ISEL

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Sistema de monitorização de aplicações e equipamentos de portagem

André Elias de Onofre Ferreira Lima
28838

Relatório do projecto realizado no âmbito de Projecto e Seminário do curso de licenciatura em Engenharia Informática e de Computadores sob a orientação de Paulo Araújo (ISEL) e co-orientação de Pedro Caetano (Pararede).

Setembro 2008

_____ André Lima

_____ Paulo Araújo

_____ Pedro Caetano

Resumo

Com o aumento das redes e do poder computacional, tanto de servidores, como de dispositivos dessas mesmas redes, são enormes as quantidades de tarefas executadas em tão curto espaço de tempo. Dessa forma, torna-se impraticável a execução de monitorizações manuais a tão grande número de servidores e dispositivos de rede e, consequentemente, essencial a implementação de um sistema de monitorização automatizado para aplicações e equipamentos de rede, que não só divulgue imediatamente o estado desses equipamentos, como também consiga, caso possível, concertar o problema automaticamente, sem intervenção dos respectivos administradores.

Este projecto resulta da necessidade da existência deste sistema automático de gestão que, mais tarde, poderá ser implementado em empresas com as mesmas necessidades que a BRISA. Esta define-se como o maior operador português de auto-estradas que, consequentemente, é detentora de uma enorme rede, entre os quais equipamentos de portagem, e servidores cujas aplicações processam uma enorme quantidade de dados fulcrais à empresa. Consequentemente, os objectivos deste projecto consistem na avaliação de aplicações compatíveis com padrões de monitorização (ex: SNMP e JMX), na recolha de parâmetros de sistema e de parâmetros aplicacionais, na disponibilização da informação respectiva à monitorização num centro de informação, na definição de limites aos valores monitorados e subsequente alarmística, através de vários canais como o *email* e JMS, caso os limites sejam violados, e no teste à integração do sistema com um *trouble ticket system* através de JMS.

Foram avaliadas três aplicações de monitorização muito conhecidas do mundo *open-source*: o Nagios, o Cacti, e o OpenNMS. Essa avaliação foi feita tendo em conta a capacidade de se constatar o estado actual de um dispositivo remoto (gestão de falhas), a capacidade de alteração da configuração de dispositivos remotos de forma a concertar/prevenir problemas (gestão de configurações), a manutenção de *logs* e capacidade de delinear a estrutura da rede, a capacidade de análise de dados respectivos ao funcionamento de um dado equipamento de rede, tipicamente apresentada em forma de gráficos (gestão de *performance*), e a análise à segurança de acesso aos dados de monitorização e à segurança da comunicação entre o terminal de monitorização e os agentes nos terminais remotos.

Entre as ferramentas avaliadas, o Nagios consiste na melhor com base na gestão de falhas e a nível de segurança, enquanto que o Cacti é melhor na gesta de *performance*, ou seja, exactamente onde o Nagios é mais fraco. Em relação ao OpenNMS, este consiste numa ferramenta generalista, pelo que apresenta o melhor equilíbrio entre todas as capacidades levadas em conta na comparação. Porém, foi da preferência do autor juntar o Nagios e o Cacti na implementação deste projecto, tendo assim, de uma forma geral, melhor desempenho do que o OpenNMS.

Das principais características do Nagios, a sua flexibilidade no que diz respeito à adaptação a novos tipos de monitorização e notificações, dada pela sua arquitectura baseada em *plugins* e *event handlers*, foi uma das principais condicionantes à sua escolha. Isto revelou-se particularmente importante na solução encontrada para a sua correcta integração com o JMX e o JMS respectivamente, especificações não suportadas nativamente.

Após a escolha das ferramentas, é dada uma solução de implementação de ambos os sistemas, o Nagios e o Cacti, delineando a arquitectura resultante da rede onde estes sistemas serão implementados. São ainda indicadas soluções a nível de configurações e de segurança, e do correcto modo de acesso de administradores aos terminais de monitorização.

Finalmente, é dado um caso de estudo onde se pretende implementar a arquitectura explicada previamente, de forma a demonstrar uma solução prática aos objectivos inicialmente traçados, com a correcta integração entre o Nagios e o Cacti.

Índice

1 Introdução

Neste projecto, pretende-se desenvolver uma arquitectura de monitorização de aplicações e equipamentos de rede, que permita ao(s) administrador(es) monitorar automaticamente essa mesma rede, e minimizar o tempo de resposta a falhas tanto das aplicações como dos equipamentos de rede, incluindo servidores.

Inicialmente são comparadas várias aplicações de monitorização tanto de aplicações como de equipamentos de rede, mais especificamente de portagens, com o intuito de implementar o resultado dessa mesma comparação, de forma a que seja integrado, de acordo com objectivos específicos, numa rede de grande escala como a da BRISA[1]. De salientar que este projecto resulta de uma parceria entre ISEL e a Pararede[39].

A BRISA é uma empresa que se define como o maior operador português de auto-estradas, com uma concessão principal de 11 auto-estradas. Essa realidade obriga à existência de um sistema de gestão de portagens que promova a interoperabilidade, dada a amplitude do próprio sistema. O sistema de portagens da BRISA designado por Toll Business Management Solution (TBMS) está organizado em três camadas:

1. Lane Management Service (LMS) – controla localmente sistemas situados em cada faixa da auto-estrada, como as antenas Dedicated Short- Range Communications (DSRC) ou Radio Frequency ID (RFID), o Automatic Vehicle Detection and Classifier (AVDC), displays, Advanced License Plate Recognition (ALPR), sendo que esta camada tem autonomia até vários dias, para o caso de falta de comunicação com a camada superior, descrita a seguir;

2. Toll Plaza Management Service (TPMS) – responsável pela gestão de toda a gama de serviços LMS, tendo comunicação em tempo real com o nível superior (3);

3. Central Coordination Service (CCS) – coordena a rede hierárquica de TPMSs e LMSs, fornecendo gestão de configuração, segurança, análise empresarial e monitorização através de interfaces intuitivas.

O TBMS tem como principais características: a infraestrutura "multi-vendor", tornando possível a associação de quaisquer dispositivos ao sistema, visto que são utilizadas tecnologias standard ou open source; a segurança, o acesso a serviços e a registo de eventos são feitos com garantia de confidencialidade e integridade utilizando algoritmos *standard* de cifra; a modularidade; a interoperabilidade, através da utilização de tecnologias baseadas em XML a integração de sistemas passa a ter um custo reduzido; e a autonomia, onde se procede à gestão e monitorização de serviços.

O grande número de praças de portagem existentes contém diversos terminais de input/ output que, por sua vez, geram enorme quantidade de dados fulcrais à lógica de negócios da BRISA. O correcto funcionamento desses terminais é consequentemente de importância extrema. Este projecto enquadra-se na necessidade da BRISA em monitorar, não só esses terminais como também, os servidores que processam os dados neles gerados. Isto deverá ser aplicado através de diagnósticos automatizados, manutenção preventiva, e alarmística de acordo com SLAs (Service Level Agreement), ou seja, parâmetros que servirão de comparação para que a aplicação saiba se os valores verificados excedem ou não os limites pretendidos pelo administrador.

Os objectivos deste projecto consistem no seguinte:

- avaliação de aplicações compatíveis com *standards* de monitorização(ex: SNMP e o JMX);
- recolha de parâmetros de sistema, como a quantidade de disco ocupada;
- recolha de parâmetros aplicacionais;
- disponibilização da informação respectiva à monitorização num centro de informação;
- definição de SLAs (*Service Level Agreements*);
- alarmística através de vários canais, como e-mail e JMS;
- teste à integração do sistema com um *trouble ticket system* através de JMS.

No capítulo 2 é feita a avaliação das ferramentas mais conhecidas, através da especificação, antes de mais, dos parâmetros a serem avaliados; a seguir explicam-se dois dos principais protocolos de monitorização, o SNMP e o JMX; analisam-se três ferramentas muito conhecidas no mundo *open-source,* o Nagios, o Cacti, e o OpenNMS; faz-se uma rápida análise à restante gama de ferramentas do mesmo género e, finalmente, justifica-se a conclusão a que se chega. No capítulo 3 é apresentada a arquitectura de monitorização que consiste na solução para a execução das duas ferramentas escolhidas de forma a que se atinjam os objectivos pretendidos, e é explicada a solução encontrada para a integração do sistema de monitorização com o sistema de *trouble ticket* utilizando alarmística JMS. No capítulo 4 é implementado um caso de estudo, que pretende demonstrar uma solução prática que atinja os objectivos delineados. Finalmente, no capítulo 5, é concluído o projecto analisando-se os problemas, as soluções, e trabalhos futuros.

2 Avaliação de ferramentas *standard* de monitorização

Antes de se efectuarem quaisquer comparações, é importante que se saiba que parâmetros utilizar nessa comparação, tendo-se em conta a finalidade e os objectivos do projecto. Nesta secção são delineados exactamente os parâmetros avaliados pelo autor na escolha da ferramenta ideal para implementar este mesmo projecto.

2.1 Parâmetros de avaliação

Existem dois modelos de gestão de redes, fundamentais na criação ou escolha de ferramentas de monitorização, que foram desenvolvidos em 1996. A *International Telecommunication Union*[2], ITU, introduziu nessa altura o modelo *Telecommunications Management Network*[3] TMN, constituída por quatro camadas lógicas: *Business Management*, *Service Management*, *Network Management* e *Element Management*. Mais tarde, em 1997, publicou um modelo que consistia na especificação da 3ª camada, *Network Management*, chamada modelo FCAPS[4]:

- Gestão de Falhas (***F**ault Management*) - verificação da disponibilidade de dispositivos remotos e de serviços públicos;
- Gestão de Configurações (**C**onfiguration Management) - alteração de configurações respectivas aos dispositivos que se monitoram;
- Relatório de Actividades (***A**ccounting*) - manutenção de relatórios com informação respectiva a quem faz o quê na rede;
- Gestão de *Performance* (***P**erformance Management*) - recolha de dados e sua posterior apresentação gráfica;
- Segurança (***S**ecurity*) - controle ao acesso aos dados disponibilizados pela aplicação e segurança na comunicação entre monitores e agentes.

Estes dois modelos são frequentemente utilizados como listas de funcionalidades a implementar por parte de quem desenvolve ferramentas de monitorização, e farão parte da análise aqui apresentada.

Em relação à segurança, o terminal de monitorização pode ser definido, na perspectiva de atacantes informáticos *(hackers),* como uma *backdoor* para a rede onde se deseja penetrar. Muitas vezes, estes terminais têm acesso a redes através de *firewalls*, sendo-lhes permitido obter informação acerca dos recursos que estão a monitorar. Porém essa informação pode ser utilizada para atacar a rede em causa. Essa conveniência torna os terminais de monitorização num sistema alvo, pois uma vez comprometida a segurança nestes, será mais fácil comprometer a segurança da rede que se deseja atacar. Como exemplo, temos o caso em que o administrador utiliza chaves *Secure Shell* (SSH) partilhadas.

Outro aspecto importante consiste na verificação da autenticidade das mensagens recebidas. Um atacante pode facilmente entregar informação falsificada à estação de monitorização fazendo com que este apresente dados estatísticos falsos, e despolete eventos com base em notificações falsas, o que pode ser um enorme problema caso tenham-se eventos que, por exemplo, reiniciem serviços, ou reiniciem o sistema (*cycling power*), ou ainda que enviem sms's falsos causando deslocações desnecessárias a administradores.

Outro potencial problema, consiste na confidencialidade das mensagens trocadas entre o o monitor e os clientes. Isto é importante pois existe a possibilidade de um atacante fazer *sniffing* à rede e obter informação que lhe ajude a efectuar outros ataques à rede ou aos sistemas em causa. Esta confidencialidade é obtida através da cifra dos canais de comunicação. Caso isto não seja implementado, corre-se o risco de se sofrerem ataques, como por exemplo, se um atacante souber a carga de processador e o número de utilizadores dum sistema, utilizados ao longo de um dia,

fica automaticamente a saber quando poderá invadir esse sistema e utilizar os recursos por este disponibilizado sem que ninguém se aperceba do facto.

Existem, no entanto, duas ameaças das quais essas ferramentas não se deverão proteger: ataques DoS e análise de tráfego. Os ataques de DoS resultam na impossibilidade de obtenção de dados por parte do monitor, dada a falta de disponibilidade resultante deste tipo de ataques. Porém, essa indisponibilidade é praticamente impossível de distinguir de falhas de rede, que deverão ser familiares a qualquer administrador, e que deverão ser tidas em conta na implementação destas ferramentas. Quanto à análise de tráfego, o tráfego de dados é feito de forma periódica, pelo que a análise desse fluxo de tráfego não tem significado algum.

Dados todos estes problemas, torna-se óbvio que a segurança é um factor crítico na escolha da ferramenta a utilizar, sendo este consequentemente o elemento de maior peso nessa escolha. Alerta-se ainda para o facto de, por muito segura que seja a ferramenta, se a segurança do sistema onde esta se executa não estiver garantida, de nada servirá o acréscimo que a segurança da ferramenta irá trazer.

A seguir faz-se uma análise aos protocolos SNMP (secção 2.2) e JMX (secção 2.3), que constituem parte importante do processo de monitorização. Mais tarde, são analisadas as aplicações OpenNMS (secção 2.4), Nagios (secção 2.5) e Cacti (secção 2.6) e outras aplicações (secção 2.7). Finalmente é feita uma apreciação final e respectivas conclusões no que diz respeito às comparações (secção 2.8).

2.2 Simple Network Management Protocol (SNMP)

O SNMP [5] é o protocolo *standard* quando se refere à gestão de recursos remotos, e tem como objectivo gerir dispositivos tirando proveito da família de protocolos TCP/IP. O SNMP utiliza o conceito de *manager*, que gere esses dispositivos, e de *agents,* que são os dispositivos geridos, como por exemplo, um *router*. Este protocolo foi concebido na camada sete do modelo OSI de forma a que a sua utilização seja indiferente às implementações de diferentes fabricantes, e aos diferentes tipos de rede que possam estar entre o *manager* e o *agent*.

O *manager* consiste numa estação de gestão que executa a versão cliente SNMP, que posteriormente, deverá requisitar informação à versão servidor SNMP executada no *agent*. A informação relacionada com a *performance* do *agent* é armazenada numa base de dados local, cujo acesso é permitido ao *manager*. Por exemplo, um *router* pode armazenar dados relativos ao número de pacotes transaccionados, e o *manager* pode ler esses valores e concluir se esse *router* está ou não congestionado. O *manager* também tem a capacidade de executar acções nos *agents*, por exemplo fazer com que um *router* reinicie após conclusão de que este se encontra bloqueado devido a um congestionamento. Contudo, os *agents* também podem contribuir em todo esse processo de gestão, através do envio de mensagens após verificação de anomalias. Essas mensagens são chamadas de *trap*.

Resumindo, o processo de gestão no SNMP reside em três conceitos básicos:

- O *manager* requisita informação ao *agent* necessária à sua monitorização;
- O *manager* provoca a execução de acções no *agent* como reacção a alterações feitas pelo próprio *manager* em valores situados na base de dados do *agent*;
- O *Agent* contribui para o processo de gestão através do envio de mensagens *trap* como aviso de anomalias.

Quanto às versões, a primeira foi o SNMPv1[6] com muitas das funcionalidades, a nível de segurança, do SNMPv3[7]. Um exemplo é a autenticação, pois o SNMPv1 antecipou essa necessidade mas solucionou-a com a implementação de um esquema de autenticação trivial baseada em comunidades. O SNMPv3 expandiu o conceito de autenticação[8] de forma a abrangir outros serviços como a confidencialidade. Essa autenticação baseia-se num esquema de cifra

assimétrica onde uma *passphrase,* administrativamente inserida, é utilizada para gerar duas chaves: uma *authKey* que é utilizada para gerar um *hash* da mensagem; e uma *privKey* que é utilizada para cifrar a mensagem de forma a obter a desejada confidencialidade. Outro exemplo é o controlo de acessos baseado em *views* que é também encontrado no SNMPv3. Esse controle de acessos[9] utiliza um *username* e uma *password* também inserida administrativamente, para atribuir, ou não, uma determinada permissão a um pedido.

A seguir ao SNMPv1, aparece o SNMPv2[10] onde algumas das melhorias foram:

• tipos estendidos - versões a 64bits como o *counter*;

• maior eficiência - comando *get-bulk*;

• melhoria no tratamento de erros e excepções.

Porém, esta versão, tecnicamente conhecida como a original SNMPv2p[10], falha no cumprimento de algumas funcionalidades a nível de segurança, pelo que foram posteriormente lançadas novas versões: SNMPv2c[11] baseada em comunidades e SNMPv2u[12] com o *User-based Security Model*[8]. Existiu também um SNMPv2*, porém este nunca se tornou num padrão.

Para garantir os serviços de confidencialidade, integridade e autenticidade, são implementados mecanismos [13] onde se solucionam as ameaças acima mencionadas:

• Integridade: é criado um *hash* de toda a mensagem que é concatenado à própria mensagem;

• Autenticidade: antes da criação do *hash*, é adicionado à mensagem um valor conhecido só pelo *manager* e pelo *agent*;

• Reordenação de mensagens: é adicionado, a cada mensagem, um *timestamp*;

• Confidencialidade: é utilizado um algoritmo de cifra simétrico.

Esses são algumas das funcionalidades abordadas pelo SNMPv3 que visam complementar o SNMPv2 e melhorar o SNMPv1. Apenas de realçar a obrigatoriedade da implementação de todos esses mecanismos em simultâneo, para que sejam atingidos os objectivos pretendidos.

Em seguida analisa-se a especificação JMX, *standard* da empresa SUN.

2.3 Java Management Extensions (JMX)

O JMX [14] é uma especificação da linguagem Java que define uma arquitectura, padrões de desenho, API's, e serviços para gestão e monitorização de aplicações e de serviços de rede.

Utilizando esta tecnologia, a gestão de um dado recurso é feita da seguinte forma: o recurso é associado a um, ou mais, objectos Java, conhecidos por *Managed Beans* (MBeans). Esses MBeans registam-se num servidor central de gestão de objectos, denominado *MBean Server*. Se a este *MBean Server* associarmos os serviços necessários para a gestão dos recursos a ele relacionados, teremos aquilo a que se chama de um JMX *Agent*. Estes JMX *Agents* são os responsáveis pela disponibilização da informação para monitorização dos recursos a eles associados, e pela gestão desses recursos conforme ordem de aplicações remotas de monitorização. O JMX ainda disponibiliza uma forma *standard* de se acederem aos JMX *Agents*, através dos chamados JMX *Connectors*. Estes são parte crucial dos JMX *Agents*, sendo que pelo menos um deverá existir. De salientar ainda que independentemente do protocolo utilizado pelos JMX *Connectors*, a interface disponibilizada é sempre a mesma, de forma a permitir uma gestão transparente, ou seja, que a aplicação de gestão remota não se preocupe com que protocolo se encontra a ser utilizado.

A utilização do JMX tem as seguintes vantagens:

• Facilidade de implementação: uma aplicação Java apenas necessita de tornar as funcionalidades, que dela se desejam monitorar, disponíveis num ou mais *MBeans*, e posteriormente registá-los num JMX *Agent*. O JMX disponibiliza uma forma *standard* de gestão para qualquer aplicação, serviço, ou dispositivo Java;

• Escalabilidade: a arquitectura dividida em camadas, permite fácil remoção e recolocação de componentes;
• Toma partido de tecnologias *standard* Java através da referência a tecnologias como *Java Naming and Directry Interface* (JNDI) ou *Java Database Connectivity API* (JDBC), pois essas referências nunca são feitas pela concorrência.

Quanto ao controle de acessos aos *MBean Servers*, este é baseado no modelo de segurança do Java conhecido por *Java Security Model* [15], mais propriamente em mecanismos de controle de acessos[16] definindo permissões que controlem o acesso ao *MBean Server*. O objectivo deste mecanismo consiste em autenticar, não o utilizador, mas sim o código que se encontra a executar. Isto evita problemas como a utilização não autorizada de recursos resultantes da execução de código não autorizado, como por exemplo *Java Applets*. Assumindo o cenário em que o método M1 invoca o método M2, que faz controle de acessos, o M2 fará uma verificação para ver se a permissão que este exige para sua execução está contida no conjunto de permissões detidas pelo M1. Essa verificação é feita apenas passando como parâmetros, o alvo da acção, ou das acções pretendidas, e as acções que se pretendem efectuar sobre esse alvo (por ex. requisição de escrita sobre um ficheiro). A única informação que resta saber, é a de que permissões se encontram na posse do método M1. Porém, isso só se sabe em tempo de execução. É por isso que a entidade responsável pela verificação de permissões é capaz de carregar em tempo de execução toda a informação relativa a M1, incluindo as respectivas permissões, que deverão ser comparadas às requisitadas por M2. Caso a comparação não resulte em permissão de acesso, é lançada uma excepção *SecurityException*.

A atribuição de permissões é feita estaticamente através de um ficheiro de configuração. Esse ficheiro é parte crucial na verificação de permissões, pelo que é carregada em tempo de execução, pela entidade responsável por essa mesma verificação. Outra forma de atribuição de permissões foi proposta por dois pesquisadores da *Helsinki University of Technology* numa publicação [17] onde esta é feita dinamicamente recorrendo acertificados *Simple Public Key Infrastructure*[18].

Para além da segurança associada ao acesso aos *MBean Servers*, é necessário também acrescentá-la no acesso aos *connector servers* que correspondem aos que ficam à espera de pedidos numa porta TCP, pedidos esses originados em aplicações remotas de monitorização. O problema é que não se sabe se a aplicação tem ou não permissão de acesso à informação que requisita.

Existem dois tipos de *Connectors*: o *Remote Method Invocation Connector* (RMI *Connector*) que é o único cuja implementação é obrigatória segundo a especificação JMX; e o *Generic Connector* cuja implementação é opcional. Ambos conseguem autenticar clientes com base em protocolos distintos, mas o resultado é o mesmo: um objecto denominado *JAAS*[19] *Subject* que representa a identidade autenticada. O RMI *Connector* fornece um nível de segurança básico que não é adequado a todas as configurações de segurança, sendo que o *Generic Connector* abrange requisitos mais avançados, incluindo por exemplo a utilização do protocolo TLS.

A seguir analisa-se uma solução de monitorização de nível empresarial que é das mais conhecidas, o OpenNMS.

2.4 OpenNMS

O OpenNMS[20] endereça as camadas Service Management e Network Management do modelo TMN e endereça igualmente todo o modelo FCAPS, com especial ênfase à gestão de falhas e gestão de performance. A gestão de falhas é feita recorrendo ao *polling*, à recepção assíncrona de mensagens, como é o caso de SNMP *traps*, e recorrendo a *thresholds* comparando-os a dados de performance. A gestão de performance é feita recorrendo a protocolos como o SNMP, JMX e à comunicação com agentes como o NSClient. Quanto à gestão de configurações esta é feita através

da WebUI. O relatório de actividades não é apresentado na parte gráfica mas pode ser enviado a estações externas que queiram processar esses dados. Finalmente, em relação à segurança, esta é suportada:

- através da integração do SNMPv3;
- através do controle local ou remoto (LDAP) a acessos à interface gráfica;
- através da integração com os sistemas de detecção de intrusões Snort[21] e avaliação de vulnerabilidades Nessus[22].

A próxima aplicação em análise é o Nagios e consiste numa das melhores soluções *Open Source* para monitorar sistemas remotos, reconhecida principalmente pela sua flexibilidade e estabilidade.

2.5 Nagios

O Nagios [23] consiste numa aplicação de monitorização de sistemas e de redes especializada, em termos do modelo FCAPS, na área de gestão de falhas. Essa monitorização é flexível e versátil devido à existência de ficheiros de configuração e à possibilidade de não só se poderem utilizar *plugins* diversos já existentes, como também criar novos de acordo com as necessidades do utilizador. Pode-se utilizar esta aplicação para monitorar serviços de rede, como a disponibilidade de servidores POP, SMTP e HTTP, e também pode-se-lhe utilizar para monitorar recursos na máquina local ou em máquinas remotas, como a carga do processador e espaço disponível no disco rígido. Esta aplicação, na verdade, consiste apenas num *daemon* que gere o processo de monitorização. A recolha dos dados, para posterior processamento por parte do Nagios, é feita por pequenas aplicações conhecidas por *plugins*[24], que retornam os dados ao *daemon* em causa.

As máquinas e serviços de rede monitorizados pelo Nagios são todos definidos através dos ficheiros de configuração que contêm informação como os contactos que deverão ser informados de possíveis falhas, informação dos *hosts* a serem monitorizados, comandos existentes, sendo que um comando consiste na definição de que *plugin* deverá ser executado e dos respectivos parâmetros, e a definição de serviços que consistem na associação do *host* a ser monitorizado, do comando a ser executado, dos contactos a serem informados em caso de alertas, e outras informações necessárias a essa monitorização como, por exemplo, o número máximo de tentativas.

Em relação aos *outputs*, o Nagios traz uma interface *web* integrada, que deve ser integrada à instalação local do *Apache*. É possível também verificar informação de estado consultando o ficheiro de *log*.

Quanto à segurança, a confidencialidade, integridade e autenticidade obtêm-se através da utilização do protocolo *Secure Socket Layer* (SSL). Porém existem aspectos que requerem alguma ponderação por parte do administrador, como é o caso da monitorização remota, onde a ferramenta oficial é o *Nagios Remote Plugin Executor* (NRPE), sendo que a outra possibilidade consiste na utilização do SSH. Ambas apresentam as mesmas vantagens, contudo cada uma tem uma desvantagem. O NRPE abre mais uma porta TCP, o que vai contra o minimalismo desejado do ponto de vista de segurança, e o SSH consome maior carga de processador, problema esse que se agrava com o número de ligações abertas simultaneamente. Seguem-se alguns dos mais importantes exemplos de recomendações dadas na documentação oficial:

- Utilização de um terminal dedicado: isto reduz o risco de outras aplicações serem comprometidas e, através delas, se comprometer todo o sistema;
- Não executar o Nagios como utilizador *root*: o Nagios não necessita dos privilégios associados a esse utilizador para se executar, e não se executando nessa conta diminui-se o número de problemas que poderão ser causados por um atacante que se apodere da máquina;

• Restringir o acesso à directoria configurada como *check_result_path*: nessa directoria, indicada no ficheiro de configuração principal do Nagios, só deverá ter acessos de leitura e escrita, o utilizador *nagios*, pois nela são armazenados os dados retornados pelos *plugins* temporariamente, antes de serem processados. Caso o acesso a essa directoria não seja vedada, poder-se-ão ter os mesmos problemas associados à falta de autenticidade e confidencialidade, causados por outros utilizadores do mesmo terminal;

• Restringir o acesso ao ficheiro configurado como *External Command File*: esse ficheiro, indicado no ficheiro de configuração principal do Nagios, funciona como um *buffer* FIFO, onde aplicações externas (ex. CGI's) podem alterar o processo de monitorização em funcionamento, como por exemplo: forçando verificações de serviços de forma a que *sniffers* sejam postos em execução; alterando comandos, ou seja, alterar o *plugin* chamado e/ou os parâmetros a ele associados; e através do cancelamento temporário de notificações;

• Proteger o acesso aos agentes remotos: como agentes remotos entendem-se aplicações como o já indicado NRPE, que consistem em pontes que fazem a comunicação entre os *plugins* e o Nagios. Outro exemplo seria a aplicação que faz a ponte entre o Nagios e sistemas Windows, que é o *NSClient*. Essa protecção deve ser aplicada pois não se quer que todos tenham acesso à informação disponibilizada por esses agentes remotos;

• Assegurar a confidencialidade dos canais de comunicação existentes entre o Nagios e os agentes remotos.

Na secção seguinte, analisa-se a aplicação Cacti que consiste numa excelente ferramenta de análise de desempenho.

2.6 Cacti

O Cacti[25] consiste numa aplicação especializada na área de monitorização de *performance*, sendo uma muito boa escolha para quem deseja monitorar o desempenho de dispositivos remotos. Esta aplicação faz a recolha, o armazenamento, e a apresentação gráfica de dados inerentes à *performance* de um dado dispositivo como, por exemplo, a carga de CPU, quantidades de memória ocupadas, ou larguras de banda consumidas numa interface.

O Cacti é constituído por uma interface gráfica, que utiliza a tecnologia PHP, para consultar uma base de dados MySQL[26], gerida também pelo Cacti. A informação é inserida, nessa base de dados, utilizando uma ferramenta denominada RRDTool[27] (*Round Robin Database Tool*) que consiste numa ferramenta que armazena e apresenta dados que se alteram com o passar do tempo, como são os casos da temperatura numa sala de servidores ou da largura de banda consumida numa interface. Por sua vez, o RRDTool utiliza o conjunto de aplicações SNMP, contidas no pacote net-snmp[28], para a recolha dos dados a armazenar na base de dados. Isto tudo significa que os seguintes softwares são cruciais na instalação do Cacti:

- net-snmp: responsável pela recolha de dados;
- RRDTool: responsável pelo armazenamento;
- Apache server (httpd): servidor web;
- MySQL (server): base de dados onde serão armazenados os dados;
- PHP: linguagem utilizada pela interface web para consulta dos dados a apresentar.

O maior poder desta aplicação consiste na representação gráfica de dados que variam com o tempo. A facilidade em gerar e gerir gráficos, como os de performance, é amplamente reconhecida. Outra grande funcionalidade é a gestão de utilizadores, que permite ao administrador criar vários utilizadores, da interface, atribuindo a cada um diferentes níveis de permissões, fazendo com que estes tenham acesso a diferentes gráficos com informações distintas., visto que essas permissões podem ser especificadas em relação a cada gráfico.

Com o aumento do número de dados a recolher, o *poller*, ou seja, o *script* responsável pela recolha e armazenamento dos dados, começa a ter problemas de desempenho. Para solucionar este problema foi desenvolvido um *poller* chamado *Spine*[29], escrito utilizando a linguagem C, de forma a torná-lo mais eficiente, não só pelo código nativo, mas também pelo facto de este tomar partido de *threads* do sistema operativo (*pthreads*).
Uma das grandes falhas desta aplicação consiste no facto desta não conter a funcionalidade de notificar administradores quando os valores de um determinado gráfico saem dos limites especificados por um *threshold*.
A nível de segurança, esta aplicação tem suporte completo ao protocolo SNMPv3.

2.7 Outras aplicações

Existem inúmeras aplicações de gestão/monitorização todos com o mesmo objectivo de monitorar e desencadear acções em dispositivos remotos em resposta à própria monitorização. Muitos especializam-se em certas áreas do modelo FCAPS, como já vimos, e outros tentam abrangi-los ao máximo. É importante deixar claro que as ferramentas aqui apresentadas não são as únicas, mas sim as melhores nas suas áreas de acordo com a avaliação do autor deste documento. Como exemplo, tem-se o Hobbit[30], inspirado no Big Brother[31], que consiste num monitor centralizado que recebe dados de *softwares* instalados nas máquinas que se desejam monitorar. Porém, este não fornece tantas funcionalidades quanto o Nagios, por exemplo. Outra aplicação de monitorização é o Monit[32] que, por sua vez, consiste numa ferramenta destinada a uma utilização mais modesta, tipicamente nos próprios sistemas que se desejam monitorar. Esta é particularmente conhecida pela sua muito boa integração com o *init* e os *rc-scripts*, o que lhe confere a capacidade de reiniciar serviços que poderão ter falhado. Finalmente, têm-se várias outras aplicações cuja única funcionalidade consiste na interpretação de mensagens SNMPv3.

2.8 Conclusão

Com base na análise aqui feita, fica claro que não existe uma ferramenta que simplesmente se caracterize como sendo a melhor de todas. A conclusão, a que se chega, é a de que, com base nas características aqui apresentadas, fica a cargo de cada administrador escolher a solução que melhor lhe sirva.

O OpenNMS é a aplicação mais completa, detentora das capacidades de analisar a disponibilidade de dispositivos remotos, alterar a configuração desses dispositivos conforme as conclusões chegadas a partir da própria monitorização, descoberta automática de dispositivos de rede, analisar a *performance* desses dispositivos demonstrando os resultados em gráficos intuitivos, e com suporte ao protocolo SNMPv3.

Por outro lado, temos dois especialistas, o Nagios e o Cacti. O Nagios especializa-se na monitorização de dispositivos remotos suportando, não só a comunicação via SNMP, mas também toda a flexibilidade e personalização proveniente das capacidades dos *plugins* que podem ter inúmeras utilidades não suportadas pelo SNMP. Essa especialização não significa a falta de suporte às outras funcionalidades, pois o Nagios também tem excelente suporte a nível de segurança, tanto na utilização do SNMPv3 como na comunicação com os respectivos agentes, tem toda a capacidade de alterar a configuração de dispositivos remotos dada pelo SNMP e tem excelente suporte para notificações.

Por último temos o Cacti, especializado na análise de desempenho, capaz de ler informação contida, não só em bases de dados MySQL como também, em ficheiros no formato RRD *files*. Ao contrário do Nagios, o Cacti, apenas contém a funcionalidade onde se especializa. Isto tudo resume-se na figura 2.1:

	AVALIAÇÃO	NOTAS	AVALIAÇÃO	NOTAS	AVALIAÇÃO	NOTAS
F	Melhor	Notificações flexíveis, verificações directas e indirectas	Bom	Notificações	ND	
C	Excelente	SNMP, expansível a aplicações próprias	Bom	SNMP	ND	
A	Bom	Delineamento da rede	Excelente	Delineamento de rede	Mau	
P	Mau	MRTG	Excelente	RRDTool	Melhor	RRDTool, Spine
S	Melhor	SNMPv3, SSL/TLS, controle de acessos	Excelente	SNMPv3, integração com Nessus e Snort, LDAP, controle de acessos	Bom	SNMPv3
MÉDIA	Excelente		Excelente		Bom	

Figura 2.1 - Principais diferenças entre as ferramentas avaliadas

Apesar do facto do OpenNMS ser o mais completo, é da preferência do autor a implementação tanto do Nagios, como do Cacti pois estas duas aplicações juntas complementam-se e dão maior flexibilidade à monitorização, o que permite endereçar problemas específicos à entidade onde se pretende implementar o sistema de monitorização.

3 Implementação

Neste capítulo é explicada a arquitectura de implementação proposta pelo autor. Esta arquitectura permitirá que, futuramente, este projecto seja correctamente implementado em casos práticos tirando-se partido de todas as funcionalidades oferecidas pelos dois sistemas escolhidos, o Nagios e o Cacti, de forma a atingir os objectivos pretendidos. Na figura 3.1, encontra-se representada a estrutura da rede correctamente associada ao terminal de monitorização:

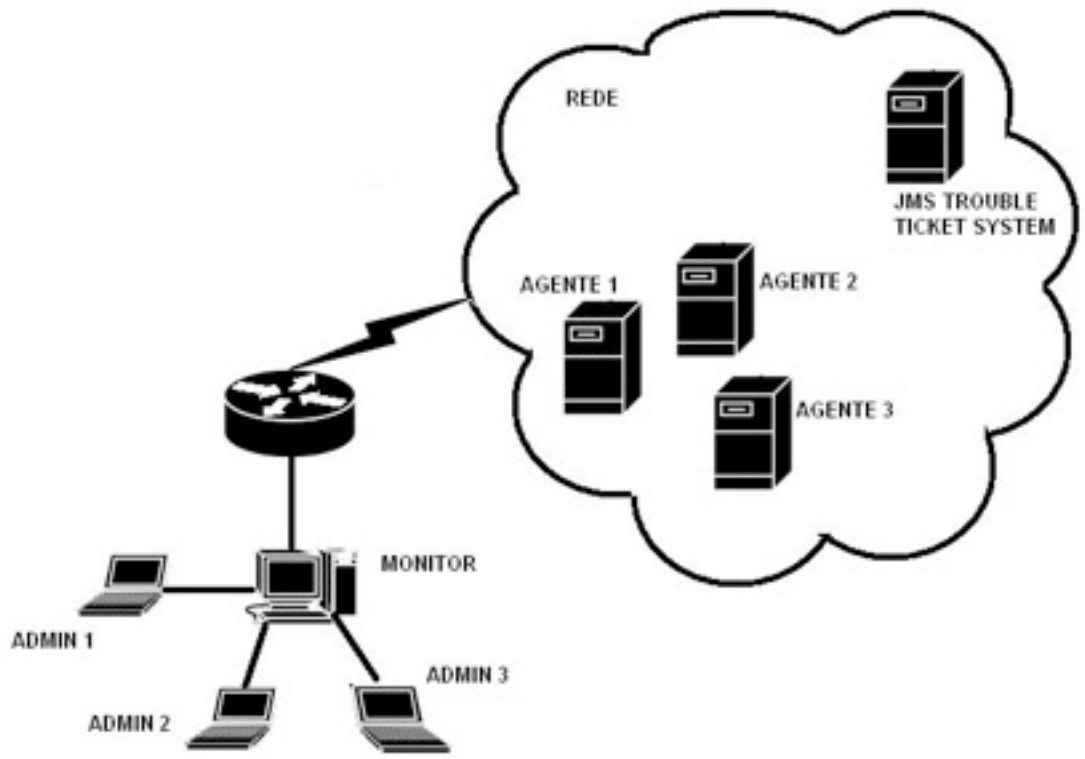

Figura 3.1 - Arquitectura da monitorização proposta

Com esta arquitectura, têm-se ambas as ferramentas a serem executadas numa só estação de trabalho dedicada somente à tarefa de monitorização. Isto permite reduzir o número de portas TCP em utilização ao mínimo possível, tornando mais fácil o controle dessas portas, e diminuindo o risco de se terem outras aplicações, com funções distintas à da monitorização, a fornecerem potenciais falhas de segurança que permitiriam acesso não autorizado à estação de monitorização. Por essa razão, e também pelo facto de se desperdiçarem recursos a nível de *hardware*, é altamente recomendado a utilização de toda a estação de trabalho somente para tarefas de monitorização. O controle, aos serviços TCP prestados pelas ferramentas, deverá ser feito através da implementação de TCP *wrappers* e através da configuração de *firewalls*. De salientar que a estação de monitorização tem apenas um caminho para comunicar com o resto da rede. Isto torna mais fácil o controle de acessos visto que, caso tivesse várias ligações através de vários *routers*, ter-se-iam que colocar diferentes regras consoante as redes a que o *router* desse acesso, o que tornaria a manutenção numa tarefa complexa. Contudo, o controle de acessos não pode ser feito apenas à estação de monitorização, mas também no acesso aos agentes, pois estes têm o potencial de fornecer informação a quem não tenha acesso a ela. Por essa razão deve-se, não só controlar

acessos aos agentes como também, assegurar a confidencialidade nos canais de comunicação, ou seja, não permitir que terceiros utilizem ferramentas que lhes permitam ler dados que são transmitidos na rede, conhecidos por *sniffers*.

Finalmente, é importante realçar o facto de, nesta arquitectura, ser possível expandir a implementação da estação de monitorização para várias estações, com possíveis comunicações de estado entre elas. Isto pode ser necessário em cenários onde a quantidade de informação processada pelas ferramentas torna-se muito grande, obrigando a que a partilha dos mesmos recursos do sistema operativo não seja mais possível. Este cenário não tem obrigatoriamente de se verificar apenas em relação às duas ferramentas. Pode acontecer por exemplo, que apenas o Nagios esteja sobrecarregado, e que se torne necessária a utilização de um processo de monitorização conhecido por Monitorização Distribuída retratada na figura 3.2 abaixo indicada.

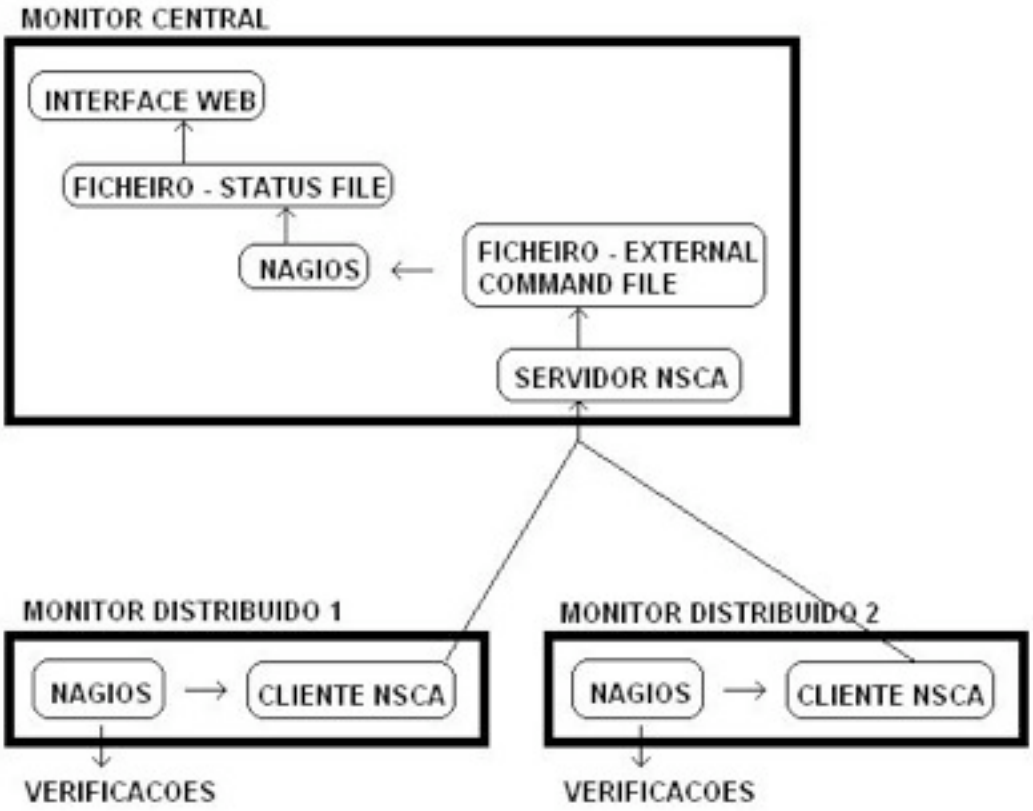

Figura 3.2 - Monitorização Distribuída

Este processo corresponde à divisão das verificações feitas pelo monitor em vários monitores, denominados monitores distribuídos. Por sua vez, estes serão monitorados passivamente por um único monitor, o central. A instalação dos monitores distribuídos não deverá incluir a interface gráfica, nem a configuração de *event handlers,* nem de notificações. A comunicação entre estes e o monitor central é feita através de um *addon* do Nagios, o NSCA. Este *addon* divide-se em duas partes, o cliente e o servidor. O cliente NSCA será executado sempre que o Nagios no monitor distribuído obtiver resultados de verificações. Essa execução resultará no envio do estado, do serviço/*host* monitorado pelo monitor distribuído, ao monitor central. Essa

informação será recebida pelo servidor NSCA localizado no monitor central que, após recepção, coloca-la-á no ficheiro de comandos externos *external command file* do Nagios. Esse ficheiro é periodicamente lido, pelo Nagios, a fim de actualizar a sua informação que, posteriormente, será consultada através da interface gráfica.

Em relação à sobrecarga do Cacti, este, apesar de não suportar o processo de monitorização distribuída, pode ser facilmente instalada noutras estações, partilhando-se as verificações pelas várias estações. A sobrecarga, no caso do Cacti, tipicamente resulta do facto das respostas ao *poller* demorarem demasiado tempo. O tempo de resposta ao *poller*, da informação necessária a todos os gráficos, é directamente proporcional ao número de dispositivos monitorados. Porém, a divisão do Cacti em várias instalações, sobre várias estações, e consequente aumento da complexidade da monitorização, só deve ser feita após se esgotarem as duas opções de optimização. A primeira consiste em aumentar o número máximo de processos concorrentes que se encontram a executar o *poller*. Isto permitirá maior concorrência, o que deverá melhorar muito o tempo de resposta ao *poller*. A segunda opção consiste em se utilizar outro *poller*, o Spine. Este consiste num *poller* escrito em C, ou seja, que recorre à execução de código pré-compilado e não de código interpretado, o que o torna muito mais eficiente. O Spine também tem a particularidade de utilizar *threads* POSIX, específicas de sistemas operativos baseados em Linux, tornando-se ainda mais eficiente, visto que a criação de *threads* é muito mais rápida do que a de processos no sistema operativo.

Quanto aos vários terminais, retratados na figura 3.1, estes consistem em terminais de acesso à estação de monitorização. A ideia é a de se ter uma única estação de monitorização com o servidor HTTP, neste caso o Apache, activado, de forma a que possa ser atribuída a diferentes administradores, a visualização da mesma, ou de diferentes partes da monitorização.

Na secção 3.1, é feita uma análise mais rigorosa à instalação do Nagios, incluindo conselhos a nível de segurança, e na secção 3.2 é feito o mesmo para o Cacti.

3.1 Nagios

Neste capítulo é feita a instalação do Nagios, juntamente com a interface gráfica e com um conjunto mínimo de *plugins* para que este tenha alguma funcionalidade, neste caso, as pretendidas para a realização deste projecto. Para este efeito são feitos, anteriormente, *downloads* tanto do Nagios como dos *plugins* que se encontram no apêndice deste projecto.

Começou-se pela instalação do Nagios numa estação de trabalho com o sistema operativo Fedora 8[33], de acordo com "Quickstart Installation Guide" na documentação[34]. Isto consistiu na instalação da aplicação, activação do servidor Apache, para que seja feita a monitorização em ambiente gráfico, e instalação dos plugins. Após tudo isso, procedeu-se à aprendizagem da monitorização de máquinas Linux/Unix, Windows, routers, switches e serviços de acesso público, como o HTTP, FTP, SSH, entre outros. Com isto, manusear os ficheiros de configuração do Nagios, verificar a sua integridade e reactivar o Nagios após alterações na configuração, tornaram-se hábitos que atribuíram ao autor uma muito maior familiaridade com esses mesmos ficheiros.

A estrutura do sistema de ficheiros onde o Nagios é instalado pode ser vista na figura 3.3:

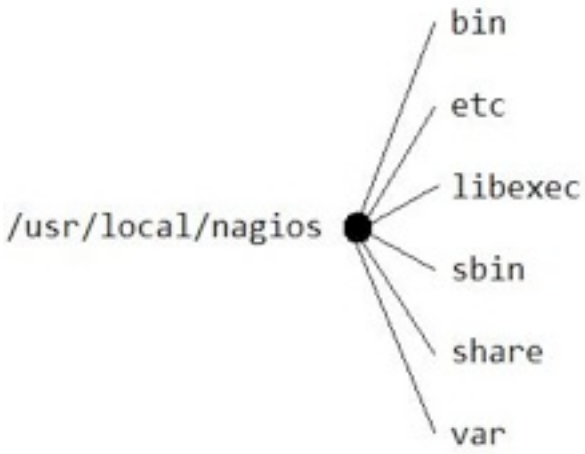

Figura 3.3 - Sistema de ficheiros na directoria raiz do Nagios

Na pasta *bin*, encontra-se o próprio Nagios, ou seja, o executável pertencente à aplicação. Encontra-se também outra aplicação Nagiostats, que apresenta estatísticas em relação à execução do Nagios, permitindo posteriormente que se optimize a *performance* do próprio. A seguir tem-se a pasta *etc* onde se encontram todos os ficheiros de configuração pertencentes ao Nagios. Nestes são, por exemplo, indicados os SLAs com os quais a aplicação deverá funcionar, bem como são definidos *hosts* e serviços. A alteração de uma qualquer configuração não terá qualquer efeito sobre a aplicação enquanto não se reiniciar o Nagios. Porém, por segurança deverá ser sempre feita a chamada "*verificação de sanidade*" para que se tenha a certeza de que as alterações à configuração respeitam a estrutura destes ficheiros:

/usr/local/nagios/bin nagios -v /usr/local/nagios/etc/nagios.cfg

Após essa verificação, dever-se-á dar procedimento ao reinicio do Nagios:

sudo /sbin/service nagios restart

O Nagios, ao contrário da maioria das ferramentas de monitorização, não contém quaisquer mecanismos internos para verificação de estado de serviços que se desejem monitorar. Ao invés disso, acrescenta uma camada de abstracção, os *plugins*, que consistem em executáveis compilados, ou scripts, que são livres de verificar o estado de um qualquer serviço, retornando depois um valor cujo formato é estipulado como um protocolo de comunicação entre os *plugins* e o próprio Nagios, definido na documentação. Estes *plugins* encontram-se na pasta *libexec*.

Em relação às restantes pastas, são as menos utilizadas, onde no *sbin* se armazenam os ficheiros cgi que permitem a parametrização remota do Nagios através da interface Web, a pasta *share* onde se encontram as páginas web da já referida interface, e a pasta *var* onde estão os ficheiros de *log*.

A recolha, de parâmetros aplicacionais e do sistema, é feita pelos *plugins*, situados na pasta *libexec*, de onde se realçam os seguintes:

Plugins	**Descrição**
check_load	percentagem de CPU em utilização
check_swap	percentagem de memória swap utilizada

Plugins	**Descrição**
check_procs	número de processos em execução no sistema
check_disk	percentagem do disco utilizado
check_file_age	tempo de vida de ficheiros
check_users	número de utilizadores activos no sistema
check_dhcp	verifica a disponibilidade do servidor DHCP
check_dns	verifica a disponibilidade do servidor DNS
check_disk_smb	verifica a disponibilidade do servidor SAMBA
check_ftp	verifica a disponibilidade do servidor FTP
check_http	verifica a disponibilidade do servidor HTTP
check_ldap	verifica a disponibilidade do servidor LDAP (autenticação)
check_nagios	verifica o estado do nagios local ou remotamente (útil em cenários distribuidos)
check_ oracle	verifica a disponibilidade do servidor de base de dados Oracle
check_ping	verifica o acesso da rede a um dispositivo remoto
check_ smtp	verifica a disponibilidade do servidor SMTP
check_pop	verifica a disponibilidade do servidor POP
check_ssh	verifica o acesso via SSH a terminais remotos
check_snmp	verifica o estado de dispositivos através do protocolo SNMP
check_jmx	verifica o estado de aplicações que se executam na máquina virtual JVM

Tabela 3.1 - Lista dos *plugins* principais

A definição de Service Level Agreements (SLAs) é feita nos ficheiros de configuração. Essa informação é definida como parâmetros, na definição dos serviços, a serem enviados, aos comandos sobre a forma de Macros. Pode-se tomar como exemplo o caso da verificação da quantidade de disco utilizada:

/usr/local/nagios/libexec/check_disk -w 20% -c 10% -p /

Neste caso está-se a especificar que: se a partição de disco, onde está montada (comando mount) a directoria root "/", tiver mais de 20% de espaço livre, o estado retornado é "OK"; se tiver menos de 20% de espaço livre e mais de 10%, o estado retorna é o de "WARNING"; se tiver menos de 10%, o estado retornado é o de "CRITICAL".

Para que se pudessem monitorar máquinas remotas, ter-se-iam que executar, nessas mesmas máquinas, os plugins de monitorização. Para tal, é necessário instalar um agente, o NRPE daemon, que tem como função executar os plugins necessários à pretendida monitorização da máquina remota, e retornar, o output resultante das suas execuções, à máquina monitora, neste caso, o Nagios. De salientar que, nestes casos, se se tiver uma firewall entre o monitor e as máquinas que se desejam monitorar, ao invés de se optar pela colocação de inúmeras regras que dêem acesso ao monitor para as diversas máquinas que se desejam monitorar, pode-se optar por inserir uma única regra que lhe dê acesso a um segundo monitor, por detrás da mesma firewall, sendo este segundo o responsável pela verificação. A isto dá-se o nome de verificações indirectas.

À comunicação entre o monitor, Nagios, e os agentes NRPE, tem de se dar especial atenção do ponto de vista de segurança. Nos casos em que se está a verificar o estado de servidores, como os de HTTP, a confidencialidade do tráfego não é importante, pois este não contém qualquer informação que se deseje manter privada. Porém, quando se requisitam resultados de plugins, executados remotamente, a agentes, esses resultados podem conter dados que não se desejem divulgar, como é o caso do número de utilizadores activos no sistema, da quantidade de CPU utilizada, ou da quantidade de memória livre pois, com essas informações, um atacante pode, por exemplo, saber quando invadir uma máquina sem que nada se note, mesmo que se tenham implementados sistemas sofisticados como *Intrusion Detection Systems*.

O NRPE suporta filtragem de pedidos, com base em endereços IP, tipicamente configurados para permitirem acesso única e exclusivamente ao(s) monitor(es). É também possível enviarem-se argumentos que sirvam de parâmetros aos comandos que se queiram executar. A utilização dessa funcionalidade consiste num risco, pelo que a sua utilização deverá ser explicitamente configurada. Um exemplo de manipulação dos argumentos prejudicial à monitorização, consiste em se alterarem os argumentos de warning e de critical de forma a que não sejam dadas indicações de situações de alerta. A execução do daemon NRPE está restrita a um utilizador e grupo, de forma a restringir o nível de permissões desta aplicação e os danos que este poderá causar em, por exemplo, casos de inserção de argumentos mal-intencionados. Para maior segurança é também suportada a cifra dos pacotes utilizando SSL/TLS, fornecendo-se assim, não só a confidencialidade desejada como também, integridade e autenticidade. Isto é feito através da utilização de rotinas de cifra AES-256, SHA e Anon-DH. Com o Anon-DH, é estabelecida uma conexão SSL/TLS sem o uso de chaves pré-geradas ou certificados. A informação relativa à chave utilizada para criar chaves SSL/TLS dinamicamente encontra-se no ficheiro dh.h, cujo conteúdo deve-se alterar por informação de uma chave, gerada pelo administrador, de 512 bits ou mais, conforme o nível de segurança assim o justificar.

Quanto aos tipos de verificações de estado de serviços ou de terminais, existem dois: activo e passivo. Nas verificações activas, da qual fazem parte as indirectas, o Nagios, é quem inicia a verificação, enquanto que nas verificações passivas, a verificação é iniciada por um processo externo. Isto é particularmente útil, no caso de verificações de serviços assíncronos, como é o caso dos SNMP traps, e nos casos em que os serviços encontram-se por detrás de firewalls, cuja rede não se tem acesso, mas que permite o acesso das máquinas monitoradas ao monitor.

Em relação à alarmística, esta é feita através de notificações. Estas consistem no alerta a um, ou conjunto de, contactos associados a um serviço ou a um host. As notificações podem ser feitas de qualquer forma, desde um simples mail, a um sms. Porém, antes dos contactos associados serem notificados, há que filtrar essas mesmas notificações. Isto permite maior flexibilidade no controlo da notificação pois, apesar destas serem sempre lançadas na ocorrência de problemas, nem sempre se desejam notificar os contactos, por exemplo, em horários pré-estabelecidos onde se sabe que um determinado servidor encontra-se desligado para manutenção/actualização. O Nagios também permite uma definição extremamente flexível no que diz respeito a períodos de tempo em que se se deve notificar um contacto, desde a exclusão de notificação nos dias de férias, alterando neste

caso o contacto para outro administrador, até à especificação das semanas alternadas em que alguém, alternadamente com outro contacto, é quem deve ser notificado.

Outra grande funcionalidade, do Nagios, consiste na monitorização redundante e à prova de falhas. A ideia básica é a de se terem dois terminais de monitorização: *master* e *slave*, onde este último nada faz enquanto o primeiro funciona. Num cenário mais simplista, pode-se facilmente implementar isto fazendo com que o *slave* monitore todos os serviços que o *master* monitora, mas com notificações desactivadas, dando a ilusão de nada estar a fazer, enquanto que simultaneamente monitora o *master* através de um serviço que execute o comando *check_nagios* e que esteja associado a um *event handler*. Caso este falhe, com o auxílio dos *event handlers*, apenas terá de activar as notificações nele próprio, ou seja, no slave. Contudo, esta abordagem apesar de funcional, não é apropriada a redes de maiores proporções, visto que vários monitores estariam a monitorar os mesmos serviços simultaneamente, consumindo largura de banda à rede. Isto pode-se facilmente resolver com a desactivação das notificações e verificações de estado de serviços, com a activação da recepção de comandos externos, e com a execução duma aplicação externa (cron) que periodicamente verifique o estado no *slave*(check_nrpe) do *master*(check_nagios) inserindo, consoante o valor de retorno, um comando na *external command* file especificado no ficheiro de configuração principal. Este comando teria como função activar ambas as notificações e as verificações dos serviços. Assim, fica-se apenas com um último problema, que é o do *slave* não ter estado inicial de tudo aquilo que monitora. Isto, por sua vez, resolve-se com a ajuda do *addon* NSCA que permite ao *master* comunicar directamente ao *slave* o estado de tudo o que monitoram.

O Nagios também permite a detecção de serviços intermitentes, ou *flap detection*. Isto é feito estudando a variação de estado do serviço nas últimos verificações feitas e, caso a percentagem de alternâncias seja maior que um valor pré-definido nos ficheiros de configuração, são notificados os respectivos contactos desse estado.

Outro aspecto importante consiste na marcação de *downtimes* para serviços que se desejem actualizar/alterar. O Nagios contém três tipos: fixos, flexíveis e triggered. Os fixos começam e terminam nos tempos especificados. Os flexíveis não têm tempos de início nem de fim, apenas a duração. Isto é útil quando sabe-se que um determinado serviço vai-se encontrar em baixo mas não se sabe exactamente quando. Os triggered são activados por um outro evento, tipicamente a falha de outro terminal ou serviço especificado. Isto é extremamente útil em casos de *downtimes* em massa.

O estado das verificações feitas no Nagios não depende única e exclusivamente do estado do serviço monitorado. Opcionalmente, pode-se fazer uma verificação depender de outra(s). A isto se chama *dependency checks*. Isto permite monitorar o estado de serviços remotos com base no estado de outros, permitindo assim melhor correlação dos resultados obtidos entre várias verificações.

Quanto às configurações no ficheiro de configuração principal e nos restantes, é importante que sejam tomadas medidas para prevenir o abuso de recursos, fornecidos pelo Nagios, por parte de terceiros. Consequentemente, não se deve executar o Nagios com o utilizador *root* por exemplo, pois este não necessita de permissões do *root* para se executar, e nunca se deve dar a uma aplicação mais permissões do que as de que necessita. Caso seja necessária a execução de aplicações/scripts por parte de, por exemplo, *event handlers* é aconselhável a utilização do comando *sudo*[38]. É igualmente importante verificar as permissões de leitura/escrita da directoria indicada, no ficheiro principal de configuração, como o *check_result_path*. Nesta são colocadas as respostas recebidas, resultantes das verificações feitas, antes de serem processadas. Acessos indevidos a esta pasta poderão resultar na falsificação de verificações, ou na sua eliminação. Outro aspecto, também a nível de segurança, é o controle de acessos ao ficheiro de comandos externos, caso esta funcionalidade esteja activada em

check_external_commands. O acesso só deverá ser permitido ao utilizador do Nagios, tipicamente *nagios*_user, e ao utilizador com o qual é executado o servidor web, tipicamente *nobody* ou *httpd*. Outra forma de controlar o Nagios é através de CGIs, normalmente inserindo-se comandos externos no ficheiro de comandos externos, pelo que é aconselhável que se faça autenticação antes de se ter acesso a esses CGIs. Esses mesmos CGIs têm acesso, tanto ao ficheiro principal de configuração, como às restantes configurações na pasta *etc* (figura 3.3). Por essa razão, não se devem armazenar *usernames* e *passwords* nesses ficheiros. O correcto é utilizar macros $USERn$ que são definidos no ficheiro /etc/resource.cfg. O Nagios dá a garantia de que os CGIs não tentarão ler desse ficheiro, pelo que pode-se restringir o acesso a este dando-lhe permissões 600 ou 660, ou seja, de leitura apenas ao dono e aos utilizadores pertencentes ao mesmo grupo do dono.

3.2 Cacti

O Cacti utiliza um conjunto de aplicações para fornecer, ao utilizador, uma interface intuitiva através da apresentação gráfica, da *performance* de dispositivos remotos, ou mesmo locais. Para testar essa apresentação, procedeu-se à instalação das seguintes componentes necessárias à execução do Cacti:

- net-snmp;
- RRDTool;
- Apache server (httpd);
- MySQL (server);
- PHP.

De salientar que, após instalação do MySQL, é necessária a criação de uma base de dados atribuindo-lhe um username e uma password, que mais tarde deverão ser especificadas num ficheiro de configuração do Cacti (/var/www/htm/cacti/include/config.php). Quanto ao seu funcionamento, o Cacti pode ser dividido em três partes distintas:

Figura 3.4 - Estrutura do funcionamento do Cacti

3.2.1 Obtenção de Dados

A primeira fase consiste na obtenção dos dados referentes ao dispositivo remoto, que servirão posteriormente para o preenchimento dos gráficos na interface gráfica. Nesta fase o Cacti utilizará o seu *poller*, que consiste num *script* PHP que, com o auxílio do pacote de aplicações *net-snmp* previamente instalado, recolherá periodicamente informação necessária a futuras apresentações gráficas. Essa recolha só é periódica com o auxílio do sistema operativo, que com o um *scheduler* deverá executar esse *poller* periodicamente. No caso do Linux, será utilizado o *crontab*[37]. Para requisitar dados de dispositivos remotos o Cacti utilizará o SNMP que, sendo considerado um padrão na indústria de monitorização, é amplamente utilizado, o que consequentemente permite ao Cacti monitorar a maior parte destes dispositivos.

Em relação ao *poller*, por ser um *script* PHP denominado cmd.php, a sua eficiência só não é um problema em ambientes relativamente pequenos. Porém, quando a eficiência se torna realmente num problema, é dada uma outra opção: o *poller* cactid, mais conhecido por *Spine*[29]. Este é escrito em C, tornando-se muito mais eficiente, tira partido das *threads* POSIX, e está ligado directamente à biblioteca net-snmp, evitando a perda de tempo na comunicação entre o script e esse pacote de aplicações SNMP.

3.2.2 Armazenamento de Dados

Após a recolha de dados, torna-se necessário o seu armazenamento. Nesta fase têm-se várias alternativas como, por exemplo, armazenarem-se os dados em ficheiros ou em bases de dados. Porém, o Cacti utiliza o RRDTool (*Round Robin Database Tool*). Esta utilização justifica-se pelo facto desta ferramenta ser capaz de recolher dados temporais, ou seja, que variam com o tempo, compactando-os e gerando imagens gráficas desses dados. Porém, a quantidade teoricamente infinita, de dados a recolher, não consiste num problema para o RRDTool, ao contrário das outras opções anteriormente mencionadas, pois esta ferramenta consegue compactar os dados recolhidos de forma a que não se expandam com o tempo, ou seja, consegue-se impor um limite no tamanho que os dados recolhidos ocuparão. Isto é possível porque o RRDTool tem a capacidade de agrupar dados num determinado espaço de tempo e calcular um valor que os represente, podendo-se configurá-lo como sendo uma média, um máximo, um mínimo, ou o último desse agrupamento de valores.

3.2.3 Apresentação de Dados

A apresentação de dados é feita numa página html, utilizando PHP. O código script PHP, nela presente, vai buscar os dados necessários à apresentação onde estes se encontram guardados, ou seja, na base de dados MySQL.

Essa página foi devidamente instalada na directoria /var/www/html/cacti. Activando-se em seguida o servidor Apache (sudo /sbin/service httpd start) tem-se facilmente acesso à página através do URL http://127.0.0.1/cacti. Ao aceder a esta página, será requisitado ao utilizador um *username* e uma *password* (admin/admin) sendo, logo em seguida, pedido que se altere a *password*.

A página encontra-se dividida em vistas: a consola, onde se podem criar a definição dos dispositivos a serem monitorizados, e onde se podem criar os respectivos gráficos e personalizá-los; e a vista de gráficos, onde se podem visualizar todos os gráficos activos.

A criação de um gráfico no Cacti, pressupõe a criação de um dispositivo. Um dispositivo consiste na representação virtual do dispositivo de rede ao qual se deseja associar o gráfico. Este, na prática, consiste num conjunto de dados referentes ao próprio dispositivo, como o *hostname* (ou IP), o tipo de dispositivo (router, máquina Linux/Windows, servidor Netware), o porto e o protocolo a utilizar para detectar se o dispositivo se encontra em baixo, e a versão SNMP a utilizar, incluindo os parâmetros de segurança, caso se escolha o SNMPv3. Após a criação do dispositivo, pode-se proceder à criação do gráfico, onde será feita a associação entre este e o dispositivo previamente criado.

3.3 Java Message Service - JMS

A correcta integração, do sistema de monitorização deste projecto, depende de uma API muito importante, o Java Message Service[35], que permitirá, ao sistema em causa, comunicar com aquele onde se integrará, através do envio de notificações no formato JMS.

Ao método de comunicação entre componentes de software, ou aplicações, dá-se o nome de *messaging*. Este método consiste num sistema *peer-to-peer*, ou seja, onde os clientes comunicam directamente entre si, sem que seja necessário o convencional sistema baseado em servidores como "pontos de encontro" para comunicar. Para utilizar esse método existe uma *Application Programming Interface* (API) desenvolvida pela Sun[36], denominada JMS API. Esta API minimiza o conjunto de conceitos que o programador necessita de saber acerca do processo de comunicação, sem tirar o suporte a funcionalidades avançadas dessa mesma comunicação.

3.3.1 Arquitectura

Uma aplicação JMS é composta por: JMS *provider*, que consiste num sistema de *messaging* que implementa as interfaces JMS e que disponibiliza funcionalidades de controle e de administração; clientes JMS, que consistem em aplicações JAVA que produzem e consomem mensagens; Mensagens, que são os objectos que contêm a informação a ser comunicada entre clientes JMS; e Objectos administrados situados no *provider*, que são objectos JMS pré-configurados para a utilização por parte de clientes, tipicamente os destinos das mensagens, parâmetros das ligações, e parâmetros de segurança.

Existem, nesta arquitectura, três domínios de comunicação. O primeiro consiste no *Point-to-Point Messaging Domain*. Uma aplicação baseada neste domínio é construída com base em três conceitos: produtor, consumidor e fila de mensagens, onde o produtor tem apenas um consumidor, não existe dependência temporal entre ambos, ou seja, para receber a mensagem o consumidor não tem que se encontrar em execução aquando do envio desta, e o consumidor envia um *acknowledge* ao produtor quando receber essa mensagem. O segundo consiste no *Publish/Subscribe Messaging Domain*, onde a mensagem é colocada à disposição, na forma de um tópico, podendo todos os consumidores que nela estejam inscritos (*subscribers*) consumir essa mensagem. O tópico só é eliminado após ter sido consumido por todos os *subscribers*. Neste domínio existe uma dependência temporal, o que significa que o consumidor tem de estar registado (*subscriber*) no tópico no momento em que este é publicado para que o possa consumir. A terceira forma de enviar uma mensagem consiste na utilização de *Common Interfaces*. Com estes pode-se utilizar o mesmo código para enviar ou receber mensagens independentemente de se estar no domínio PTP ou pub/sub. Isto torna o código mais flexível e reutilizável. A distinção entre que domínio se encontra a ser realmente utilizado é feita através da análise ao destino, sendo este uma *queue*, ou um tópico. Em relação ao consumo de mensagens, esta pode ser feita síncrona ou assincronamente. O consumo síncrono é feito através da chamada ao método *receive* que pode se bloquear à espera de uma mensagem, retornando assim que a receba, ou pode expirar por *timeout*. Quanto ao consumo assíncrono, este é feito utilizando *message listeners*. Estes consistem em classes que implementam a interface *MessageListener*, que contém um método *onMessage* que é chamado quando o provider recebe uma mensagem, recebendo como parâmetro a mensagem em causa. Estes *listeners* são anexados a consumidores que, quando recebem mensagens, chamam os respectivos *onMessage*'s dos *listeners*.

3.3.2 Integração com Nagios: Event Handlers

Antes de se perceber como funcionam os *event handlers* do Nagios, é importante entender-se o conceito de "tipo de estado" de *hosts*/serviços. Existem dois tipos de estado no Nagios: SOFT e HARD. O entendimento destes estados é crucial para que se saiba quando o Nagios irá lançar *event handlers* e notificações.

Para que o Nagios se previna do envio de falsos alarmes, este permite que o administrador indique o número máximo de tentativas a serem feitas antes que se decida que um *host*/serviço se

encontra em baixo. Este número é especificado pela opção *max_check_attempts* dentro da definição de serviços/*hosts* nos respectivos ficheiros de configuração.

Um estado SOFT ocorre nas seguintes situações:

1. Quando um *host* passa a um estado diferente de UP ou, no caso de um serviço, diferente de OK, e o número *max_check_attempts* ainda não tenha sido atingido.
2. Quando um *host*/serviço recupera de um estado SOFT.

A passagem ao estado SOFT pode ser inserido no *log file*, caso esteja dada essa indicação no ficheiro principal de configuração, e essa mesma passagem promove a execução de *event handlers*.

Um estado HARD ocorre quando:

1. Quando um *host* passa a um estado diferente de UP ou, no caso de um serviço, diferente de OK, e o número *max_check_attempts* tenha sido atingido.
2. Quando um serviço, que se encontre no estado HARD, passa de um estado diferente de OK para outro também diferente de OK (exemplo: de CRITICAL para WARNING ou vice-versa).
3. Quando um *host*, que se encontre no estado HARD, passa de um estado diferente de UP para outro também diferente de UP (exemplo: DOWN para UNREACHABLE).
4. Quando a verificação de um serviço resulta num estado diferente de OK, e o seu *host* correspondente se encontra DOWN ou UNREACHABLE.

Voltando finalmente aos *event handlers*, estes são *scripts* ou executáveis que são executados quando há alteração de estado (SOFT para HARD e vice-versa) de um *host*/serviço. Estes são muito úteis nos seguintes casos:

1. na prevenção de que o estado de um *host*/serviço passe de SOFT a HARD, onde administradores serão notificados, por exemplo, reiniciando o serviço remoto;
2. na inserção de avisos em sistemas para isso preparados;
3. para se fazer *logging* de informação relativa ao erro numa base de dados.

Os *event handlers* são executados: quando o serviço/*host* se encontra no estado SOFT; quando o estado faz a transição para HARD; e quando há recuperação do estado SOFT ou do estado HARD. Existem quatro tipos de *event handlers*. Os dois primeiros são globais, sendo um para serviços e outros para *hosts*. Estes são executados para todos os serviços e *hosts* respectivamente, e são indicados no ficheiro de configuração principal nas opções *global_host_event_handler* e *global_service_event_handler*. Contudo, é também possível a indicação específica a cada *host* ou serviço, através da directiva *event_handler* nas respectivas definições.

Para que a execução dos *event handlers* seja possível, é necessária a activação dessa funcionalidade. Isso pode ser feito activando-a para toda a aplicação, com a ajuda da opção *enable_event_handlers* no ficheiro de configuração principal, mas também pode ser feito a nível de cada serviço/*host*, através da directiva *event_handler_enabled* na definição do *host*/serviço. Apenas de salientar que se a activação não tiver sido feita a nível de toda a aplicação, a activação a nível de cada *host*/serviço não surtirá efeito. Em relação à ordem de execução, os *event handlers* globais são executados antes dos específicos e, nos casos em que o estado é HARD e consequentemente serão enviadas notificações, os *event handlers* serão enviados logo a seguir a essas notificações.

Um exemplo dum serviço com um *event handler* associado é o seguinte:

```
define service {
        hostname                um_host_qualquer
        service_description     HTTP
        max_check_attempts      4
        event_handler           JMSProducer
        ...
}
```

Código 3.1 - Definição do serviço e do respectivo *event handler*

O JMSProducer consiste num comando que é definido em seguida:

```
define command {
        command_name            JMSProducer
        command_line            /usr/local/nagios/libexec/eventhanders/JMSProducer
$SERVICESTATE$ $SERVICESTATETYPE$ $SERVICEATTEMPT$
}
```

Código 3.2 - Definição do *event hadler* a ser executado pelo serviço defnido em 3.1

Neste comando, mais propriamente na directiva *command_line*, é indicado o *script/* executável a ser executado e são enviados os parâmetros necessários que deverão constar da mensagem enviada. É exactamente este o ponto de integração entre o JMS e o Nagios. No *script* indicado na directiva *command_line* deverá estar uma linha de código que invoque, na máquina virtual java, o ".class" correspondente ao produtor da mensagem. Esta mensagem deverá ser enviada ao cliente JMS, neste caso um consumidor, resultando assim no cumprimento de mais um objectivo: a correcta integração com um *trouble ticket system* através do JMS.

4 Testes

Neste capítulo, é dado um caso de estudo, onde aplicar-se-á um caso prático simples que atinja os objectivos inicialmente designados. Este caso de estudo, demonstrado na figura 4.1, visa focar os seguintes aspectos:

- monitorização de parâmetros de sistema
- monitorização de parâmetros aplicacionais com JMX
- monitorização de servidores: DHCP, HTTP, FTP, SSH
- SNMP

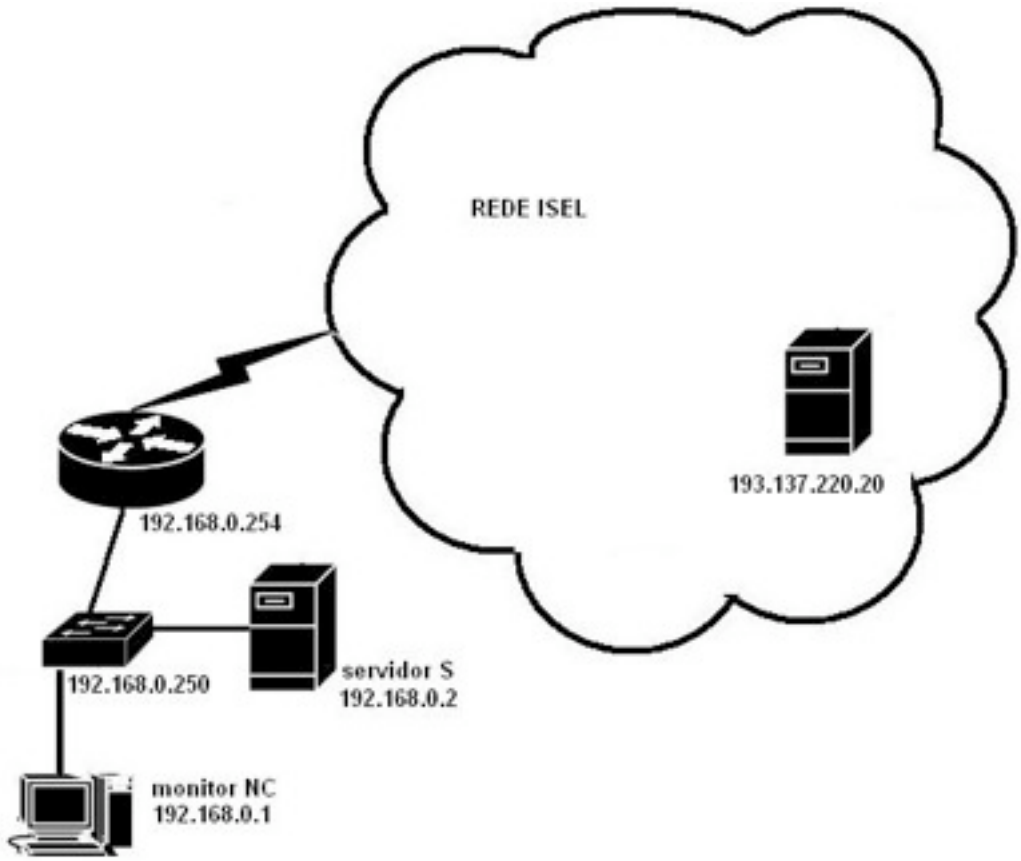

Figura 4.1 - Cenário a estudar

A estação de monitorização NC (Nagios-Cacti) procederá à monitorização do servidor S (HTTP, FTP, SSH), do *router* .254 e do *switch* .250 (SNMP), de um dos servidores DNS do ISEL 193.137.220.20 e, finalmente, monitorará um agente JMX também situado no servidor S.

Para se montar este cenário de testes foi necessário fazerem-se as seguintes configurações respectivamente para o *router* e para o *switch*:

```
// MODO INTERFACE
interface fastethernet 0
ip address dhcp
ip nat outside
no shutdown
```

```
// MODO INTERFACE
interface ethernet 0
ip address 192.168.0.254 255.255.255.0
ip nat inside
no shutdown

//NAT
access-list 1 permit 192.168.0.0 0.0.0.255
ip nat inside source list 1 interface fastEthernet 0 overload

//SNMP
snmp-server community mycom view mycom_view ro
snmp-server view mycom_view 1.3.6.1.2.1.1.3 included //sysUpTime
snmp-server view mycom_view 1.3.6.1.2.1.2 included // interfaces

//REMOTE
enable secret cisco
line vty 0 4
password 0 cisco
login
exit
```

Código 4.1 - Configuração do *router*

```
// atribuicao de IP
interface vlan1
ip address 192.168.0.250 255.255.255.0
no shutdown
exit

//REMOTE
enable secret cisco
line vty 0 4
password 0 cisco
login
exit

//SNMP
snmp-server community mycom view mycom_view ro
snmp-server view mycom_view 1.3.6.1.2.1.1.3 included
```

Código 4.2 - Configuração do *switch*

De salientar a não configuração de um servidor DHCP no *router*. Esta opção foi feita pelo facto de, os computadores existentes na rede 192.168.0.0/24, a estação de monitorização e o servidor, serem exemplos de estações que tipicamente possuem IP estático, devido aos serviços que fornecem. Outra opção seria a de configurar a atribuição de IPs de acordo com os endereços MAC, ou Client ID mas, por simplicidade, optou-se por não implementá-la, dado o ambiente de testes ser relativamente pequeno.

4.1 JMX

Para implementar o JMX, criou-se um cenário onde, na máquina onde se pretende monitorar um determinado recurso, neste caso as propriedades de uma classe, criou-se um JMX agent que, através de JMX connectors, deverá comunicar com o terminal de monitorização, onde será utilizado o *plugin* check_jmx que comunicará então remotamente com o agente. Para se proceder com a implementação do agente, teve-se de criar um MBean (*standard MBean*). Este é

definido por uma interface (TestMBean.java) e uma classe que implementa essa interface (Test.java). A classe contém métodos e propriedades (*getters* e *setters*) que posteriormente serão executados por ordem de aplicações remotas:

```
package com.ALproject;

public interface TestMBean{
	public void sayHello();
	public int add(int x,int y);
	public String getName();
	public int getCacheSize();
	public void setCacheSize(int x);
}
```

```
package com.ALproject;

import javax.management.*;

public class Test extends NotificationBroadcasterSupport implements TestMBean {
	public void sayHello() {
		System.out.println("Project: testing hello");
	}
	public int add(int x, int y) {
		return x+y;
	}
	public String getName() {return name;}
	public synchronized int getCacheSize() {return cacheSize;}
	public synchronized void setCacheSize(int x) {
		int oldCacheSize = cacheSize;
		cacheSize = x;
		//confirmation:
		System.out.println("new cache size: " + cacheSize);
		Notification n = new AttributeChangeNotification(this,sequenceNumber+
+,System.currentTimeMillis(),"Cache Size Changed","CacheSize","int",oldCacheSize,cacheSize);
		sendNotification(n);
	}
	public MBeanNotificationInfo[] getNotificationInfo() {
		String[] types = new String[] {AttributeChangeNotification.ATTRIBUTE_CHANGE};
		String name = AttributeChangeNotification.class.getName();
		String description = "An attribute of this MBean has changed";
		MBeanNotificationInfo info = new MBeanNotificationInfo(types,name,description);
		return new MBeanNotificationInfo[] {info};
	}
	private final String name = "Andre Lima";
	private int cacheSize = 500;
	private int sequenceNumber = 0;
}
```

Código 4.3 - Código responsável pela criaçõ do MBean

Em seguida, foi criado o JMX agent cuja componente nuclear consiste no MBean server:

```
package com.ALproject;

import java.lang.management.*;
import javax.management.*;

public class MainPG {
        public static void main(String[] args) throws Exception {
                MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
                ObjectName name = new ObjectName("com.ALproject:type=Test");
                Test mbean = new Test();
                mbs.registerMBean(mbean,name);
                System.out.println("waiting forever...");
                Thread.sleep(Long.MAX_VALUE);
        }
}
```

Código 4.4 - Agente JMX

O JMX *agent* MainPG começa por requisitar um *MBean* server da plataforma. Caso um já exista, é devolvido, caso contrário, é criado e igualmente retornado. Em seguida é definido um nome único, que tem como objectivo identificar o *MBean* criado na linha seguinte. A seguir, o *MBean* é registado com o nome definido pelo *ObjectName*, no *MBean* server, e a aplicação fica à espera indefinidamente. Esta classe será finalmente compilada, juntamente com a classe correspondente ao mbean e à interface, da seguinte forma:

```
java -Dcom.sun.management.jmxremote.port=9999 -Dcom.sun.management.jmxremote.authenticate=false
        -Dcom.sun.management.jmxremote.ssl=false com.ALproject.MainPG
```

Desta forma pode-se atribuir explicitamente um porto TCP onde o JMX agent deverá ficar à escuta. Em relação à autenticação e utilização do protocolo SSL, estes são desactivados pois o *plugin* utilizado para a monitorização não os suporta.

A monitorização neste caso será feita pelo plugin *check_jmx*:

```
check_jmx -U service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi -O com.ALproject:type=Test -A CacheSize -w 400 -c 600
```

Neste comando: indica-se o URL onde se encontra o agente, URL esse que obedece a uma sintaxe específica; indica-se o nome do *MBean*, tal como indicado no *ObjectName* definido no JMX *agent*; é indicado o nome da propriedade que se deseja inspeccionar, neste caso o *CacheSize*; e são indicados os valores limite para correcta indicação dos estados *warning* e *critical*.

4.2 JMS - Modelo de Programação e Implementação

No desenvolvimento de uma aplicação JMS, encontram-se cinco conceitos básicos:

1. Objectos administrados
2. Conexões/Ligações
3. Sessões
4. Produtores de mensagens
5. Consumidores de mensagens
6. Mensagens

Os objectos administrados podem ser divididos em duas partes: destinos e fábrica de conexões, que são melhor geridos administrativamente do que programaticamente. Isto porque é de se esperar que a tecnologia de *messaging* na qual os *providers* se baseiam seja diferente, e

também que tenham parâmetros diferentes, como por exemplo de segurança. Por isso tudo, e por se pretender que as aplicações cliente sejam portáveis, estes objectos são geridos administrativamente. As fábricas de conexões são objectos que o cliente utiliza para criar conexões com o *provider*. Uma fábrica encapsula um conjunto de configurações, a utilizar nas conexões por ela gerada, configuradas pelo administrador do *provider*. O destinos são objectos que o cliente utiliza para especificar quem será o consumidor da mensagem. No caso de comunicações PTP, estes destinos chamam-se de *queues*, e no caso de sub/pub, são chamados de tópicos.

Uma conexão JMS representa uma ligação virtual entre o cliente e o *provider*. Na prática, pode ser um um *socket* sobre TCP/IP entre ambos. Numa conexão são criadas uma ou mais sessões.

Em relação às sessões, é através delas que se criam os produtores, os consumidores e as mensagens. Uma sessão consegue enviar um conjunto de mensagens ou recebê-las, utilizando o conceito de transacções, ou seja, garantindo a atomicidade no envio/recepção desse conjunto. É igualmente possível, numa sessão, efectuarem-se várias transacções, terminando cada uma delas com a ajuda do método *commit*.

Os consumidores e produtores são objectos JMS criados apartir de uma sessão, que recebem como parâmetro o destino do qual deverão receber/enviar mensagens. Esse destino pode ser uma *queue* ou um tópico.

As mensagens consistem no base fundamental de toda esta API, pois é a sua construção, envio e recepção que motivam toda a sua existência. Estas mensagens encontram-se divididas em três partes: cabeçalho, propriedades e corpo. O cabeçalho contém campos pré-definidos, que contêm informação necessária à identificação e encaminhamento da mensagem. As propriedades consistem em campos adicionais aos do cabeçalho, caso sejam necessários por exemplo na utilização de *selectors* (filtros de mensagens). O corpo da mensagem pode ser de cinco tipos diferentes: *TextMessage*(String), *MapMessage*(conjunto par-valor), *ByteMessage*(stream de bytes não interpretados), *StreamMessage*(stream de tipos primitivos), *ObjectMessage*(objecto serializável), e *Message*(corpo vazio).

Para testar esta API foram desenvolvidas as seguintes classes:

- Producer.java: esta classe recebe, na linha de comandos, dois parâmetros, onde a primeira indica qual o destino ("queue" ou "topic") e a segunda o número de mensagens a enviar. Caso o segundo parâmetro não seja indicado, assume-se o valor 1. Após se criar o destino e se validarem os parâmetros, é ciada uma conexão, uma sessão não transaccional apartir dessa conexão, é criado um produtor e uma mensagem, e finalmente são enviadas as mensagens antes de se fechar a conexão. De salientar que, após o envio do número desejado de mensagens, é enviada mais uma do tipo "*Message*" (que não contém corpo - vazia) o que indica o término da comunicação.

```
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.Queue;
import javax.jms.Topic;
import javax.jms.Connection;
import javax.jms.Session;
import javax.jms.MessageProducer;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.annotation.Resource;

public class Producer {
    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;
```

```
    public static void main(String[] args) {
        final int NUM_MSGS;
        Connection connection = null;

        if ((args.length < 1) || (args.length > 2)) {
            System.err.println(
                    "Programa recebe um ou dois parâmetros: "
                    + "<tipo_destino> [<número de mensagens>]");
            System.exit(1);
        }
        String destType = args[0];
        System.out.println("O destino é do tipo " + destType);

        if (!(destType.equals("queue") || destType.equals("topic"))) {
            System.err.println("O 1º parâmetro deve ser \"queue\" ou " + "\"topic\"");
            System.exit(1);
        }

        if (args.length == 2) {
            NUM_MSGS = (new Integer(args[1])).intValue();
        } else {
            NUM_MSGS = 1;
        }

        Destination dest = null;

        try {
            if (destType.equals("queue")) {
                dest = (Destination) queue;
            } else {
                dest = (Destination) topic;
            }
        } catch (Exception e) {
            System.err.println("Erro ao estabelecer o destino: " + e.toString());
            e.printStackTrace();
            System.exit(1);
        }

        try {
            connection = connectionFactory.createConnection();

            Session session = connection.createSession(
                    false,
                    Session.AUTO_ACKNOWLEDGE);

            MessageProducer producer = session.createProducer(dest);
            TextMessage message = session.createTextMessage();

            for (int i = 0; i < NUM_MSGS; i++) {
                message.setText("Esta é a mensagem " + (i + 1));
                System.out.println("Enviando mensagem: " + message.getText());
                producer.send(message);
            }
            producer.send(session.createMessage());
        } catch (JMSException e) {
            System.err.println("Exception occurred: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                }
            }
        }
    }
}
```

Código 4.5 - Produtor

-AsynchConsumer.java: esta classe consiste apenas num método *main* que recebe mensagens enquanto o utilizador não premir a tecla 'q' ou 'Q'. As mensagens são recebidas de um *queue* ou de um *topic* consoante a indicação dada na linha de comandos. Antes de entrar num ciclo, onde eternamente fica à espera que se prima a tecla que terminará a aplicação, é lançado um *listener* TextListener (código apresentado a seguir ao do AsynchConsumer.java), que é associado aqui ao consumidor, e cujo método OnMessage será invocado quando o consumidor receber uma mensagem.

```
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.Queue;
import javax.jms.Topic;
import javax.jms.Connection;
import javax.jms.Session;
import javax.jms.MessageConsumer;
import javax.jms.TextMessage;
import javax.jms.JMSException;
import javax.annotation.Resource;
import java.io.InputStreamReader;
import java.io.IOException;

public class AsynchConsumer {
    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;

    public static void main(String[] args) {
        String destType = null;
        Connection connection = null;
        Session session = null;
        Destination dest = null;
        MessageConsumer consumer = null;
        TextListener listener = null;
        TextMessage message = null;
        InputStreamReader inputStreamReader = null;
        char answer = '\0';

        if (args.length != 1) {
            System.err.println("O programa recebe um único parâmetro: <tipo_destino>");
            System.exit(1);
        }

        destType = args[0];
        System.out.println("Tipo de destino é " + destType);

        if (!(destType.equals("queue") || destType.equals("topic"))) {
            System.err.println("Parâmetro tem de ser \"queue\" ou \"topic\"");
            System.exit(1);
        }

        try {
            if (destType.equals("queue")) {
                dest = (Destination) queue;
            } else {
                dest = (Destination) topic;
            }
        } catch (Exception e) {
            System.err.println("Error setting destination: " + e.toString());
            e.printStackTrace();
            System.exit(1);
        }
```

```
        try {
            connection = connectionFactory.createConnection();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            consumer = session.createConsumer(dest);
            listener = new TextListener();
            consumer.setMessageListener(listener);
            connection.start();
            System.out.println(
                    "Para terminar programa prima Q or q, " + "e depois <enter>");
            inputStreamReader = new InputStreamReader(System.in);

            while (!((answer == 'q') || (answer == 'Q'))) {
                try {
                    answer = (char) inputStreamReader.read();
                } catch (IOException e) {
                    System.err.println("I/O exception: " + e.toString());
                }
            }
        } catch (JMSException e) {
            System.err.println("Ocorrência de excepção: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                }
            }
        }
    }
}
```

```
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.JMSException;

public class TextListener implements MessageListener {

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Reading message: " + msg.getText());
            } else {
                System.err.println("Message is not a TextMessage");
            }
        } catch (JMSException e) {
            System.err.println("JMSException in onMessage(): " + e.toString());
        } catch (Throwable t) {
            System.err.println("Exception in onMessage():" + t.getMessage());
        }
    }
}
```

Código 4.6 - Consumidor

Neste cenário de teste para JMS, é necessária a integração do código aqui descrito, neste caso o do produtor, com o Nagios. Para isso é necessária a implementação prática da solução dada na secção 3.3.2.

5 Conclusão

Este projecto resultou da necessidade de se monitorarem dispositivos remotos em grandes redes, atribuindo a todo o sistema melhor autonomia. Essa autonomia foi concedida ao sistema, através de diagnósticos automatizados utilizando, para esse efeito, o Nagios e o Cacti, e através de uma manutenção preventiva, com a existência de *event handlers* que são accionados quando existem verificações feitas, através dos diagnósticos automatizados, que retornam valores fora dos parâmetros normais, devidamente especificados nos SLAs nos ficheiros de configuração. Em relação à alarmística, que consiste num dos objectivos fundamentais do projecto, o Nagios suporta, por si só, o envio de mensagens por vários canais, como por exemplo, o e-mail. Quanto ao envio de mensagens JMS, apesar do Nagios não o suportar nativamente, foi encontrada uma solução que consistiu em aproveitar o já existente conceito de *event handlers*, e integrar no *script*, que estes são capazes de invocar, o código necessário à chamada de uma classe Java na respectiva máquina virtual, que envie a mensagem pretendida.

A primeira parte de todo este projecto, consistiu na escolha da ferramenta a utilizar para atingir todos os objectivos pretendidos e designados na introdução. Essa escolha teve como principal argumento a flexibilidade, dada ao Nagios, através da sua arquitectura baseada em *plugins* e *event handers*, flexibilidade essa essencial na posterior integração tanto do JMX, como do JMS. Na segunda parte do projecto, utilizando as ferramentas Nagios e Cacti, apresentou-se uma arquitectura de monitorização, onde se demonstra como integrar ambos os sistemas numa rede, de forma a que se atinjam os objectivos pretendidos. Também são dadas sugestões a nível de configurações e controle de acessos, de forma a que se proteja a estação de monitorização que, potencialmente, pode ser utilizada como *backdoor* para outros sistemas, devido ao amplo acesso tipicamente dado aos monitores como, por exemplo, a livre passagem por *firewalls*. Na terceira, e última parte do trabalho, é elaborado um caso de estudo que visa implementar a arquitectura proposta anteriormente, de forma a demonstrar uma solução prática que atinge os objectivos pretendidos.

Como trabalho futuro, pode-se criar um *plugin* para o Nagios, em relação ao JMX, pois o retirado para implementação neste projecto, não permite a exploração completa de todas as funcionalidades desta especificação, como a verificação de parâmetros diferentes do tipo inteiro, e também não permite a configuração de parâmetros de segurança. Em caso de implementação, este novo *plugin*, para que funcione correctamente, terá de obedecer ao protocolo de comunicação, com o Nagios, especificado no manual que se encontra no *site*[23]. Outro aspecto deste trabalho que se pode vir a trabalhar futuramente, consiste na alteração do código fonte do Cacti, para superar a sua única falha: o facto de não enviar notificações.

As respectivas instalações, estudo das configurações, da activação de serviços e servidores, e o estudo do *scheduler crontab*, vieram trazer ao autor um maior à vontade com o sistema operativo Linux. Este à vontade, juntamente com a familiarização, com a complexa linha de comandos, permitirá ao autor efectuar a instalação, de todo esse sistema, poupando ao máximo os recursos da estação de monitorização, através da não instalação da componente gráfica do sistema operativo. Nessa estação deverá ter activado o servidor Apache (*httpd*), através do qual serão feitas as monitorizações. Esta implementação é mais flexível, dado permitir verificações independentes, administrativamente atribuídas a uma ou mais pessoas, tornando a verificação/autorização da origem dos acessos num ponto importante na implementação real.

7. REFERÊNCIAS

[1] BRISA, http://www.brisa.pt/Brisa/vPT
[2] International Telecommunication Union, http://www.itu.int/net/home/index.aspx
[3] ITU-T, "TMN", Fevereiro 2000, http://www.itu.int/rec/T-REC-M.3000-200002-I/en
[4] Flextronics, "FCAPS", http://www.futsoft.com/pdf/fcapswp.pdf
[5] Dr Jeff Case, "Simple Network Management Protocol", http://www.snmp.com/
[6] IETF, "SNMPv1", Maio 1990, RFC 1155, 1156, 1157, 1213
[7] IETF, "SNMPv3", RFC 3410, 3411, 3412, 3413, 3414, 3415, 3416
[8] IETF, "User-based Security Model(USM)", http://www.ietf.org/rfc/rfc2574.txt
[9] IETF, "View-based Access Control Model", http://www.ietf.org/rfc/rfc2575.txt
[10] IETF, "SNMPv2p", RFC 1441-52
[11] IETF, "SNMPv2c" RFC 1901-08
[12] IETF, "SNMPv2u", RFC 1909, 1910
[13] IETF, "SNMP security protocol", http://tools.ietf.org/html/rfc1352
[14] Sun MicroSystems, "JMX Especification 1.4", Novembro 2006, http://java.sun.com/javase/6/docs/technotes/guides/jmx/JMX_1_4_specification.pdf
[15] Sun MicroSystems, "Java Security Model", http://java.sun.com/javase/technologies/security/
[16] Anne Anderson (Internet Security Research Group in Sun Labs), "Java Access Control Mechanisms", http://research.sun.com/techrep/2002/smli_tr-2002-108.pdf
[17] Pekka Nikander & Jonna Partanen, "Distributed Policy Management", http://www.isoc.org/isoc/conferences/ndss/99/proceedings/papers/nikander.pdf
[18] IETF, "Simple Public Key Infrastructure (SPKI)", http://world.std.com/~cme/html/spki.html
[19] Sun Microsystems, "Java Authentication and Authorization Service", http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRefGuide.html
[20] OpenNMS, http://www.opennms.org/index.php/Main_Page
[21] Snort, http://www.snort.org/
[22] Nessus, http://www.nessus.org/nessus/
[23] Nagios, http://www.nagios.org
[24] Nagios Plugins Site, http://nagiosplugins.org/
[25] Cacti, http://www.cacti.net/
[26] MySQL, http://dev.mysql.com/doc/
[27] RRDTool, http://oss.oetiker.ch/rrdtool/
[28] net-snmp, http://www.net-snmp.org/
[29] Spine, http://www.cacti.net/spine_info.php
[30] Hobbit Monitor, http://hobbitmon.sourceforge.net/
[31] Big Brother, http://www.bb4.org
[32] Monit, http://www.tildeslash.com/monit
[33] Fedora Project, http://fedoraproject.org/
[34] Ethan Galstad, "Nagios Version 3.x Documentation", http://nagios.sourceforge.net/docs/nagios-3.pdf
[35] Sun Microsystems, "Java Message Service", http://java.sun.com/products/jms/
[36] Sun - http://www.sun.com
[37] Crontab, http://www.linuxhelp.net/guides/cron/
[38] Todd Miller, Chris Jepeway, Aaron Spangler, Jeff Nieusma, Dave Hieb, "Sudo", http://www.courtesan.com/sudo/sudo.html
[39] Pararede - http://www.pararede.com/