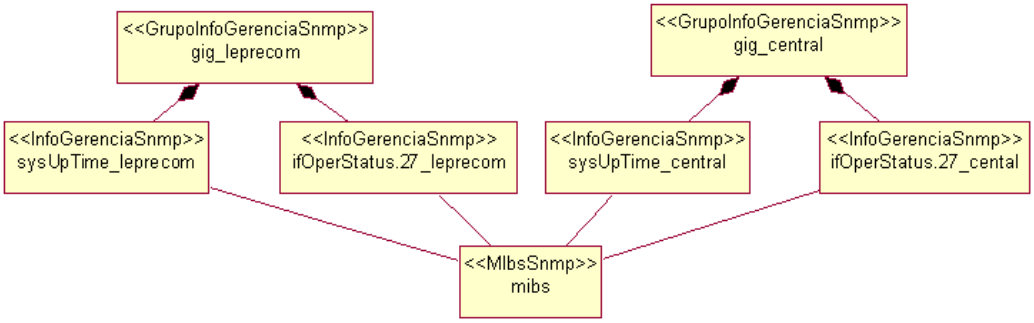


Figura 4.9. Componentes GrupoInfoGerenciaSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Conectividade.

```
...
<!-- Carga de MIBs -->

<bean class="br.ufpb.dsc.grc.fwGF.MibsSnmp" id="mibs">
        <call-method name="loadMibModule">
                <string>..\..\mibs\RFC1213-MIB</string>
        </call-method>
</bean>
...
```

Figura 4.10. Trecho do script BML para configuração do componente MibsSnmp.

```
...
<!—Informações de Gerência de LEPRECOM -->

<bean class="br.ufpb.dsc.grc.fwGF.InfoGerenciaSnmp" id="sysUpTime_leprecom">
        <property name="oid" value="system.sysUpTime.0"/>
        <property name="nome" value="sysUpTime_leprecom"/>
</bean>

<bean class="br.ufpb.dsc.grc.fwGF.InfoGerenciaSnmp" id="ifOperStatus.27_ leprecom">
        <property name="oid" value=" interfaces.ifTable.ifEntry.ifOperStatus.27"/>
        <property name="nome" value="interface 27 de leprecom"/>
</bean>

<!—Informações de Gerência de CENTRAL -->

<bean class="br.ufpb.dsc.grc.fwGF.InfoGerenciaSnmp" id="sysUpTime_central">
        <property name="oid" value=" system.sysUpTime.0"/>
        <property name="nome" value="sysUpTime_central"/>
</bean>

<bean class="br.ufpb.dsc.grc.fwGF.InfoGerenciaSnmp" id="ifOperStatus.27_ central">
        <property name="oid" value=" interfaces.ifTable.ifEntry.ifOperStatus.27"/>
        <property name="nome" value="interface 27 de central"/>
</bean>
...
```

Figura 4.11. Trecho do script BML para configuração dos componentes InfoGerenciaSnmp

```
...
<!-- Grupos de Informações de Gerência -->

<bean class="br.ufpb.dsc.grc.fwGF.GrupoInfoGerenciaSnmp" id="gig_leprecom">
        <property name="nome" value="Grupo leprecom"/>
        <call-method name="addInfoGerencia">
                <bean source="sysUpTime_leprecom"/>
        </call-method>
        <call-method name="addInfoGerencia">
                <bean source="ifOperStatus.27_leprecom"/>
        </call-method>
</bean>

<bean class="br.ufpb.dsc.grc.fwGF.GrupoInfoGerenciaSnmp" id="gig_central">
        <property name="nome" value="Grupo central"/>
        <call-method name="addInfoGerencia">
                <bean source="sysUpTime_central"/>
        </call-method>
        <call-method name="addInfoGerencia">
                <bean source="ifOperStatus.27_ central"/>
        </call-method>
</bean>
...
```

Figura 4.12. Trecho do script BML para configuração dos componentes GrupoInfoGerenciaSnmp

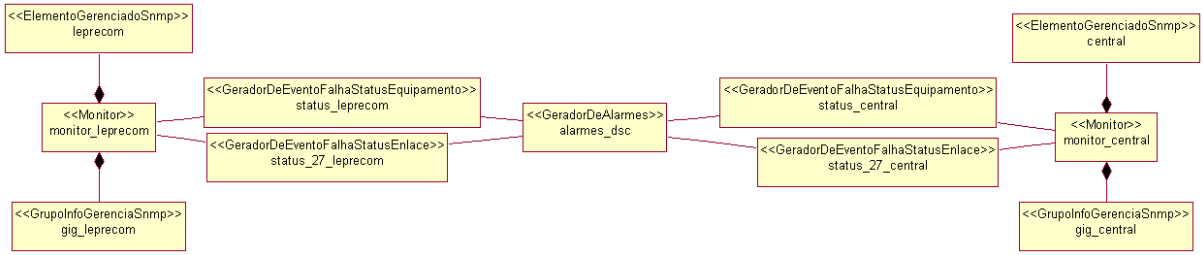


Figura 4.13. Componentes Monitor, GeradorDeEventoFalhaStatusEquipamento, GeradorDeEventoFalhaStatusEnlace e GeradorDeAlarmesStatus para a aplicação Conectividade.

```
...
<!-- Geradores de Alarmes -->

<bean class="br.ufpb.dsc.grc.fwGF.GeradorDeAlarmesStatus" id="alarmes_dsc">
        <property name="nome" value="Gerador de Alarmes - DSC"/>
</bean>
...
```

Figura 4.14. Trecho do script BML para configuração do componente GeradorDeAlarmesStatus

7. Quatro componentes geradores de eventos que monitoram o estado dos dispositivos da rede (`GeradorDeEventoFalhaStatusEquipamento`) e de suas interfaces (`GeradorDeEventoFalhaStatusEnlace`), conforme mostrado na **Figura 4.13**. O trecho do script BML para a configuração destes componentes é mostrado na **Figura 4.15** e na **Figura 4.16**.

```
	...
	<!-- Geradores de Eventos -->

	<bean 		class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEquipamento"
id="status_leprecom">
		<property name="nome" value="Status de leprecom"/>
		<property name="nomeIg">
			<property target="sysUpTime_leprecom" name="nome"/>
		</property>
		<property name="responsavel" value="Pedro Sérgio Nicolletti"/>
		<property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
		<property name="descricao" value="O comutador do leprecom está down."/>
		<property name="descricaoEventoRearm" value="O comutador do leprecom está
up."/>
		<property name="prioridade">
			<field name="PRIORIDADE_ALTA"/>
		</property>
		<call-method name="addEventoFalhaListener">
			<bean source="alarmes_dsc"/>
		</call-method>
	</bean>

	<bean 			class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEnlace"
id="status_27_leprecom">
		<property name="nome" value="Status da interface 27 de leprecom"/>
		<property name="nomeIg">
			<property target="ifOpeStatus.27_ leprecom" name="nome"/>
		</property>
		<property name="responsavel" value="Pedro Sérgio Nicolletti"/>
		<property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
		<property name="descricao" value="interface 27 de leprecom está down/testing"/>
		<property name="descricaoEventoRearm" value="interf. 27 de leprecom está up."/>
		<call-method name="addEventoFalhaListener">
			<bean source="alarmes_dsc"/>
		</call-method>
	</bean>
	...
```

Figura 4.15. Trecho do script BML para configuração dos componentes GeradorDeEventoFalhaStatusEquipamento e GeradorDeEventoFalhaStatusEnlace

```
        ...
        <bean                class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEquipamento"
id="status_central">
                <property name="nome" value="Status de central"/>
                <property name="nomeIg">
                        <property target="sysUpTime_central" name="nome"/>
                </property>
                <property name="responsavel" value="Pedro Sérgio Nicolletti"/>
                <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
                <property name="descricao" value="O comutador central está down."/>
                <property name="descricaoEventoRearm" value="O comutador central está up."/>
                <property name="prioridade">
                        <field name="PRIORIDADE_ALTA"/>
                </property>
                <call-method name="addEventoFalhaListener">
                        <bean source="alarmes_dsc"/>
                </call-method>
        </bean>

        <bean                      class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEnlace"
id="status_27_central">
                <property name="nome" value="Status da interface 27 de central"/>
                <property name="nomeIg">
                        <property target="ifOpeStatus.27_ central" name="nome"/>
                </property>
                <property name="responsavel" value="Pedro Sérgio Nicolletti"/>
                <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
                <property name="descricao" value="interface 27 de central está down/testing"/>
                <property name="descricaoEventoRearm" value="interf. 27 de central está up."/>
                <call-method name="addEventoFalhaListener">
                        <bean source="alarmes_dsc"/>
                </call-method>
        </bean>
        ...
```

Figura 4.16. Trecho do script BML para configuração dos componentes GeradorDeEventoFalhaStatusEquipamento e GeradorDeEventoFalhaStatusEnlace (continuação).

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.13**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados. Na **Figura 4.17** encontra-se o trecho BML para a configuração destes monitores.

```
	...
	<!-- Monitores -->

	<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_leprecom">
		<property name="periodo" value="60"/>
		<property name="eg">
			<bean source="leprecom"/>
		</property>
		<property name="gig">
			<bean source="gig_leprecom"/>
		</property>
		<call-method name="addMonitorListener">
			<bean source="status_leprecom"/>
		</call-method>
		<call-method name="addMonitorListener">
			<bean source="status_27_leprecom"/>
		</call-method>
		<call-method name="start"/>
	</bean>

	<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_central">
		<property name="periodo" value="60"/>
		<property name="eg">
			<bean source="central"/>
		</property>
		<property name="gig">
			<bean source="gig_central"/>
		</property>
		<call-method name="addMonitorListener">
			<bean source="status_central"/>
		</call-method>
		<call-method name="addMonitorListener">
			<bean source="status_27_central"/>
		</call-method>
		<call-method name="start"/>
	</bean>


</script>
```

Figura 4.17. Trecho do script BML para configuração dos componentes Monitor

Ao término da construção desta aplicação, pode-se concluir que o *framework* atendeu aos requisitos iniciais da mesma de forma satisfatória.

Entretanto, a solução apresentada pelo *framework* para a construção de aplicações deste tipo é ainda parcial, pois é desejável que uma aplicação assim possa realizar uma correlação de eventos baseada na topologia da rede.

Considere-se, por exemplo, que para uma requisição feita pela estação de gerência da rede (que está rodando esta aplicação) chegar ao comutador *central*, tenha antes que passar pelo comutador *leprecom*. Portanto, para se ter acesso a central é necessário que o *leprecom* esteja funcionando (além da interface de conexão entre estes dois swithces).

Assim sendo, caso o *leprecom* pare de funcionar, a aplicação aqui construída irá gerar dois alarmes indicando dispositivos fora do ar (sem levar em consideração ainda os alarmes gerados devido às interfaces de comunicação): um para *leprecom* e outro para *central*, independente de central está ou não funcionando.

Portanto, como o conjunto de correlatores presentes atualmente no *framework* ainda é bastante limitado, não é possível, no momento, construir uma aplicação como a construída aqui e que leve em consideração a topologia da rede. Entretanto, tal problema pode ser resolvido com a adição de um novo componente ao *framework* que contenha informação topológica (Em [Meira & Nogueira, 1997] e [Meira, 1997] também podem ser encontrados vários outros métodos e algoritmos para a correlação de eventos e que podem ser perfeitamente aplicados na construção de correlatores para o fwGF).

4.1.1.2. Monitoramento de Erros em Interfaces de Comunicação

O objetivo desta aplicação é testar a integridade dos dados que entram nas interfaces de interconexão dos dispositivos da rede. Para isso, pode-se verificar erros nos pacotes que entram em cada interface, gerando eventos quando os limiares estabelecidos forem ultrapassados. A importância dessa aplicação está no fato de que uma taxa de erros elevada pode indicar um problema em um equipamento ou interface de comunicação. Assim sendo, quando um limiar configurado numa aplicação deste tipo é ultrapassado, isso pode indicar que a interface ou o próprio dispositivo não está funcionando perfeitamente.

Com o intuito de reduzir o espaço de gerenciamento da aplicação aqui construída, apenas se está efetuando um gerenciamento parcial da rede do DSC, já que estão sendo monitorados somente os comutadores *leprecom* e *central* (**Figura 4.3**).

Configuração e instanciação dos componentes desta aplicação

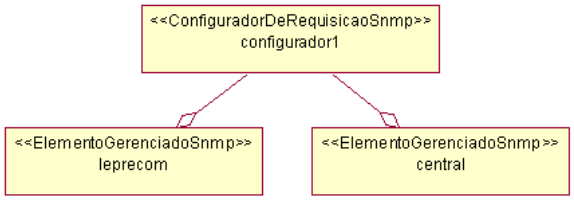


Figura 4.18. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Monitoramento de Erros em Interfaces de Comunicação.

1. Um `ConfiguradorDeRequisicaoSnmp` (*configurador1*), conforme mostrado na **Figura 4.18**, para configurar os parâmetros de requisição para o protocolo SNMP.

   Propriedades de *configurador1*:

   comunidade: dscnet

   timeout: 2

   numeroRetransmissoes: 0

2. Dois componentes `ElementoGerenciadoSnmp` (*leprecom* e *central*) representando os dispositivos gerenciados da rede (**Figura 4.18**).

   Propriedades de *leprecom*:

   nome: 150.165.75.152

   endIp: 150.165.75.152

   Propriedades de *central*:

   nome: 150.165.75.153

   endIp: 150.165.75.153

3. Um componente `MibsSnmp` (**Figura 4.19**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Nove unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.19**), sendo cinco associadas a interfaces do *comutador leprecom* e quatro a interfaces do *comutador central*, para a obtenção dos valores da variável MIB `ifInErrors` de cada uma delas.

   Propriedades de *ifInErrors.2_leprecom*:

   oid: interfaces.ifTable.ifEntry.ifInErrors.2

   nome: ifInErrors.2_leprecom

Propriedades de *ifInErrors.4_leprecom*:

oid: interfaces.ifTable.ifEntry.ifInErrors.4

nome: ifInErrors.4_leprecom

Propriedades de *ifInErrors.6_leprecom*:

oid: interfaces.ifTable.ifEntry.ifInErrors.6

nome: ifInErrors.6_leprecom

Propriedades de *ifInErrors.26_leprecom*:

oid: interfaces.ifTable.ifEntry.ifInErrors.26

nome: ifInErrors.26_leprecom

Propriedades de *ifInErrors.27_leprecom*:

oid: interfaces.ifTable.ifEntry.ifInErrors.27

nome: ifInErrors.27_leprecom

Propriedades de *ifInErrors.9_central*:

oid: interfaces.ifTable.ifEntry.ifInErrors.9

nome: ifInErrors.9_central

Propriedades de *ifInErrors.15_central*:

oid: interfaces.ifTable.ifEntry.ifInErrors.15

nome: ifInErrors.15_central

Propriedades de *ifInErrors.16_central*:

oid: interfaces.ifTable.ifEntry.ifInErrors.16

nome: ifInErrors.16_central

Propriedades de *ifInErrors.26_central*:

oid: interfaces.ifTable.ifEntry.ifInErrors.26

nome: ifInErrors.26_central

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.19**) para agrupar várias unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas através da chamada ao método `addInfoGerencia(InfoGerencia info)` deste componente.

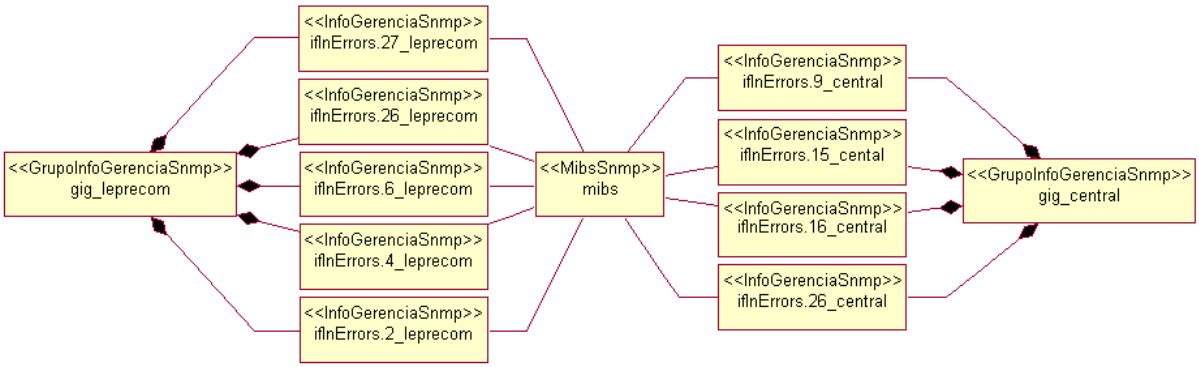


Figura 4.19. Componentes GrupoInfoGerenciaSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Monitoramento de Erros em Interfaces de Comunicação.

6. Um componente para a geração de alarmes `GeradorDeAlarmesTaxa` (**Figura 4.20**) que, ao receber um evento, imprime na tela uma mensagem de notificação para o responsável pela rede. Este componente recebe eventos de um `GeradorDeEventoFalhaComHisterese` e imprime na tela o valor da propriedade `descricao` do gerador caso o `limiar` estabelecido tenha sido ultrapassado ou o valor de `descricaoEventoRearm` caso o valor ultrapassado seja o de `rearm` (limiar oposto). O GeradorDeEventoFalhaComHisterese usa o mecanismo de histerese, o qual serve para limitar o número de eventos gerado. Este mecanismo gera um evento quando um limiar é ultrapassado em um determinado sentido, sendo que nenhum outro evento é gerado até que o limiar oposto seja atingido [Waldbusser, 1995].

7. Nove geradores de eventos (`GeradorDeEventoFalhaComHisterese`) para analisarem a taxa de erros em cada uma das nove interfaces monitoradas por esta aplicação (**Figura 4.20**).

   Propriedades comuns aos componentes abaixo:

   limiar: 2

   rearm: 0

   nome: Alta Taxa de Erros de Entrada

   responsavel: Pedro Sérgio Nicolletti

   contatoResponsavel: peter@dsc.ufpb.br

   nomeEventoRearm: Taxa de Erros de Entrada OK

   Propriedades de *erros.2_leprecom*:

   descricao: Taxa de erros de entrada na interface 2 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 2 de leprecom OK.

nomeIg: ifInErrors.2_leprecom

Propriedades de *erros.4_leprecom*:

descricao: Taxa de erros de entrada na interface 4 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 4 de leprecom OK.

nomeIg: ifInErrors.4_leprecom

Propriedades de *erros.6_leprecom*:

descricao: Taxa de erros de entrada na interface 6 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 6 de leprecom OK.

nomeIg: ifInErrors.6_leprecom

Propriedades de *erros.26_leprecom*:

descricao: Taxa de erros de entrada na interface 26 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 26 de leprecom OK.

nomeIg: ifInErrors.26_leprecom

Propriedades de *erros.27_leprecom*:

descricao: Taxa de erros de entrada na interface 27 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 27 de leprecom OK.

nomeIg: ifInErrors.27_leprecom

Propriedades de *erros.9_central*:

descricao: Taxa de erros de entrada na interface 9 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 9 de leprecom OK.

nomeIg: ifInErrors.9_central

Propriedades de *erros.15_central*:

descricao: Taxa de erros de entrada na interface 15 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 15 de leprecom OK.

nomeIg: ifInErrors.15_central

Propriedades de *erros.16_central*:

descricao: Taxa de erros de entrada na interface 16 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 16 de leprecom OK.

nomeIg: ifInErrors.16_central

Propriedades de *erros.26_central*:

descricao: Taxa de erros de entrada na interface 26 de leprecom está alta.

descricaoEventoRearm: Taxa de erros/entrada na interface 26 de leprecom OK.

nomeIg: ifInErrors.26_central

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.20**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados.

   Propriedades de *monitor_leprecom*:

   periodo: 60

   Propriedades de *monitor_central*:

   periodo: 60

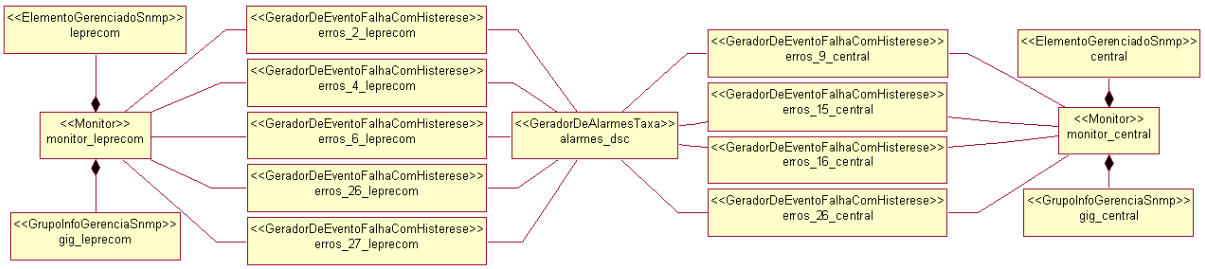


Figura 4.20. Componentes Monitor, GeradorDeEventoFalhaComHisterese e GeradorDeAlarmesStatus para a aplicação Monitoramento de Erros em Interfaces de Comunicação.

Com relação à construção desta aplicação, pode-se concluir que o *framework* atendeu satisfatoriamente aos requisitos originais da mesma.

4.1.1.3. Monitoramento de Erros de Tamanho de Quadros e de Colisões

A aplicação aqui desenvolvida tem o objetivo de testar a integridade dos quadros que entram ou saem nas diversas interfaces dos dispositivos gerenciados observando a ocorrência de erros no tamanho dos quadros e erros de colisão. Limiares devem ser estabelecidos (e valores de *rearm*) para que sejam gerados eventos quando os mesmos forem cruzados.

Configuração e instanciação dos componentes desta aplicação

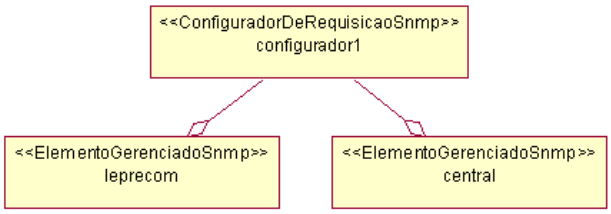


Figura 4.21. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Monitoramento de Erros de Tamanho de Quadros e de Colisões.

1. Um `ConfiguradorDeRequisicaoSnmp` (*configurador1*), conforme mostrado na **Figura 4.21**, para configurar os parâmetros de requisição para o protocolo SNMP.

2. Dois componentes `ElementoGerenciadoSnmp` (*leprecom* e *central*) representando os dispositivos gerenciados da rede (**Figura 4.21**).

3. Um componente `MibsSnmp` (**Figura 4.22**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Doze unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.22**), sendo cinco associadas a interfaces do *comutador leprecom* e quatro a interfaces do *comutador central*, para a obtenção dos valores das variáveis MIB `etherStatsCollisions` *(oid* rmon.statistics.etherStatsTableetherStatsEntry.etherStatsCollisions*)*, `etherStatsOversizePkts` *(oid* rmon.statistics.etherStatsTable.etherStatsEntry. etherStatsOversizePkts) e `etherStatsUndersizePkts` *(oid* rmon.statistics. etherStatsTable.etherStatsEntry.etherStatsUndersizePkts) de cada uma delas.

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.22**) para agrupar várias unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas através da chamada ao método `addInfoGerencia(InfoGerencia info)` deste componente.

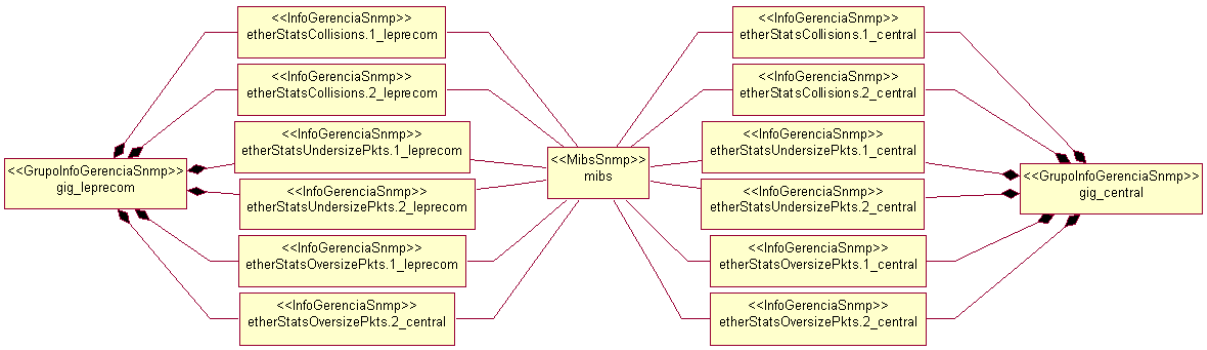


Figura 4.22. Componentes GrupoInfoGerenciaSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Monitoramento de Erros de Tamanho de Quadros e de Colisões.

6. Um componente para a geração de alarmes `GeradorDeAlarmesTaxa` (**Figura 4.23**) que, ao receber um evento, imprime na tela uma mensagem de notificação para o responsável pela rede. Este componente recebe eventos de um `GeradorDeEventoFalhaComHisterese` e imprime na tela o valor da propriedade descrição do gerador caso o limiar estabelecido tenha sido ultrapassado ou o valor de descriçãoEventoRearm caso o valor ultrapassado seja o de rearm.

7. Doze geradores de eventos (`GeradorDeEventoFalhaComHisterese`) para analisarem a taxa de erros em cada uma das nove interfaces monitoradas por esta aplicação (**Figura 4.23**).

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.23**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados

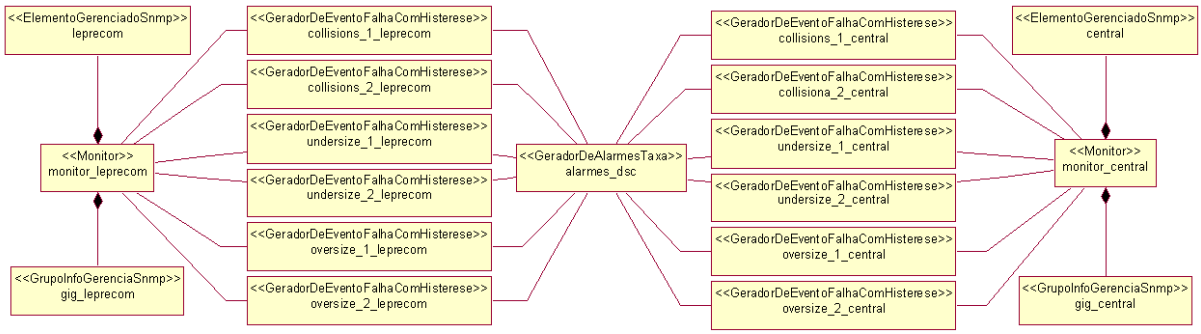


Figura 4.23. Componentes Monitor, GeradorDeEventoFalhaComHisterese e GeradorDeAlarmesTaxa para a aplicação Monitoramento de Erros de Tamanho de Quadros e de Colisões.

Esta é uma aplicação para a qual o *framework* conseguiu atender aos seus requisitos iniciais.

4.1.1.4. Monitoramento de Tráfego em Interfaces de Comunicação

Esta aplicação tem como objetivo verificar o tráfego de entrada (total, unicast, broadcast, multicast) em cada interface dos dispositivos da rede. Limiares devem ser estabelecidos (e valores de *rearm*) para que sejam gerados eventos quando os mesmos forem cruzados. Por motivos de simplicidade, a aplicação utilizada como exemplo apenas monitora o tráfego total de entrada e saída das interfaces, não preocupando-se com o tráfego unicast, broadcast e multicast.

Com relação à importância desta aplicação para o gerenciamento de faltas, pode-se dizer que a mesma é bastante relevante, tendo em vista que se pode prever congestionamentos em enlaces de comunicação e, assim, tomarem-se as providências cabíveis para se prevenir a ocorrência de faltas.

Nesta aplicação também se está efetuando apenas um gerenciamento parcial sobre a rede do DSC. Os dispositivos gerenciados aqui são os comutadores *leprecom* e *central*. Além disso, nestes equipamentos apenas algumas de suas interfaces estão sendo monitoradas (as consideradas mais importantes em relação a tráfego e prioridade de funcionamento, ou seja, as que dão acesso aos laboratórios e servidores do DSC).

Configuração e instanciação dos componentes desta aplicação

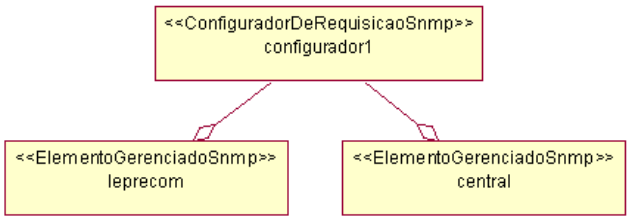


Figura 4.24. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Monitoramento de Tráfego em Interfaces de Comunicação.

1. Um `ConfiguradorDeRequisicaoSnmp` (*configurador1*), conforme mostrado na **Figura 4.24**, para configurar os parâmetros de requisição para o protocolo SNMP.

2. Dois componentes `ElementoGerenciadoSnmp` (*leprecom* e *central*) representando os dispositivos gerenciados da rede (**Figura 4.24**).

3. Um componente `MibsSnmp` (**Figura 4.25**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Nove unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.25**), sendo cinco associadas a interfaces do *comutador leprecom* e quatro a interfaces do *comutador central*, para a obtenção dos valores da variável MIB `ifInOctets` *(oid* interfaces.ifTable.ifEntry.ifInOctets) de cada uma delas.

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.25**) para agrupar várias unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas através da chamada ao método `addInfoGerencia(InfoGerencia info)` deste componente.

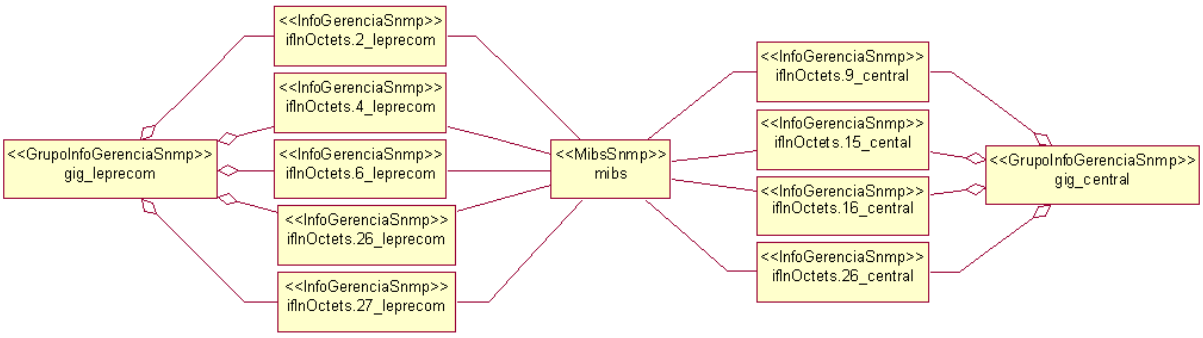


Figura 4.25. Componentes GrupoInfoSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Monitoramento de Tráfego em Interfaces de Comunicação.

6. Um componente para a geração de alarmes `GeradorDeAlarmesTaxa` (**Figura 4.26**) que, ao receber um evento, imprime na tela uma mensagem de notificação para o responsável pela rede. Este componente recebe eventos de um `GeradorDeEventoFalhaComHisterese` e imprime na tela o valor da propriedade descrição do gerador caso o limiar estabelecido tenha sido ultrapassado ou o valor de descriçãoEventoRearm caso o valor ultrapassado seja o de rearm.

7. Nove geradores de eventos (`GeradorDeEventoFalhaComHisterese`) para analisarem a taxa de erros em cada uma das nove interfaces monitoradas por esta aplicação (**Figura 4.26**).

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.26**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados

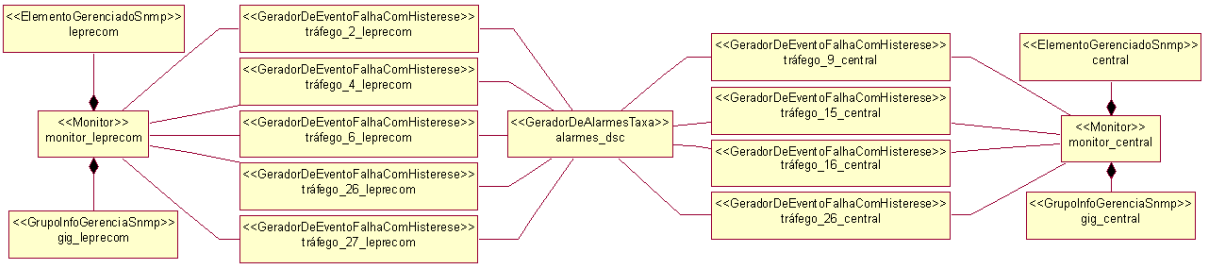


Figura 4.26. Componentes Monitor, GeradorDeEventoFalhaComHisterese e GeradorDeAlarmesTaxa para a aplicação Monitoramento de Tráfego em Interfaces de Comunicação.

Com relação à construção desta aplicação, pode-se concluir que o *framework* atendeu satisfatoriamente aos requisitos originais da mesma.

4.1.1.5. Monitoramento de Espaço em Disco de Servidores

O objetivo desta aplicação é monitorar a ocupação dos discos rígidos dos servidores da rede do DSC. O valor referente à ocupação de um disco é dado pela variável MIB *host.hrStorage.hrStorageTable.hrStorageEntry.hrStorageUsed.X*, onde X indica o índice do disco.

Esta é uma aplicação bastante importante no gerenciamento de faltas, tendo em vista que com sua utilização o responsável pela administração dos servidores gerenciados pode ser avisado tão logo a ocupação dos dispositivos de armazenamento destes servidores ultrapasse os limiares estabelecidos. Dessa forma, o responsável pode então tomar as providências cabíveis e evitar a ocorrência de danos maiores.

Na construção desta aplicação, se está levando em consideração um gerenciamento parcial dos servidores da rede do DSC, mais especificamente, os servidores *anjinho* e *bidu* são os que estão sendo gerenciados.

Configuração e instanciação dos componentes desta aplicação

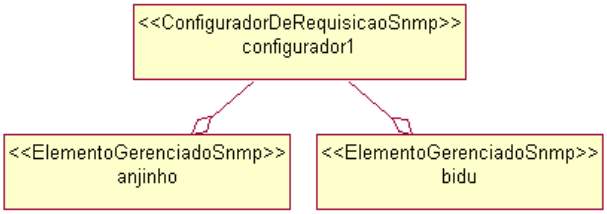


Figura 4.27. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Monitoramento de Espaço em Disco de Servidores.

1. Um `ConfiguradorDeRequisicaoSnmp` (*configurador1*), conforme mostrado na **Figura 4.27**, para configurar os parâmetros de requisição para o protocolo SNMP.

2. Dois componentes `ElementoGerenciadoSnmp` (*anjinho* e *bidu*) representando os servidores gerenciados da rede (**Figura 4.27**).

3. Um componente `MibsSnmp` (**Figura 4.28**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Duas unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.28**), cada uma associada a um disco rígido de cada servidor para a coleta dos valores da variável `host.hrStogare.hrStorageTable.hrStorageEntry.hrStoragedUsed.X`, onde X indica o índice associado ao disco.

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.28**) para agrupar as unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas através da chamada ao método `addInfoGerencia(InfoGerencia info)` deste componente.

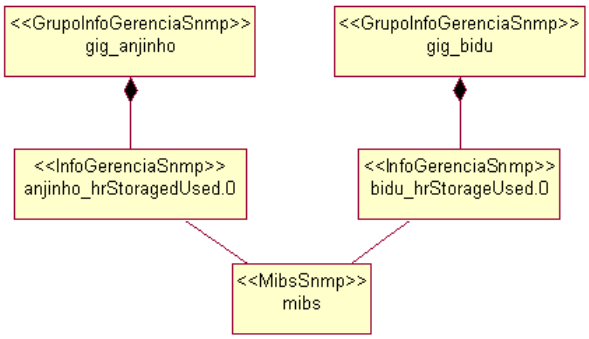


Figura 4.28. Componentes GrupoInfoSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Monitoramento de Espaço em Disco de Servidores.

6. Um componente para a geração de alarmes `GeradorDeAlarmesTaxa` (**Figura 4.29**) para notificar o responsável pela rede através da impressão na tela de uma mensagem de notificação. Este componente recebe eventos de um `GeradorDeEventoFalhaComHisterese` e imprime na tela o valor da propriedade `descricao` do gerador caso o `limiar` estabelecido tenha sido ultrapassado ou o valor de `descricaoEventoRearm` caso o valor ultrapassado seja o de `rearm`.

7. Dois geradores de eventos (`GeradorDeEventoFalhaComHisterese`) para analisarem o espaço ocupado nos dispositivos de armazenamento gerenciados (**Figura 4.29**).

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.29**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados.

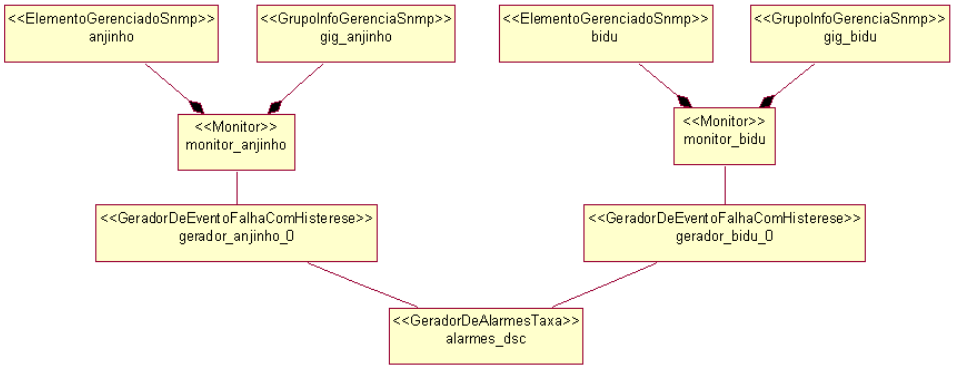


Figura 4.29. Componentes Monitor, GeradorDeEventoFalhaComHisterese e GeradorDeAlarmesTaxa para a aplicação Monitoramento de Espaço em Disco de Servidores.

Com relação à construção desta aplicação, pode-se concluir que o *framework* atendeu satisfatoriamente aos requisitos originais da mesma.

4.1.1.6. Monitoramento de Saturação de Conexões

O objetivo desta aplicação é verificar se o número de conexões TCP estabelecidas (`tcpCurrEstab`) com os servidores da rede ultrapassou um limiar previamente estabelecido. Sendo assim, pode-se gerar um alarme para o administrador da rede para que o mesmo tome as providências. Esta aplicação é bastante útil, por exemplo, para *sites* Web com um limite no número de usuários simultâneos. Dessa forma, o responsável pelo site pode ser informado quando o número de conexões simultâneas estiver atingindo valores altos e, assim, ele pode tomar as providências necessárias para a solução do problema.

Nesta aplicação apenas parte da rede do DSC está sendo gerenciada, mais especificamente, os servidores *anjinho* e *bidu*.

Configuração e instanciação dos componentes desta aplicação

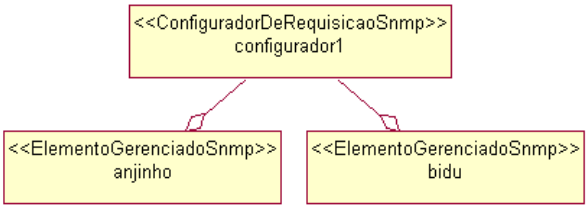


Figura 4.30. Componentes ConfiguradorDeRequisiçãoSnmp e ElementoGerenciadoSnmp para a aplicação Monitoramento de Saturação de Conexões.

1. Um `ConfiguradorDeRequisicaoSnmp` (*configurador1*), conforme mostrado na **Figura 4.30**, para configurar os parâmetros de requisição para o protocolo SNMP.

2. Dois componentes `ElementoGerenciadoSnmp` (*leprecom* e *central*) representando os dispositivos gerenciados da rede (**Figura 4.30**).

3. Um componente `MibsSnmp` (**Figura 4.31**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Duas unidades de informação de gerência `InfoGerenciaSnmp` (**Figura 4.31**), cada uma associada a um servidor gerenciado, para a obtenção dos valores da variável MIB *tcpCurrEstab* (*oid* tcp.tcpCurrestab.0) de cada um deles.

5. Dois grupos de informação de gerência `GrupoInfoGerenciaSnmp` (**Figura 4.31**) para agrupar várias unidades de informação a ser enviadas na requisição. Em cada grupo, deve-se adicionar as unidades de informação desejadas e previamente instanciadas através da chamada ao método `addInfoGerencia(InfoGerencia info)` deste componente.

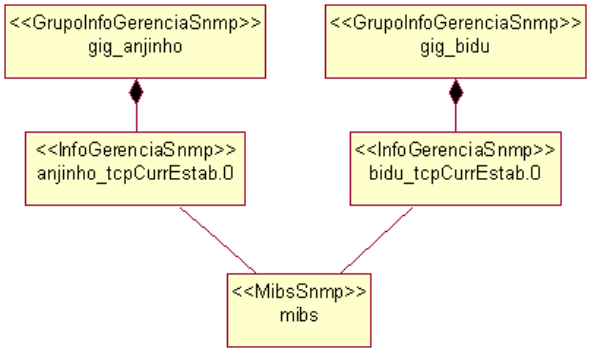


Figura 4.31. Componentes GrupoInfoSnmp, InfoGerenciaSnmp e MibsSnmp para a aplicação Monitoramento de Saturação de Conexões.

6. Um gerador de alarmes (`GeradorDeEventoFalhaTaxa`) para notificar ao administrador caso a quantidade de conexões TCP estabelecidas com os servidores ultrapasse os limiares estabelecidos no geradores de eventos.

7. Dois geradores de eventos (`GeradorDeEventoFalhaComHisterese`) para analisarem a quantidade de conexões TCP estabelecidas com os servidores (**Figura 4.32**).

8. Dois monitores (`Monitor`) responsáveis por controlar o envio de requisições dos grupos de informação de gerência pelos elementos gerenciados a cada intervalo de tempo dado pelo valor da propriedade período (**Figura 4.32**). Em cada monitor são cadastrados os consumidores de eventos (*listeners*) interessados em receber eventos por ele gerados

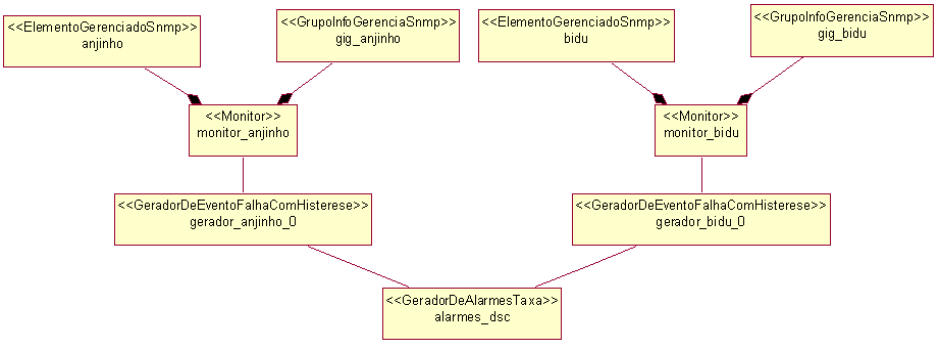


Figura 4.32. Componentes Monitor, GeradorDeEventoFalhaComHisterese e GeradorDeAlarmesTaxa para a aplicação Monitoramento de Saturação de Conexões.

Esta também é uma aplicação que teve seus requisitos iniciais atendidos satisfatoriamente com a utilização do *framework*.

4.1.1.7. Monitoramento Assíncrono da Rede

O objetivo desta aplicação é monitorar as interfaces dos dispositivos de uma rede de forma *assíncrona*, isto é, o monitoramento deve ser efetuado não através da requisição periódica aos elementos gerenciados, mas sim, através da recepção de notificações (*traps*) que são enviados pelos dispositivos (agentes) para a estação de gerência. Nesta aplicação, tem-se interesse na ocorrência de traps SNMP dos tipos *linkup* e *linkdown*, que indicam que uma interface está ligada e desligada, respectivamente.

Configuração e instanciação dos componentes desta aplicação

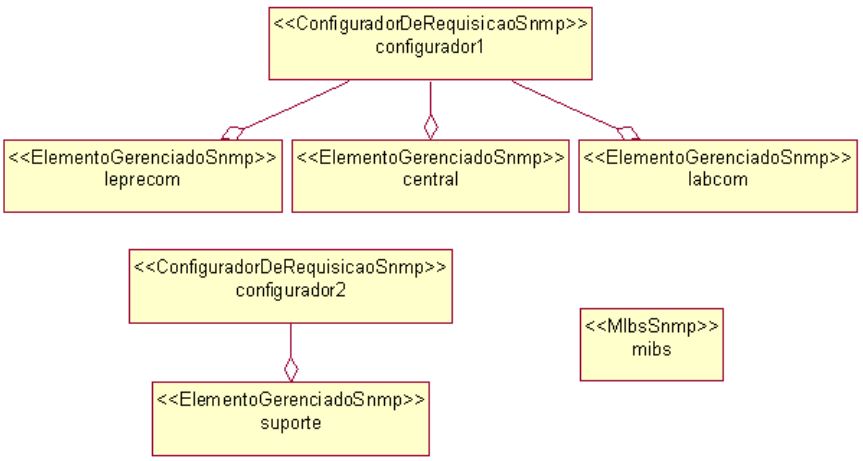


Figura 4.33. Componentes ConfiguradorDeRequisiçãoSnmp, ElementoGerenciadoSnmp e MibsSnmp para a aplicação Monitoramento Assíncrono da Rede.

1. Dois `ConfiguradorDeRequisicaoSnmp` (*configurador1* e *configurador2*), conforme mostrado na **Figura 4.33**, para configurar os parâmetros de requisição para o protocolo SNMP.

2. Quatro componentes `ElementoGerenciadoSnmp` (*leprecom*, *central*, *labcom* e *suporte*) representando os dispositivos gerenciados da rede (**Figura 4.33**).

3. Um componente `MibsSnmp` (**Figura 4.33**) no qual é informado qual arquivo MIB (ou quais) deve ser carregado.

4. Um componente para a geração de alarmes `GeradorDeAlarmesTrap` (**Figura 4.34**) para notificar ao administrador da ocorrência de traps na rede.

5. Quatro componentes `CorrelatorDeEventoFalhaSupressao`, que estão associados aos componentes GeradorDeEventoFalhaLinkDown com o objetivo de se limitar o número de eventos gerados (**Figura 4.34**).

6. Quatro geradores de eventos (`GeradorDeEventoFalhaTrapLinkDown`) para monitorar a ocorrência de traps SNMP do tipo *LinkDown* nas interfaces dos elementos gerenciados recebidos pelo `ReceptorDeTrapsSnmp` (**Figura 4.34**).

7. Um `ReceptorDeTrapsSnmp` que recebe os traps gerados pelos agentes SNMP e gera eventos para os componentes `GeradorDeEventoFalhaLinkDown`, no caso dessa aplicação (**Figura 4.34**).

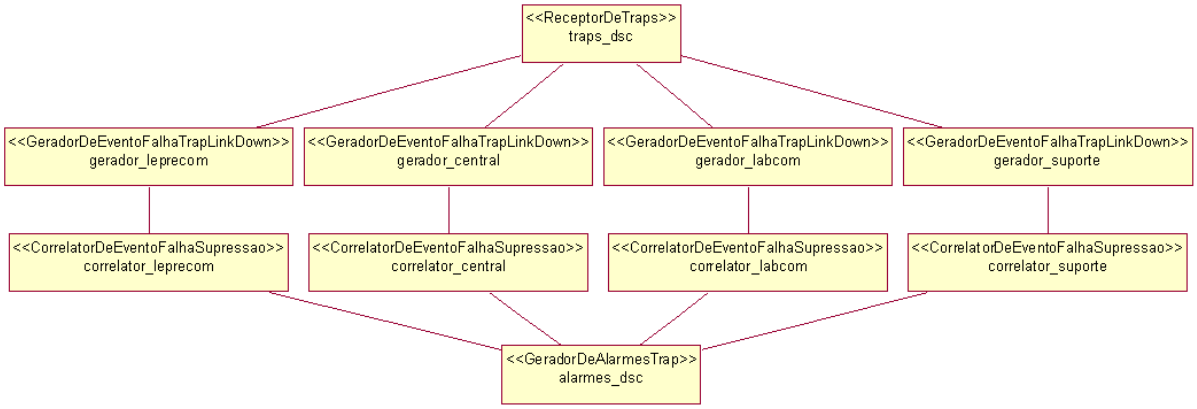


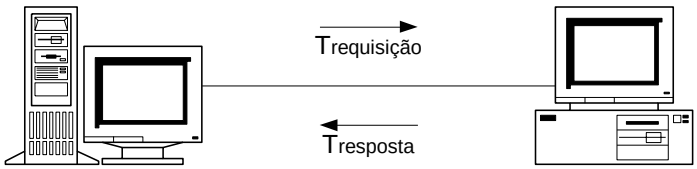
Figura 4.34. Componentes ReceptorDeTrapsSnmp, CorrelatorDeEventoFalhaSupressao, GeradorDeEventoFalhaTrapLinkDown e GeradorDeAlarmesTrap para a aplicação Monitoramento Assíncrono da Rede.

Com relação à construção desta aplicação, pode-se concluir que o *framework* atendeu satisfatoriamente aos requisitos originais da mesma.

4.1.2. Aplicações Mal Sucedidas

4.1.2.1. Tempo de Resposta

O objetivo desta aplicação é medir o tempo gasto entre o momento em que uma requisição é enviada a um equipamento e a obtenção da resposta a esta requisição (**Figura 4.35**). Além disso, é também desejável obter-se o tempo de resposta entre dois dispositivos quaisquer da rede, não sendo necessário que um deles seja a estação de gerência, isto é, o dispositivo onde executa a aplicação. Esta aplicação é importante no gerenciamento de faltas pois pode dar ao responsável pela rede uma idéia do quanto a mesma está congestionada, dos horários em que o volume de tráfego é maior (horário de pico) e dos "gargalos" da rede.



Tempo de Resposta = $T_{requisição} + T_{resposta}$

Figura 4.35. Tempo de resposta entre dois dispositivos de uma rede.

Para a construção desta aplicação, é necessário que o *framework* possua algum componente capaz de medir tempo de resposta. Entretanto, o *framework* não possui atualmente nenhum componente com esta característica.

Além disso, a medição do tempo de resposta entre dois dispositivos quaisquer (que não sejam a estação de gerência) também não é possível de ser efetuada devido ao fato de o *framework* ser baseado na **gerência de redes centralizada** ([Martin-Flatin & Znaty, 1997b], [Martin-Flatin *et al.*, 1999]), ou seja, toda aplicação construída com ele executa em uma única estação e, consequentemente, toda requisição é sempre efetuada entre esta estação e um dispositivo gerenciado. Assim sendo, o tempo de resposta entre dois dispositivos distintos daquele onde a aplicação está executando não pode ser calculado.

A solução, portanto, para que seja possível construir aplicações capazes de medir tempo de resposta baseadas no *framework* seria a inclusão de componentes com esta característica no conjunto de componentes do *framework*. Além disso, para a medição entre dois dispositivos quaisquer, o *framework* deveria possibilitar a geração de aplicações que pudessem ser distribuídas em mais de uma estação da rede, ou seja, o *framework* deveria se basear na **gerência de redes distribuída** ([Martin-Flatin & Znaty, 1997b], [Martin-Flatin *et al.*, 1999]).

4.1.2.2. Configuração de Alarmes na MIB RMON

A MIB RMON fornece um mecanismo pelo qual se pode selecionar uma variável MIB e indicar limiares para que sejam gerados alarmes quando os mesmos forem ultrapassados. Portanto, deseja-se construir uma aplicação através da qual seja possível informar os valores dos limiares e a variável desejada para que o *probe* RMON monitore-a e gere alarmes quando do cruzamento dos limiares.

Entretanto, esta aplicação não pode ser construída atualmente através da utilização do *framework*, pois o mesmo não possui nenhum componente com conhecimento sobre a semântica RMON. Além disso, o *framework* também não possui componentes com capacidade para efetuar operações do tipo SET (a arquitetura nunca considerou esta operação), sendo possível apenas efetuar operações do tipo GET.

4.1.2.3. Descobrimento e Estatísticas de Tráfego de Hosts na MIB RMON

A MIB RMON mantém uma tabela com a lista de todos os *hosts* descobertos através da análise dos pacotes que passam pelas interfaces além de um histórico com informações estatísticas de tráfego sobre cada *host*, como por exemplo, tráfego total enviado ou recebido por ele, tráfego *broadcast* e *multicast* gerado, entre outras. Portanto, esta tabela

varia dinamicamente, com novas entradas sendo adicionados a ela quando um novo *host* é descoberto e entradas existentes sendo excluídas quando, por exemplo, ocorre falta de recursos no equipamento.

Deseja-se então monitorar esta tabela informando ao responsável pela rede gerenciada, por exemplo, quais os hosts que estão gerando uma maior volume de tráfego, quais os horários de pico (aqueles em que há uma utilização considerável do canal de comunicação), entre outras. Dessa forma, o responsável pela rede pode ter uma noção de toda a comunicação que está ocorrendo na sua rede e do volume de tráfego que está sendo gerado e por quem.

Entretanto, esta é uma aplicação que não pode ser construída com o *framework*, tendo em vista que o mesmo apenas executa operações do tipo GET. Para que ele pudesse trabalhar com esta característica dinâmica desta tabela (ou de outra qualquer), seria necessário que ele efetuasse requisições do tipo GET-NEXT (considerando o SNMP, ou equivalente para outro protocolo).

Além disso, ele teria que ter a capacidade de instanciar componentes dinamicamente, tendo em vista que, para cada nova entrada, seria necessário instanciar diversos componentes, como por exemplo, unidades de informação de gerência, geradores de eventos, entre outros. Ele também deveria efetuar as ligações (*event binding*) entre estes novos componentes. Da mesma forma, quando uma entrada fosse excluída, ele teria que desfazer as ligações estabelecidas.

4.2. Desempenho do *Framework*

Para se efetuar a monitoração síncrona na rede, ou seja, aquela efetuada periodicamente, o fwGF provê o componente `Monitor`. Este componente é composto por um `ElementoGerenciado` e um `GrupoInfoGerencia` que contém as variáveis de gerência a serem enviadas em uma requisição. Dessa forma, a cada intervalo de tempo dado pela propriedade `periodo` do `Monitor`, este envia uma mensagem para o `ElementoGerenciado` para que seja efetuada uma requisição do grupo de informações à respectiva entidade gerenciada. Quando uma requisição é concluída, o `Monitor` então gera eventos que são enviados para os consumidores nele cadastrados. Pode-se dizer, então, que o `Monitor` é o iniciador de eventos do *framework*. Tudo no *framework* ocorre em decorrência de eventos gerados pelos componentes deste tipo. Cada `Monitor` instanciado tem uma *thread* em separado (mais detalhes sobre thread podem ser encontrados em [Eckel, 2000].

Quando o `Monitor` envia eventos para os consumidores nele cadastrados, os componentes chamados executarão na thread do `Monitor`. Se estes componentes, por sua vez, chamarem outros interessados (até o envio do e-mail, conforme a **Figura 2.6**, por exemplo) tudo continua ocorrendo ainda na *thread* do `Monitor`. Portanto, todo este processamento pode ser demorado. Para se evitar que um processamento demore demais, uma alternativa de implementação é fazer com que cada componente interessado abra sua própria thread para executar alguma tarefa, liberando a thread original para que continue sem demora (**Figura 4.36**). No caso do fwGF, se está utilizando a primeira situação, ou seja, todo o processamento é feito na thread do `Monitor`.

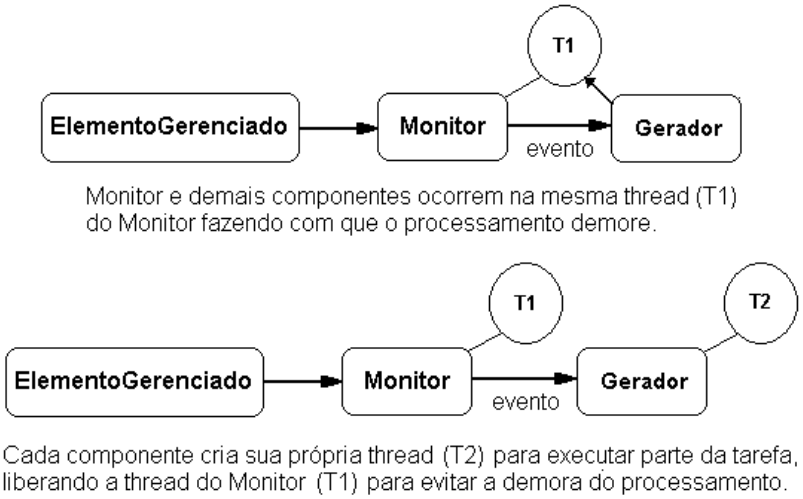


Figura 4.36. Processamento efetuado em uma ou mais threads

Contudo, para que o intervalo de tempo entre as monitorações seja sempre igual, é necessário considerar o tempo gasto com as atividades que o `Monitor` tem que fazer entre as mesmas, como por exemplo, o envio de eventos para os consumidores cadastrados. Dessa forma, este tempo gasto sempre é subtraído do intervalo de tempo de monitoração, para que a monitoração seja feita em intervalos regulares (**Figura 4.37**).

Entretanto, qual é a quantidade de componentes que podem ser instanciados em uma aplicação de forma que esta não prejudique o processo de monitoração? Em outras palavras, como se pode medir a escalabilidade do fwGF?

Considerando que cada `Monitor` roda em uma *thread* em separado e que para cada dispositivo que se deseja gerenciar deve-se associar um Monitor a ele, então a monitoração de um conjunto de dispositivos gerenciados é efetuada de forma paralela e independente.

Sendo assim, o que provoca impacto na monitoração de uma rede é a quantidade de componentes instanciados em uma aplicação, o que faz com que recursos da estação de gerência, como por exemplo memória RAM e CPU, tenham que ser compartilhados pelos componentes. Consequentemente, alguns destes componentes terão que esperar (tempo de latência) até que todos os recursos necessários sejam liberados para então poderem ser executados.

Para poder então se ter uma idéia da escalabilidade do fwGF, aplicações de teste foram construídas nas quais o tempo de monitoração de todos os componentes do tipo `Monitor` foi estabelecido em cinco segundos.

Utilizando-se um computador com processador Pentium II de 300 MHz com 128 Mbytes de memória RAM e rodando o sistema operacional Linux e JDK 1.2.2 (*Java Development Kit*), a quantidade de componentes `Monitor` instanciada em uma aplicação de forma a não permitir que a monitoração extrapole o intervalo de tempo estabelecido ficou em torno de cem unidades. Considerando-se que:

1. o número total de componentes instanciado na aplicação é ainda maior, pois para cada `Monitor`, no mínimo, foram necessários instanciar mais quarenta outros, entre eles componentes do tipo `ElementoGerenciado`, `InfoGerencia`, `GrupoInfoGerencia`, `GeradorDeEventoFalha`, etc.;

2. e que o intervalo de tempo estabelecido para monitoração nestas aplicações de teste é muito pequeno (cinco segundos),

podemos admitir que a quantidade de equipamentos que possam ser monitorados por uma aplicação construída com o fwGF possa atingir bem mais do que a centena aqui conseguida, tendo em vista que o intervalo de monitoração de aplicações, na prática, é muito maior (geralmente, quinze minutos).

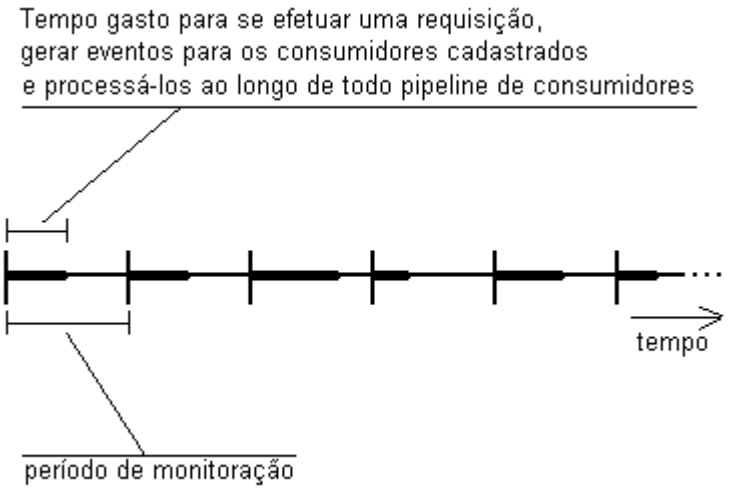


Figura 4.37. Monitoração de uma entidade gerenciada ao longo do tempo

4.3. Conclusões

Neste capítulo foram selecionadas algumas aplicações de gerência de faltas em redes de computadores para ser construídas através da utilização do *framework*. Dentre este conjunto de aplicações, como pode-se observar, o *framework* obteve sucesso na construção de algumas enquanto outras foram mal-sucedidas.

Com relação às aplicações que não puderam ser construídas com o *framework*, pode-se dizer que a principal causa de seu insucesso é a falta de um conjunto maior de componentes. Mesmo o *framework* não apresentando conhecimentos sobre a semântica de RMON nem estando apto a realizar operações do tipo SET ou GET-NEXT, conforme visto na construção de algumas aplicações anteriores, ainda assim, este é um problema que pode ser solucionado com a adição de novos componentes que suportem estas características.

Entretanto, esta limitação que o *framework* apresenta é normal, tendo em vista que o mesmo encontra-se no início do processo de sua evolução [Roberts & Johnson, 1996]. A partir do momento em que a biblioteca de componentes do *framework* for enriquecida de novos componentes, tais limitações tendem a desaparecer.

Por fim, um aspecto importante a ser notado aqui é que o *framework* não tem apresentado problemas em sua arquitetura, o que colabora ainda mais para o fato de que suas limitações se devem apenas à falta de um conjunto maior de componentes.

5. Evolução do *Framework* fwGF

Neste capítulo serão apresentados os detalhes referentes à etapa de **evolução do *framework* fwGF**. Tal evolução é baseada no processo descrito em [Roberts & Johnson, 1996] e tem por objetivo acrescentar ao fwGF as características necessárias para que o mesmo possa ser utilizado na construção de aplicações que dêem suporte ao paradigma de **gerência de redes distribuída** [Martin-Flatin *et al.*, 1999].

Nas seções a seguir serão apresentados todos os passos envolvidos na evolução do fwGF. Na **seção 5.1** será apresentada uma breve descrição sobre o processo de evolução de *frameworks* mostrado em [Roberts & Johnson, 1996]; Na **seção 5.2**, descrever-se-á a gerência de redes centralizada e distribuída, apresentando-se suas características fundamentais e os motivos pelos quais a gerência centralizada não atende aos requisitos impostos pelas redes atuais; em seguida, na **seção 5.3**, será apresentada a evolução do fwGF, ou seja, será mostrada a especificação dos componentes do *framework* para dar suporte à gerência distribuída. Dentro desta seção, serão mostrados os requisitos estabelecidos para a evolução do fwGF, os novos componentes incorporados ao fwGF, a instanciação de aplicações, bem como um exemplo de uma aplicação distribuída.

5.1. O Processo de Evolução de *Frameworks*

Desenvolver um *framework* é uma tarefa dispendiosa, pois ela requer que os seus desenvolvedores tenham um bom conhecimento sobre o domínio do problema ao qual o *framework* se aplica, de forma que o produto final não apenas seja simples de ser utilizado e entendido como também forneça as características necessárias para que se possa "rapidamente" construir aplicações com o mesmo [Roberts & Johnson, 1996]. De modo geral, pode-se dizer que as etapas que envolvem o desenvolvimento de um *framework* são [Rogers, 1997]:

- a coleta de requisitos de aplicações que constituem o *domínio do problema* para o qual o *framework* está sendo desenvolvido;
- a especificação do *framework* através da generalização destes requisitos;
- projeto e implementação do *framework*.

Entretanto, um *framework* não é um produto estático. Pelo contrário, ele possui um ciclo de vida próprio ao longo do qual o mesmo evolui, passando por diversas fases que contribuem para que o mesmo possa cada vez mais atender aos requisitos do domínio do problema no qual está inserido.

O ciclo de vida do desenvolvimento de um *framework* é representado por um conjunto de etapas que ocorrem no decorrer do tempo conforme mostrado na **Figura 5.1**. Esta figura foi adaptada de [Roberts & Johnson, 1996] e os nomes das etapas nela constantes foram traduzidos.

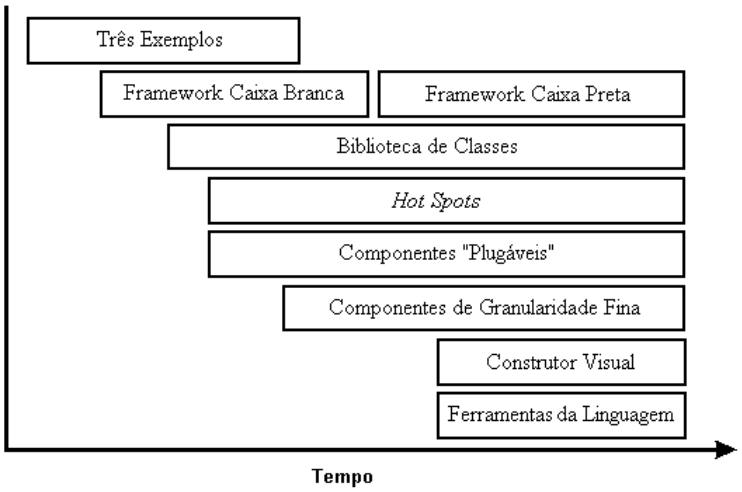


Figura 5.1. Evolução de *Frameworks*

As atividades desenvolvidas nestas etapas, como pode ser observado nesta figura, podem ocorrer em paralelo evoluindo conjuntamente. Além disso elas podem ser aplicadas mais de uma vez ao longo do desenvolvimento do *framework*.

De forma suscinta, a seguir tem-se uma descrição de cada etapa mostrada na **Figura 5.1**. Caso o leitor deseje conhecer mais detalhes sobre o processo de evolução aqui utilizado, é fortemente recomendada a leitura de [Roberts & Johnson, 1996].

- *Três Exemplos*: construir três aplicações (ou mais) pertencentes a um mesmo domínio de problema, de foram que a partir delas se possa fatorar o que há de comum e assim prover as abstrações adequadas que irão compor o *framework*;
- *Framework Caixa Branca*: a partir das abstrações determinadas, deve-se construir as classes que constituirão o *framework*, não se preocupando com a semântica da herança, mas somente na habilidade de reutilizar código;

- *Biblioteca de Classes*: a partir das classes abstratas presentes no *framework*, prover classes concretas para serem usadas nas aplicações, constituindo assim uma biblioteca de classes;
- *Hot Spots*: nesta etapa, deve-se observar pontos onde há o mesmo código sendo reescrito, com o objetivo de eliminá-lo. Tal código deve ser encapsulado em objetos ao invés de métodos, pois eles são mais reutilizáveis do que estes últimos;
- *Componentes "Plugáveis"*: o objetivo aqui é evitar que se construam classes triviais toda vez que se deseja utilizar o *framework*. Portanto, a solução aqui apresentada é construir subclasses que podem ser parametrizadas, pois fornecer este tipo de classes prover reuso sem esforço de programação;
- *Componentes de Granularidade Fina*: busca-se aumentar a granularidade da biblioteca de componentes do *framework*, "quebrando" os componentes em objetos menores, com o intuito de torná-los mais reutilizáveis;
- *Framework Caixa Preta*: neste ponto, como se está passando pelas etapas *Componentes "Plugáveis"*, *Hot Spots* e *Componentes de Granularidade Fina*, busca-se então tornar o *framework* o mais "componentizado" possível, ou seja, na construção de novos componentes deve-se fazer reutilização daqueles já existentes plugando-os sem se preocupar com suas tarefas individuais, ao contrário do que ocorre com um *framework* white-box que requer um entendimento de como as classes funcionam para que se possa construir as subclasses corretas;
- *Construtor Visual*: construir uma ferramenta gráfica que auxilie na construção de aplicações de forma visual, na qual os componentes que constituirão a aplicação possam ser conectados entre si e todo o código necessário deve ser gerado automaticamente;
- *Ferramentas da Linguagem*: criar ferramentas especializadas para facilitar a depuração e inspeção das composições (geralmente complexas) criadas com a construtor visual.

Como estas etapas ocorrem ao longo do tempo, há momentos do desenvolvimento de um *framework* em que nem todas elas terão sido realizadas. Este é o caso do fwGF, cujo desenvolvimento atual ainda não passou pelas etapas de *Construtor Visual* e *Ferramentas da Linguagem*, por exemplo. Para o fwGF, pode-se dizer que o mesmo encontra-se como

mostrado na **Figura 5.2**, onde a área cinza representa a evolução do *framework* indicando as atividades que já foram realizadas. No momento, o fwGF apresenta-se parte como um *framework Caixa Preta*, parte como um *Caixa Branca*, tendo em vista que a construção de novos componentes ainda requer, em alguns casos, o conhecimento da semântica de herança dos componentes já existentes.

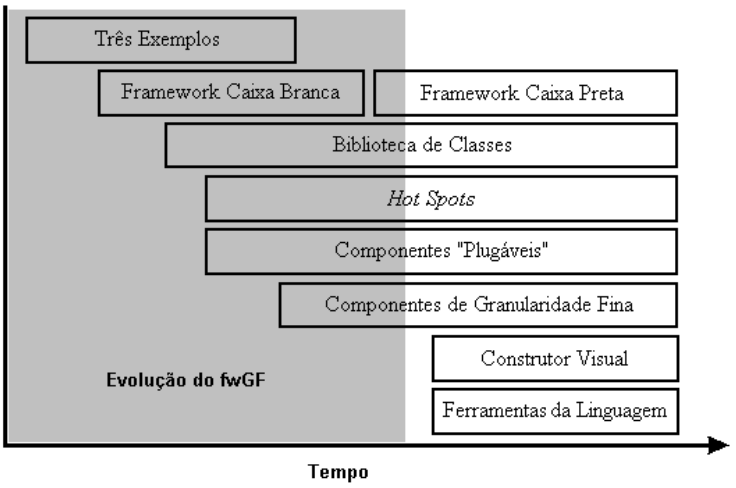


Figura 5.2. Evolução de fwGF

No trabalho descrito neste capítulo, a evolução do fwGF se dará no provimento de novos componentes que dêem suporte à gerência de falhas distribuída. A descrição deste tipo de gerência, bem como dos componentes construídos é feita ao longo das seções que se seguem.

5.2. Gerência de Redes Centralizada e Distribuída

Para se efetuar um gerenciamento de uma rede de computadores, quatro são os elementos fundamentais que devem se fazer presentes:

- Estação de gerência;
- Entidades gerenciadas;
- Informação de gerência;
- Protocolo de gerência.

Durante vários anos, o gerenciamento das redes foi baseado em um modelo contendo apenas uma única estação de gerência responsável por monitorar a rede, coletar as informações necessárias das entidades gerenciadas através de um protocolo de gerência, analisá-las e gerar informações para os responsáveis pela administração de tais redes

[Goldszmidt & Yemini, 1995a]. Este modelo é conhecido como **gerência de redes centralizada** (**Figura 5.3)**.

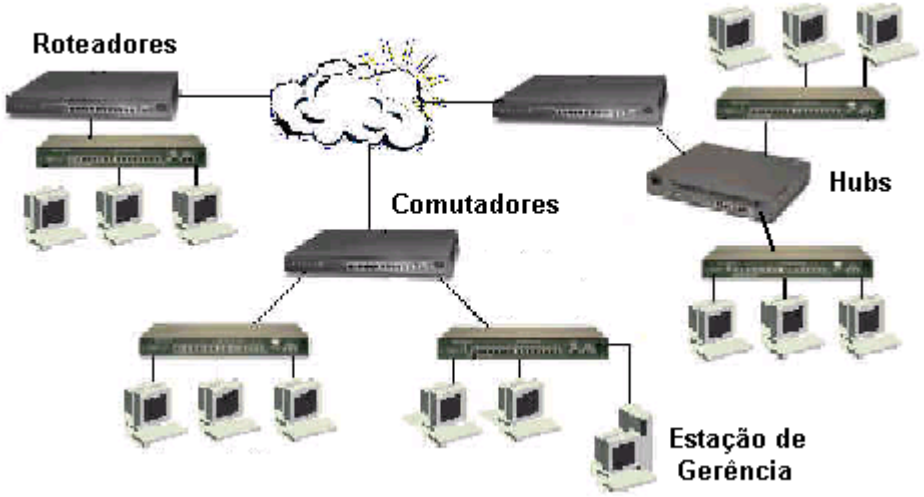


Figura 5.3. Gerência de Redes Centralizada

Este modelo de gerência era eficiente à época, onde os recursos de hardware eram limitados comparados com os atuais. Porém, com o passar dos anos, as redes foram crescendo enormemente em tamanho. Muitas delas passaram de dezenas para milhares de equipamentos. Hoje, várias empresas de grande porte possuem mais de 100.000 equipamentos nas suas redes corporativas. Portanto, acompanhando este crescimento, aumentou também a necessidade de se ter um gerenciamento mais eficiente. Foi justamente este aumento de tamanho das redes que gerou a primeira causa da ineficiência do modelo de gerência centralizada: **escalabilidade**. Como a inteligência do gerenciamento está centralizada em um único ponto (estação de gerência), para redes muito grandes, a quantidade de informações a ser recebida das entidades gerenciadas e analisada pela estação faz com que ela fique impossibilitada de recebê-las ou processá-las em tempo hábil.

Um outro problema também encontrado no modelo de gerência centralizada é a pouca **robustez** que ele oferece. Se, por algum motivo, ocorrer alguma falha na estação de gerência, todo o gerenciamento da rede fica comprometido, pois perde-se o único ponto capaz de coletar as informações das entidades gerenciadas.

Portanto, a **gerência de redes distribuída**[10] surge como uma forma de superar as limitações acima citadas inerentes à gerência centralizada. Como a inteligência, que antes era

[10] A gerência de redes distribuída pode ser classificada em vários tipos, de acordo com os critérios adotados em [Martin-Flatin *et al.*, 1999], cuja leitura é aqui recomendada.

concentrada em um único ponto, agora é dividida entre várias estações de gerência, ela provê escalabilidade para adaptar-se às expansões da rede, tendo em vista que as tarefas do gerenciamento podem ser dispersadas continuamente pela rede quando esta aumenta de tamanho. Além disso, com a distribuição das tarefas, reduz-se a complexidade das aplicações de gerência.

Devido ao aumento na capacidade computacional dos equipamentos que constituem uma rede (roteadores, comutadores, etc.), pode-se também colocar tarefas de gerência nestes inclusive, para que algumas ações corretivas sejam tomadas na presença de certos problemas. Tais problemas podem ser solucionados sem a intervenção da estação de gerência, sendo esta apenas notificada da ocorrência do mesmo e da ação tomada para sua recuperação (**Figura 5.4**).

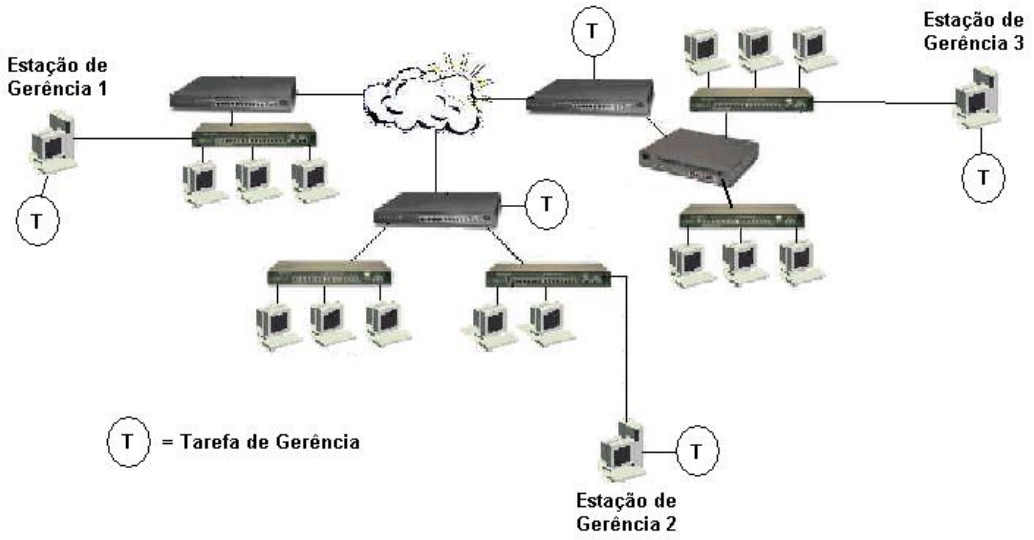


Figura 5.4. Gerência de Redes Distribuída

Este modelo também é muito mais robusto que o centralizado, pois permite que se tenham operações redundantes, ou seja, como uma estação gerencia uma área da rede, pode-se ter sobreposição de áreas, o que implica que uma entidade gerenciada pode ser monitorada por mais de uma estação. Consequentemente, na ocorrência de falha em uma estação, aquela entidade não deixa de ser monitorada (**Figura 5.5**). Nesta figura, cada estação cuida do gerenciamento da área na qual está inserida. Além disso, na presença de problemas, estas estações podem tomar as ações corretivas necessárias de maneira muito mais eficiente e rápida do que no modelo centralizado, o que contribui ainda para a redução da quantidade de informações que trafegam na rede.

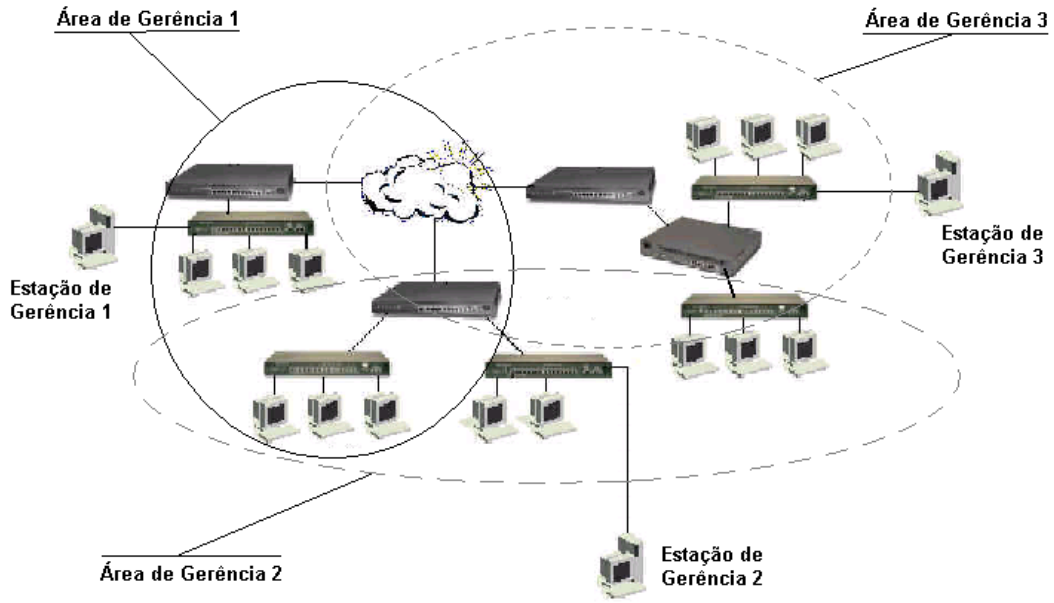


Figura 5.5. Sobreposição de áreas de gerenciamento

Para prover suporte à gerência de redes distribuída, nos últimos anos, várias abordagens têm sido especificadas e alguns produtos têm sido desenvolvidos. Em [Goldszmidt & Yemini, 1995a] são apresentadas métricas que podem ser utilizadas para se determinar o grau de descentralização apropriada para uma dada aplicação de gerência. Em [Martin-Flatin & Znaty, 1997a] e [Martin-Flatin & Znaty, 1997b,] é feita uma revisão de todos os paradigmas de gerência existentes e uma forma simples de classificá-los é proposta. Em [Martin-Flatin *et al*, 1999], os autores elaboram uma classificação destes paradigmas baseados em quatro critérios: granularidade de delegação, riqueza de semântica do modelo de informação, o grau de especificação de uma tarefa e o grau de automação do gerenciamento. [Goldszmidt & Yemini, 1995b], por sua vez, apresenta a Gerência Distribuída por Delegação (*Distributed Management by Delegation*) que usa o conceito de agentes móveis[11] para gerenciar dispositivos.

[11] Um agente móvel pode ser descrito como um pequeno programa que tem a habilidade de migrar entre hosts através de uma rede durante sua execução.

5.3. Evolução do fwGF

5.3.1. Requisitos

Com base na evolução de *frameworks* apresentada em [Roberts & Johnson, 1996], a evolução do fwGF detalhada neste capítulo se dará basicamente na ampliação de sua *Biblioteca de Classes*, com a inclusão de novos componentes para que o *framework* possa ser utilizado na construção de aplicações de gerência de falhas distribuída.

Para se efetuar esta evolução, alguns requisitos aqui considerados importantes devem ser atendidos.

R1. a construção de aplicações de gerência que funcionem de forma centralizada ou distribuída na rede deve ser o mais transparente possível. Em outras palavras, o desenvolvedor pode construir uma aplicação distribuída utilizando os mesmos conhecimentos já aplicados na gerência centralizada;

R2. o requisito inicial de fornecer abstrações de alto nível para o desenvolvedor de aplicações especificado em [Freire, 2000] deve ser mantido. Portanto, a construção de aplicações de gerência distribuída não deve implicar na aquisição de nenhum conhecimento relativo a detalhes de baixo nível por parte do desenvolvedor.

R3. para manter a portabilidade das aplicações, a implementação das características relativas à gerência distribuída deve utilizar uma solução puramente Java;

R4. os componentes que constituem uma aplicação podem ser instanciados e distribuídos pelos diversos pontos da rede (desde que possuam uma JVM - Java Virtual Machine [Arnold & Gosling, 2000]). Estes últimos passarão a exercer o papel de estações de gerência, devendo o *framework* prover os mecanismos necessários para localizar e interconectar componentes instanciados em estações distintas;

R5. o *framework* deve prover a sincronização durante a instanciação dos componentes distribuídos de uma aplicação. Considere, por exemplo, que dois componentes X e Y serão instanciados em pontos distintos da rede e que X deve se conectar a Y (X pode ser um *listener* de Y). Porém, se X for instanciado antes de Y, ele não vai poder cadastrar-se neste. Portanto,

mecanismos de sincronização devem ser providos como solução para que tal situação não venha a ocorrer.

Na seção a seguir é então apresentada uma solução para que o fwGF possa atender aos requisitos acima estabelecidos e prover o suporte à gerência distribuída.

5.3.2. Especificação da Solução

Inicialmente, como forma de atender ao requisito **R3** estabelecido acima, a tecnologia puramente Java escolhida para fornecer as características necessárias à construção de aplicações distribuídas foi RMI (*Remote Method Invocation*). Esta tecnologia provê os recursos para a comunicação remota entre aplicações Java, ou seja, permite que um objeto rodando em uma JVM possa invocar métodos de um objeto rodando em outra JVM [Sun, 2001].

Uma aplicação distribuída RMI é composta de duas partes: uma servidora e uma cliente. O servidor instancia objetos remotos, disponibiliza referências a estes e espera pela invocação dos seus métodos pelos clientes. Estes clientes, por sua vez, obtêm a referência a estes objetos remotos e então podem efetuar chamadas a seus métodos.

Para um objeto tornar-se remoto, é preciso que o mesmo implemente uma interface remota, a qual deve possuir duas características fundamentais: implementar a interface `java.rmi.Remote` e cada método da interface deve declarar `java.rmi.RemoteException` em sua cláusula `throws`, em adição às exceções já declaradas [Eckel, 2000].

Um objeto remoto deve ainda ou estender a classe `java.rmi.server.UnicastRemoteObject` ou ser passado como argumento para um dos métodos estáticos `exportObject` dessa classe, tendo em vista que esta é quem provê os mecanismos necessários para a comunicação entre objetos localizados em JVMs distintas na arquitetura RMI.

Basicamente, uma aplicação RMI distribuída precisa [Sun, 2001]:

- *Localizar objetos remotos*: objetos remotos devem ser registrados no serviço de nomes RMI, rmiregistry, para que os mesmos possam ser então referenciados pelos clientes;
- *Efetuar comunicação entre objetos remotos*: todos os aspectos referentes à comunicação entre os objetos é tratada para o programador pelo RMI, fazendo

com que chamadas a métodos remotos sejam iguais a de métodos de objetos locais;

- *Carga dinâmica de código*: esta é a principal vantagem da arquitetura RMI. Como ela permite que sejam passados objetos como argumentos para métodos remotos, ele provê mecanismos para fazer o download automático do código do objeto passado, bem como dos seus dados.

RMI trata objetos remotos e não-remotos de forma distinta quando os mesmo são passados de uma JVM para outra. No caso de um objeto não-remoto, RMI passa uma cópia do mesmo para a outra JVM, o que implica que qualquer modificação efetuada neste objeto na JVM de destino não afetará o seu estado na JVM de origem.

Por outro lado, quando um objeto remoto é passado para uma outra JVM, na realidade o que é passado é uma referência ao mesmo, ou seja, um *stub*. Este stub é carregado dinamicamente da JVM de origem para a do cliente. Como o stub é uma referência ao objeto remoto, quando um método é chamado, ele então estabelece uma comunicação com o objeto remoto para a execução do método (o *stub*, bem como seu correspondente no lado servidor, o *skeleton*, são gerado pelo compilador *rmic*, a partir do arquivo .class do objeto remoto).

A **Figura 5.6** apresenta a arquitetura de uma aplicação distribuída RMI. Nesta figura, um objeto remoto é registrado no serviço de nomes RMI. Quando um cliente deseja chamar um método remoto, ele localiza o objeto através do serviço de nomes, fornecendo o nome com o qual o objeto remoto foi registrado. A partir daí, é feito o download automático do stub. Este, juntamente com o *skeleton*, provê a infra-estrutura para a comunicação entre as JVMs. A Figura também mostra que o sistema RMI usa um servidor WEB já existente para poder fazer o download automático do código dos objetos do servidor para o cliente e vice-versa, quando necessário.

Portanto, com base na arquitetura de uma aplicação RMI mostrada na **Figura 1.6**, para que o fwGF possa dar suporte à gerência de redes distribuída é necessário que os seus componentes sejam transformados em objetos remotos. Para isso, é necessário alterar as interfaces dos mesmo, fazendo com que estendam a interface `java.rmi.Remote`. Além disso, é preciso também modificar a assinatura dos métodos de cada uma dessas interfaces, adicionando a exceção `java.rmi.RemoteException` em suas cláusulas `throws`.

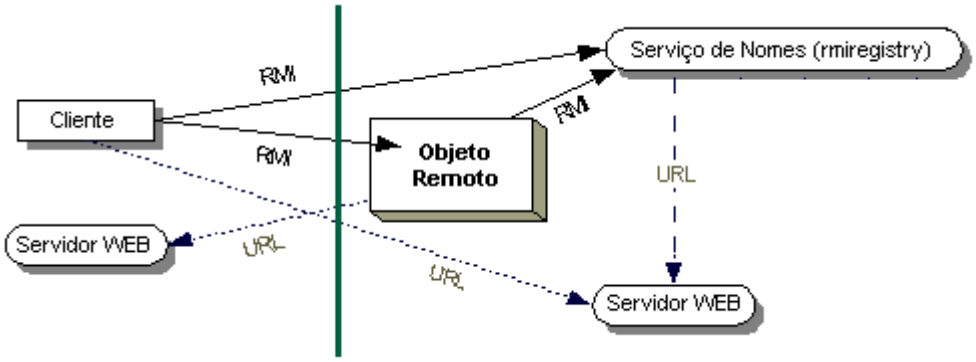


Figura 5.6. Arquitetura de uma aplicação RMI

Portanto, ao invés de se criar novos componentes para que seja possível construir aplicações distribuídas com o fwGF, apenas foi modificada a estrutura daqueles já existentes, reaproveitando o código que já tinha sido anteriormente implementado, o que implica que tem-se classes únicas usadas tanto em aplicações centralizadas como distribuídas. Dessa forma, o requisito **R2** é então satisfeito tendo em vista que os componentes existentes já forneciam abstrações de alto nível e as modificações agora acrescentadas não afetam esta característica.

A **Tabela 5.1** apresenta as interfaces do *framework* que foram transformadas em interfaces remotas, ou seja, que passaram a estender `java.rmi.Remote`.

Os componentes `MonitorEvent` e `EventoFalhaEvent` não foram transformados em remotos devido ao fato de os mesmos serem utilizados apenas para a comunicação entre componentes. Além disso, as propriedades que eventualmente possam ser alteradas remotamente já implementam uma interface remota: `MonitorInterface` no caso do `MonitorEvent` e `GeradordeEventoFalha` em `EventoFalhaEvent`.

Analogamente a RMI, no caso específico do fwGF há componentes que exercem o papel de servidor, de cliente ou ainda de ambos. Portanto, como em RMI os objetos remotos devem ser registrados (*rmiregistry*), o fwGF também deve prover algum meio para registrar os seus componentes remotos. Além disso, também deve ser provida uma forma para se informar a um componente instanciado que exerça o papel de cliente qual (ou quais) o componente que será o seu servidor. É óbvio que para aqueles componentes que possam funcionar como cliente e servidor, o fwGF deve prover ambos os mecanismos citados acima.

Como já dito anteriormente, para um objeto remoto em RMI poder receber chamadas de clientes, é necessário que o mesmo estenda a classe `UnicastRemotObject` ou então que seja passado como argumento do método `exportObject` dessa classe. No fwGF, optou-se por não fazer com que os componentes remotos estendessem

`UnicastRemoteObject`. Cada componente remoto então provê o método `exportObject` que faz uma chamada ao método estático `UnicastRemoteObject.exportObject`, passando a si próprio como argumento. Além disso, ele também implementa o método `unexportObject(boolean force)` o qual é responsável por remover o componente do ambiente RMI, fazendo com que o mesmo não esteja mais disponível para atender chamadas remotas a seus métodos.

Interfaces Remotas	Descrição
`ConfiguradorDeRequisicao`	Em RMI, como um objeto remoto é passado por referência entre JVMs, estas interfaces foram transformadas em remotas para que as alterações ocorridas nas propriedades dos componentes que as implementam possam ter efeito
`ElementoGerenciado`	
`GrupoInfoGerencia`	
`InfoGerencia`	
`MonitorInterface`	Os componentes que implementam esta interface exercem o papel de servidor
`ReceptorDeTraps`	Os componentes que implementam estas interfaces podem exercer o papel de servidor e de cliente
`GeradorDeEventoFalha`	
`CorrelatorDeEventoFalha`	
`GeradorDeAlarmes`	
`MonitorListener`	Estas interfaces foram transformadas em remotas para justamente permitir que um componente cliente possa se cadastrar em um servidor localizado em outra JVM
`EventoFalhaListener`	

Tabela 5.1. Interfaces remotas do fwGF

Da mesma forma como os objetos remotos precisam ter suas referências registradas no serviço de nomes RMI, o fwGF também provê um meio para tal. Sendo assim, cada componente implementa os métodos `register(String name)` e `unregister(String name)`. Estes são utilizados para registrar e remover a referência registrada com o nome dado por *name* no serviço de nomes RMI, respectivamente. Portanto, com o uso de RMI e com as adaptações descritas acima para transformar os componentes do fwGF em remotos, o requisito **R4** da **seção 5.3.1** é atendido.

Para que um cliente possa obter uma referência a um servidor, ele deve localizá-lo pelo respectivo nome utilizado para registrá-lo no serviço de nomes. Portanto, todo componente que exerce o papel de cliente implementa o método `lookup(String name)`, o qual obtém uma referência ao componente servidor dado pelo parâmetro name. Este método deve também fornecer os mecanismos necessários para sincronização conforme especificado

na **seção 5.3.1**. Assim sendo, se um cliente tentar obter uma referência a um servidor remoto e caso a mesma ainda não tenha sido registrada, ele deve aguardar um intervalo de tempo (*timeout*). Ao final deste intervalo, uma nova tentativa de se obter a referência é então efetuada. O número de tentativas bem como o valor do timeout podem ser especificados para cada componente de acordo com a vontade do usuário do *framework*. Portanto, o requisito **R5** estabelecido na **seção 5.3.1** é, desta forma, atendido.

Para atender ao requisito **R1**, a construção de aplicações distribuídas também faz uso da linguagem BML [Weerawarana & Duftler, 1999], cuja forma de se construir este tipo de aplicações é a mesma que a utilizada para a construção de aplicações centralizadas, conforme será visto na seção a seguir.

5.3.3. Instanciação de Aplicações

Para se instanciar os componentes das aplicações de gerência distribuída, a linguagem BML pode ser utilizada da mesma forma como se vinha utilizando-a quando da instanciação de aplicações centralizadas (**seção 3.7**) exceto pelos detalhes mostrados a seguir.

Primeiramente, o desenvolvedor de aplicações distribuídas deve construir, para cada estação de gerência onde a aplicação rodará, um script BML contendo os respectivos componentes a serem instanciados naquela estação. O desenvolvedor deve ainda, antes da execução do script BML, executar o serviço de nomes do RMI, através do aplicativo rmiregistry, para que seja possível que os componentes servidores possam ser registrados e referenciados pelos clientes.

Nos scripts BML, para todos os componentes remotos, deve ser executado o método exportObject de cada um deles. Para aqueles componentes que terão o papel de servidor, o método register(String name) deve ser executado, cujo parâmetro é o nome a ser registrado no serviço de nomes RMI. Para os componentes que terão o papel de clientes (*listeners*), ou seja, aqueles que são *listeners* devem executar o método `lookup(String name)`, o qual já deve implementar os mecanismos necessários para localizar o servidor e cadastrar-se neste. Obviamente, como este método cadastra o cliente em apenas um servidor, se o componente precisar se cadastrar em mais de um servidor, ele deverá ser chamado para cada um deles. Vale salientar que se um componente exerce os papéis de servidor e cliente ao mesmo tempo (por exemplo, um `GeradorDeEventoFalha` que pode ser cliente de um `Monitor` e servidor de um `CorrelatorDeEventoFalha`), então, os métodos `register` e `lookup` deverão ser executados passando-se os respectivos argumentos.

A **Figura 5.7** mostra um trecho de um script BML utilizado para se instanciar um `GeradorDeEventoFalhaComHisterese` e um `Monitor` em uma aplicação centralizada. Observe que para se cadastrar o gerador como *listener* do monitor, deve-se executar uma chamada ao método `addMonitorListener` deste último, passando-se o gerador como argumento.

```
...
<bean class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEquipamento" id="status_leprecom">
	<property name="nome" value="Status de leprecom"/>
	<property name="nomeIg">
		<property target="sysUpTime_leprecom" name="nome"/>
	</property>
	<property name="responsavel" value="Pedro Sérgio Nicolletti"/>
	<property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
	<property name="descricao" value="O comutador do leprecom está down."/>
	<property name="descricaoEventoRearm" value="O comutador do leprecom está up."/>
	<property name="prioridade">
		<field name="PRIORIDADE_ALTA"/>
	</property>
</bean>

...

<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_leprecom">
	<property name="periodo" value="60"/>
	<property name="eg">
		<bean source="leprecom"/>
	</property>
	<property name="gig">
		<bean source="gig_leprecom"/>
	</property>
	<call-method name="addMonitorListener">
		<bean source="status_leprecom"/>
	</call-method>
	<call-method name="start"/>
</bean>
...
```

Figura 5.7. Trecho de um *script* BML para uma aplicação centralizada

A **Figura 5.8**, por sua vez, mostra um trecho de um outro *script* BML onde é instanciado um `Monitor` remoto. Observe que são executados os métodos `exportObject` e `register`, cujo argumento, neste último, é justamente o nome a ser utilizado para se registrar este componente no serviço de nomes RMI. A **Figura 5.9**, apresenta um trecho BML que é executado em uma JVM diferente daquela onde roda o *script* da **Figura 5.8**. Neste

script é instanciado um `GeradorDeEventoFalhaComHisterese`. Observe que são executados os métodos `exportObject` e `lookup`, cujo argumento é o nome do servidor ao qual este componente deve-se cadastrar.

```
...
<bean class="br.ufpb.dsc.grc.fwGF.Monitor" id="monitor_leprecom">
        <property name="periodo" value="60"/>
        <property name="eg">
                <bean source="leprecom"/>
        </property>
        <property name="gig">
                <bean source="gig_leprecom"/>
        </property>
        <call-method name="exportObject"/>
        <call-method name="register">
                <string>MONITOR_LEPRECOM</string>
        </call-method>
        <call-method name="start"/>
</bean>
...
```

Figura 5.8. Trecho de um *script* BML para uma aplicação distribuída: instanciação de um Monitor

```
...
<bean class="br.ufpb.dsc.grc.fwGF.GeradorDeEventoFalhaStatusEquipamento" id="status_leprecom">
        <property name="nome" value="Status de leprecom"/>
        <property name="nomeIg">
                <property target="sysUpTime_leprecom" name="nome"/>
        </property>
        <property name="responsavel" value="Pedro Sérgio Nicolletti"/>
        <property name="contatoResponsavel" value="peter@dsc.ufpb.br"/>
        <property name="descricao" value="O comutador do leprecom está down."/>
        <property name="descricaoEventoRearm" value="O comutador do leprecom está up."/>
        <property name="prioridade">
                <field name="PRIORIDADE_ALTA"/>
        </property>
        <call-method name="exportObject"/>
        <call-method name="lookup">
                <string>MONITOR_LEPRECOM</string>
        </call-method>
</bean>
...
```

Figura 5.9. Trecho de um *script* BML para uma aplicação distribuída: instanciação de um GeradorDeEventoFalhaComHisterese

Então, resumindo, quando se tem uma aplicação centralizada, para os componentes que são produtores de eventos deve-se chamar o método correspondente para se cadastrar os consumidores interessados em receber eventos. Por outro lado, em uma aplicação

distribuída, deve-se chamar os métodos `register` para aqueles componentes que são servidores (produtores de eventos) e `lookup` para os clientes (consumidores), o qual já provê o meio para que estes possam se cadastrar nos servidores. Para todos eles deve-se sempre chamar o método `exportObject` para que os mesmos possam se comunicar remotamente.

5.4. Exemplo de uma Aplicação Distribuída Construída com o fwGF

Nesta seção será apresentado um exemplo de uma aplicação distribuída que pode ser construída com base nos componentes aqui especificados para o fwGF. O objetivo desta aplicação é monitorar os equipamentos que constituem o *backbone* da RNP (Rede Nacional de Pesquisa), cuja estrutura apresenta-se como mostrado na **Figura 5.10**.

Portanto, suponha agora uma aplicação centralizada que monitore estes equipamentos conforme mostrado na **Figura 5.11**. Como se pode observar através dessa figura, dependendo da quantidade de informações, a escalabilidade dessa aplicação pode ser afetada. Além disso, este gerenciamento não é robusto, tendo em vista que se a estação de gerência falhar, todo o gerenciamento será comprometido.

Na **Figura 5.12** é apresentado o esboço de uma aplicação cuja funcionalidade é distribuída entre algumas estações de gerência. Note que aqui existem cinco estações, cada uma responsável por monitorar os equipamentos de uma determinada área (coincidentemente, cada área corresponde a uma região geográfica). Além disso, existem mais duas estações (chamadas aqui de estações principais) que recebem informações das outras cinco e registram-nas em um banco de dados. A aplicação de gerência é, portanto, dividida entre as estações de gerência, cada uma destas contendo um script BML que é executado para a instanciação dos componentes que a constituem. Considere, ainda, que o objetivo desta aplicação é monitorar o estado dos dispositivos de interconexão que formam este *backbone*.

Portanto, em cada estação deve-se instanciar um `Monitor` e um `GeradorDeEventoFalhaStatusEquipamento` para cada equipamento gerenciado[12]. Cada gerador é associado a um `Monitor` para poder receber os eventos por este gerado (`MonitorEvent`). Quando um evento indica que uma falha ocorreu, ou seja, que um

[12] Note que aqui não estamos citando os demais componentes que precisam ser instanciados, como ElementoGerenciadoSnmp, GrupoInfoGerenciaSnmp, etc.

equipamento não está operacional, o gerador envia eventos (`EventoFalhaEvent`) para o seus respectivos *listeners*.

Como os geradores instanciados exercerão o papel de servidores, pois devem gerar eventos para as estações principais, eles devem ser componentes remotos. Portanto, para cada gerador, devem ser executados os seus métodos `exportObject` e `register`, para exportá-lo e registrá-lo, respectivamente. Na chamada ao método `register`, deve ser informado o nome com o qual o componente será registrado no serviço de nomes (veja **Figura 5.6**).

Nas estações principais, são instanciados os componentes (`GeradorDeAlarmes`) da aplicação encarregados de armazenar as informações recebidas no banco de dados correspondente. Portanto, para estes componentes receberem os eventos enviados pelos geradores remotos, deve-se chamar o método `lookup` de cada um deles passando-se como argumento o nome do servidor.

Observe que, nesta aplicação, mesmo que ocorra uma falha em uma das estações principais, o gerenciamento da rede não é afetado. Além disso, uma falha em uma das outras cinco estações apenas afeta o gerenciamento naquela área. Entretanto, isto pode ser resolvido com uma sobreposição de tais áreas, analogamente ao que foi mostrado na **Figura 5.5**.

É óbvio que este é um exemplo simples, apenas utilizado para mostrar como pode-se construir uma aplicação com o fwGF. Entretanto, aplicações mais complexas podem ser arquitetadas. Por exemplo, pode-se montar uma aplicação onde algumas ações corretivas para a recuperação automática de falhas estejam a cargo de uma estação principal, enquanto outras ações podem ser colocadas em estações secundárias. Sendo assim, na ocorrência de uma determinada falha, uma estação pode aplicar as devidas ações corretivas sem a intervenção da estação principal (caso tais ações se apliquem a esta falha) ou pode apenas repassar a indicação da ocorrência da falha para que a estação principal tome as devidas providências para a solucionar o problema.

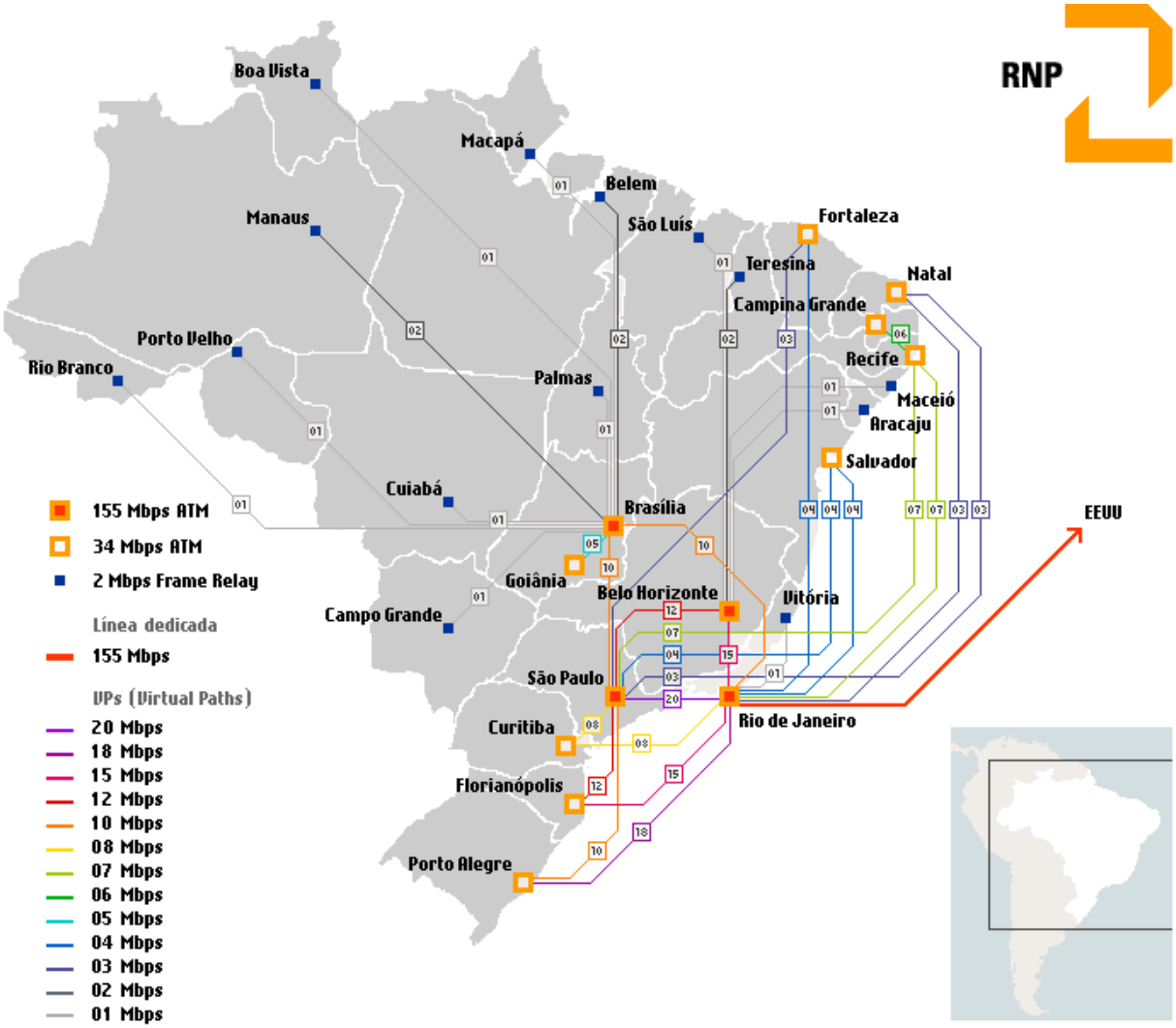


Figura 5.10. Estrutura do *Backbone* da Rede Nacional de Pesquisa (RNP)

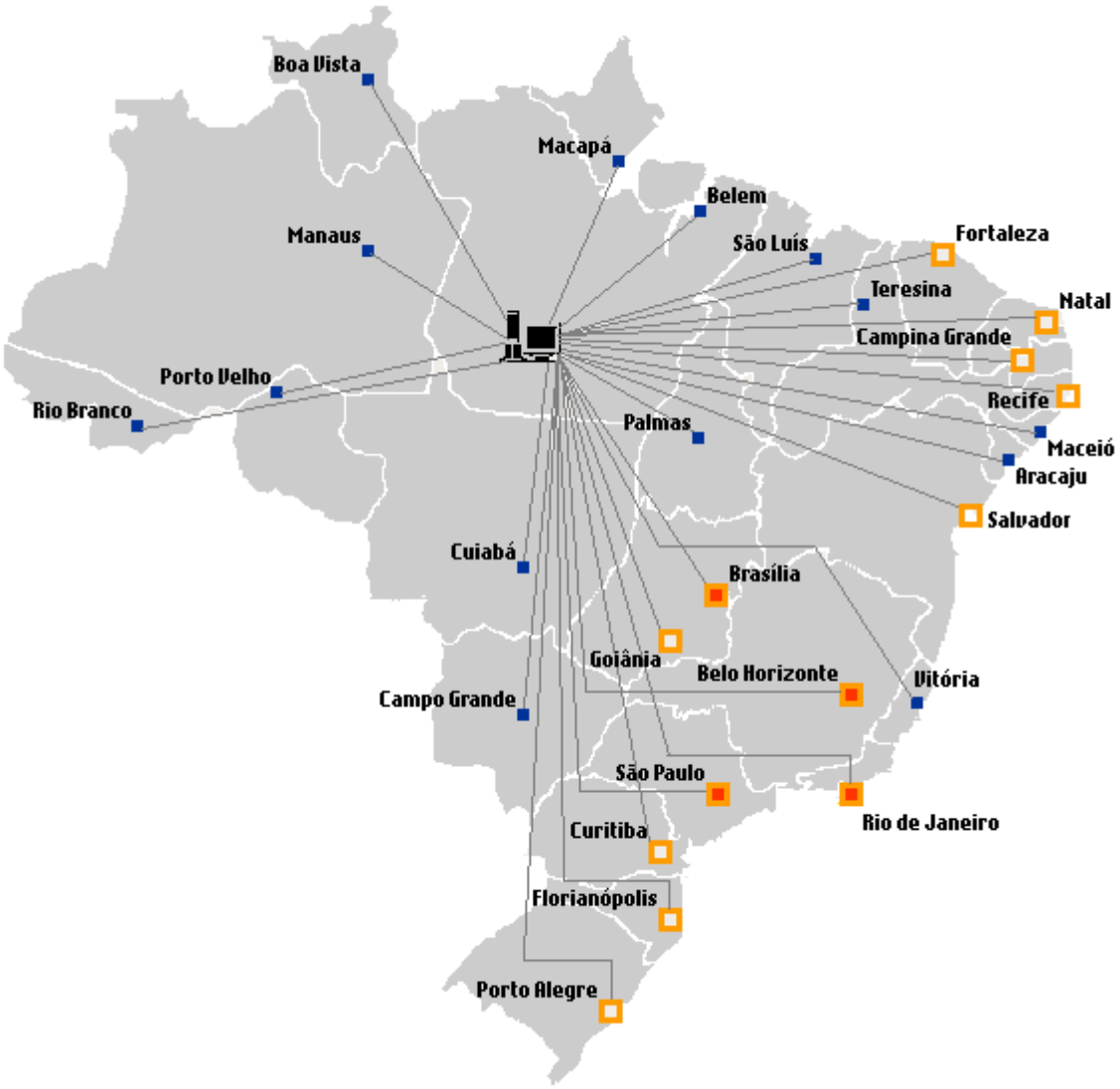


Figura 5.11. *Backbone* da RNP monitorado por uma aplicação de gerência centralizada

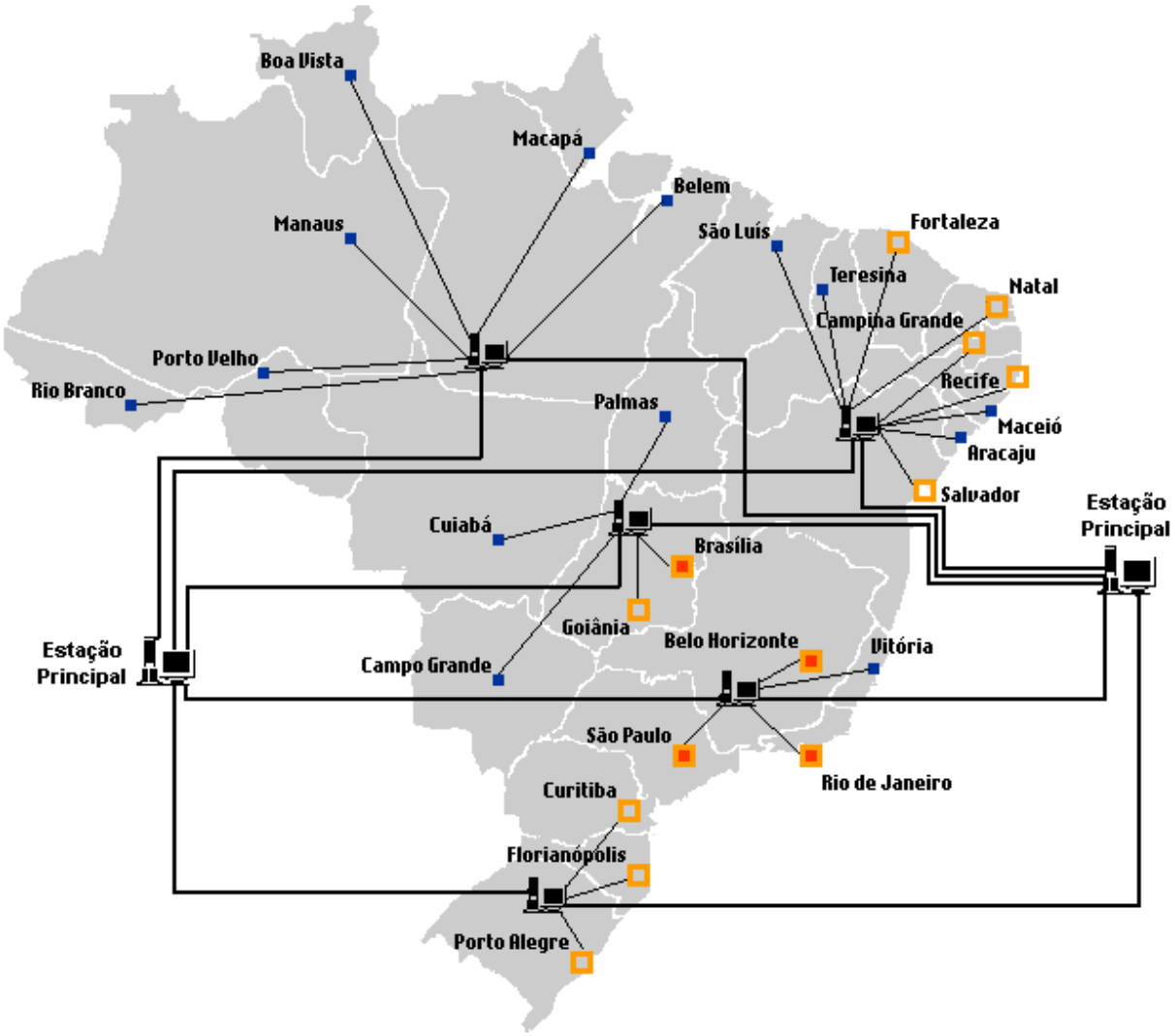


Figura 5.12. *Backbone* da RNP monitorado por uma aplicação de gerência distribuída

6. Conclusões

Neste trabalho foi efetuada a evolução do *framework* desenvolvido em [Freire, 2000], aqui denominado de fwGF, de acordo com o processo de evolução proposto por [Roberts & Johnson, 1996]. Este *framework* visa o desenvolvimento de aplicações de gerência de falhas em redes de computadores, fornecendo ao seu usuário um nível mais elevado de abstrações para a escrita de aplicações de gerência .

Como o fwGF encontra-se em [Freire, 2000] apenas como uma especificação, o trabalho aqui efetuado iniciou-se por sua implementação. Ao todo foram implementadas 5080 linhas de código, divididas em 50 classes e interfaces, com um total de 19 componentes. Durante a implementação também foram elaborados testes de unidade a serem executados pelo JUnit [Beck e Gamma, 1998] (um *framework* para automatização de testes de unidade). Estes testes são muito importantes em um produto de software, tendo em vista que os mesmos testam o comportamento de classes, estruturas de dados e condições limites sempre com o objetivo de se detectar problemas e corrigi-los para, ao final, assegurar a integridade e qualidade do produto.

Devido ao fato do fwGF ser um *framework* baseado em componentes, se faz necessária uma ferramenta que seja capaz de construir aplicações através da composição e instanciação destes componentes. Para isso, foi utilizada a linguagem BML (Bean Markup Language) que é uma extensão de XML (Extended Markup Language) e é voltada para a instanciação de componentes na linguagem Java.

Com relação à validação do fwGF, foi aplicado um método não-formal onde fora escolhido um subconjunto de aplicações, dentro do espaço de aplicações de gerência de falhas possível, para ser construído com o fwGF. Com relação àquelas aplicações que não puderam ser construídas, pode-se observar que a principal razão é a falta de um conjunto maior de componentes. Entretanto, essa limitação apresentada pelo fwGF é normal, tendo em vista que o mesmo encontra-se no início do seu processo de evolução. É óbvio que, com o passar do tempo e com a adição de novos componentes ao fwGF, essas limitações tendem a desaparecer.

De forma geral, o fwGF permite construir aplicações que façam a monitoração de variáveis de gerência, gerando eventos que caracterizam a ocorrência de falhas com base em limiares estabelecidos para as mesmas. Além disso, pode-se efetuar corelação de eventos com base em algoritmos simples, com o objetivo de limitar o número de eventos gerados. Por

outro lado, o fwGF não permite a construção de aplicações que envolvam a correlação de eventos com algoritmos mais elaborados (com base em informações topológicas da rede, por exemplo) ou que envolvam a semântica RMON [Stallings, 1996], devido justamente à falta de componentes que dêem suporte a essas características.

Por fim, continuando com o processo de evolução do fwGF, foi proposta uma adaptação ao mesmo para que ele pudesse dar suporte à construção de aplicações distribuídas de gerência de falhas. A especificação original deste *framework* é baseada no modelo de gerência centralizada. Entretanto, como as redes têm crescido bastante nos últimos anos, uma aplicação de gerência deste tipo é afetada pela falta de escalabilidade, tendo em vista a grande quantidade de equipamentos a ser gerenciada e a grande quantidade de informações a ser recebida e analisada pela mesma. Além disso, se ocorrer alguma falha nessa aplicação, perde-se todo o gerenciamento da rede.

A solução apresentada para que o fwGF possa construir aplicações distribuídas permite também que o programador possa construir aplicações centralizadas utilizando os mesmos componentes. Além disso, ela é uma solução bastante simples por dois motivos: primeiro, devido à própria natureza deste *framework*: sua orientação a componentes contribui bastante para um desacoplamento muito grande entre os componentes que o constituem; em segundo lugar, o padrão *Observer* [Gamma *et al.*, 1995] reforça esse desacoplamento. Com isso, a distribuição geográfica dos componentes instanciados torna-se simples e natural.

6.1. Trabalhos Futuros

Apesar dos componentes básicos do fwGF terem sido implementados nesse trabalho, este *framework* ainda demanda esforços de implementação para ampliar a sua funcionalidade e aumentar seu conjunto de componentes, contribuindo, dessa forma, para que novas aplicações possam então ser construídas.

Inicialmente, é preciso implementar as adaptações propostas nesse trabalho para que o fwGF possa dar suporte à gerência distribuída. Esta implementação refere-se à adição dos métodos correspondentes em cada classe do fwGF, reaproveitando o restante do código já escrito anteriormente. Uma vez implementadas, as adaptações propostas darão origem a um fwGF estendido, mais funcional e que permita a construção de aplicações mais escaláveis.

Em segundo lugar, é importante a construção de novos componentes para aumentar a funcionalidade do fwGF. Ainda faltam muitos componentes, entre os quais podem ser destacados: componentes para a correlação de eventos/alarmes, de configuração, auto-

descobrimento da topologia da rede, com melhor suporte a RMON, monitoramento de informação via TELNET/HTTP (ao invés de SNMP), para a gerência de serviços e para facilitar a construção de interfaces gráficas para as aplicações.

Seguindo o processo de evolução do fwGF, se faz necessária a construção de uma ferramenta gráfica para a construção de aplicações através da composição e configuração dos componentes de forma visual. A solução alternativa que foi aqui utilizada (BML) é eficiente quando se deseja construir aplicações pequenas. Porém, quando se tem redes de grande porte, a construção dos scripts BML torna-se cansativa devido à grande quantidade de componentes a serem configurados. Além disso, o desenvolvedor da aplicação está mais susceptível a cometer erros. Além desta ferramenta gráfica, a construção de componentes de auto-descobrimento de topologia (conforme já citada acima) é importante, pois estes ajudam na configuração das aplicações.

7. Bibliografia

[AdventNet, 2001] AdventNet SNMP API Release 3.2, disponível em: http://www.adventnet.com. 2001. Consulta: março/2001.

[Beck e Gamma, 1998] Beck, K., Gamma, E., *Test Infected: Programmers Love Writing Tests*. *On-line*: www.junit.org/junit/doc/testinfected/testing.htm. 1998. Consulta: maio/2001.

[Black, 1998] Black, D. P., *Managing Switched Local Area Networks: A Practical Guide*. Addison-Wesley, Massachusetts, USA, 1998.

[Dustin *et al.*, 1999] Dustin, E, Rashka, J. S., Paul, J., *Automated Software Testing: Introduction, Management and Performance*. Addison-Wesley, Massachusetts, USA, 1999.

[D'Souza e Wills, 1999] D'Souza, D. F., Wills, A. C., *Objects, Components and Frameworks with UML - The Catalysis Approach*. Addison-Wesley, Massachusetts, USA, 1999.

[Eckel, 2000] Eckel, B, *Thinking in Java*. Prentice Hall, New Jersey, USA, 2000.

[Fowler, 1999] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Massachusetts, USA, 1999.

[Fowler e Scott, 1998] Fowler, M., Scott, K., *UML Distilled – Applying the Standard Object Modeling Language*. Addison-Wesley, 1998.

[Freire, 2000] Freire, R. D., *Especificação de um Framework Baseado em Componentes de Software Reutilizáveis para Aplicações de Gerência de Falhas em Redes de Computadores*, Dissertação de Mestrado, UFPB, Campina Grande, Paraíba, Fevereiro de 2000.

[Gamma *et al.*, 1995] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Massachusetts, USA, 1995.

[Goldszmidt & Yemini, 1995a] Goldszmidt, G., Yemini, Y., *Decentralizing Control and Intelligence in Network Management*, Proceedings of the 4th International Symposium on Integrated Network Management, Santa Barbara, California, May 1995.

[Goldszmidt & Yemini, 1995b] Goldszmidt, G., Yemini, Y., *Distributed Management by Delegation*, Proceedings of the 15th International Conference on Distributed Computing Systems, June 1995.

[Johnson, 1999] Johnson, M., *Cover Story: Bean Markup Language, Part 1*, JavaWorld August 1999, disponível em: http://www.javaworld.com/javaworld/jw-08-1999/jw-08-beans_p.html. 1999. Consulta: abril/2001.

[Kaner, 1999] Kaner, C., *Testing Computer Software*, 2nd edition, John Wiley, USA, 1995.

[Kesselman e Duftler, 1999] Kesselman, J., Duftler, M. J., *Bean Markup Language: Tutorial*, IBM TJ Watson Research Center, NY, USA, 1999.

[Mainetti, 1997] Mainetti Jr., S., *Objetos Distribuídos*, Visionnaire/FAE-CDE, on-line: http://www.visionnaire.com.br/informes.htm, Curitiba, Paraná, Junho de 1997. Consulta: janeiro/2000.

[Martin-Flatin & Znaty, 1997a] Martin-Flatin, J. P., Znaty, S., *Annotated Simple Typology of Distributed Network Management Paradigms*, Technical Report SSC/1997/008, SSC, EPFL, Lausanne, Switzerland, March 1997.

[Martin-Flatin & Znaty, 1997b] Martin-Flatin, J. P., Znaty, S., *A Simple Typology of Distributed Network Management Paradigms*, Proceedings of the 8th IFIP/IEEE Int. Workshop on Distributed Systems: Operations & Management (DSOM'97), Sydney, Australia, pp. 13-24, October 1997.

[Martin-Flatin *et al.*, 1999] Martin-Flatin, J. P., Znaty, S., Hubaux, J. P., *A Survey of Distributed Enterprise Network and Systems Management Paradigms*, Journal of Network and Systems Management, Vol. 7, No. 1, pp. 9-26, 1999.

[Meira, 1997] Meira D. M., *Um Modelo para Correlação de Alarmes em Redes de Telecomunicações*. Phd Thesis, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 1997.

[Meira & Nogueira, 1997] Meira, D. M., Nogueira, J. M. S., *Métodos e Algoritmos para Correlação de Alarmes em Redes de Telecomunicações.* XV Simpósio Brasileiro de Redes de Computadores, São Carlos, SP, Brasil, pp. 79-98, Maio 1997.

[Roberts & Johnson, 1996] Roberts, D., Johnson, R., *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*. Pattern Languages of Programs Conference (PloP'96), 1996.

[Rogers, 1997] Rogers, G. F., *Framework-Based Software Development in C++*, Prentice Hall, New Jersey, USA, 1997.

[Rose e McCloghrie, 1995] Rose, M. T., McCloghrie, K., *How to Manage Your Network Using SNMP: The Network Managemet Practicum*. Prentice Hall, New Jersey, USA, 1995.

[Rose, 1996] Rose, M. T., *The Simple Book: An Introduction to Internet Management*, Revised Second Edition. Prentice Hall, New Jersey, USA, 1996.

[Rumbaulgh *et al.*, 1999] Rumbaugh, J., Booch, G., Jacobson, I., *The Unified Modeling Language Reference Manual*. Addison-Wesley, Massachusetts, USA, 1999.

[Stallings, 1996] Stallings, William. *SNMP, SNMPv2 and RMON: Pratical Network Management*. 2nd ed. Addison Wesley, Massachusetts, USA, 1996.

[Szyperski, 1998] Szyperski, C., *Component Software - Beyond Object-Oriented Programming*, Addison Wesley, Massachusetts, USA, 1999.

[Sun, 1999] Sun Microsystems Inc., *Java™ Platform 1.2 API Specification*. On-line: http://java.sun.com. 1999. Consulta: dezembro/1999.

[Sun, 2001] Sun Microsystems Inc., *The Java Tutorial: A Practical Guide for Programmers*. On-line: http://java.sun.com/docs/books/tutorial. 2001. Consulta: maio/2001.

[Waldbusser, 1995] Waldbusser, S. *Remote Network Monitoring Management Information Base*. Request for Comments 1757, Carnegie Mellon University, Pittsburgh, PA, USA, Fevereiro, 1995.

[Weerawarana e Duftler, 1999] Weerawarana, S., Duftler, M. J., *Bean Markup Language: User's Guide*, IBM TJ Watson Research Center, NY, USA, 1999.