NATASCHA PETRY LIGOCKI

UMA FERRAMENTA DE MONITORAMENTO DE REDES USANDO SISTEMAS GERENCIADORES DE STREAMS DE DADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profª Drª Carmem Satie Hara

CURITIBA

2008

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O monitoramento de redes é uma tarefa complexa que geralmente envolve a utilização de diversas ferramentas para fins específicos. Este trabalho descreve uma ferramenta flexível de monitoramento de redes, chamada *PaQueT*, projetada para atender um grande número de necessidades de monitoramento. O usuário pode definir métricas como consultas, em um processo similar ao de escrever consultas em um sistema de banco de dados. Desta forma, a ferramenta provê um mecanismo que permite adaptá-la facilmente à medida que os requisitos do sistema mudam. A *PaQueT* permite que se monitorem valores desde métricas no nível de pacotes àquelas normalmente fornecidas apenas por ferramentas baseadas em Netflow e SNMP. Além disso, a ferramenta possui uma curva de aprendizagem acentuada, permitindo que se obtenha a informação desejada rapidamente. A *PaQueT* foi desenvolvida como uma extensão do Sistema Gerenciador de *Streams* de Dados Borealis. A primeira vantagem deste modelo é a possibilidade de gerar métricas em tempo real, minimizando o volume de dados a ser armazenado; segundo, a ferramenta pode ser facilmente estendida para dar suporte a diferentes tipos de protocolos de rede. Foi conduzido um estudo experimental para verificar a efetividade da proposta e para determinar a sua capacidade de processamento de grandes volumes de dados.

ABSTRACT

Network monitoring is a complex task that generally requires the use of different tools for specific purposes. This paper describes a flexible network monitoring tool, called *PaQueT*, designed to meet a wide range of monitoring needs. The user can define metrics as queries in a process similar to writing queries on a database management system. This approach provides an easy mechanism to adapt the tool as system requirements evolve. *PaQueT* allows one to monitor values ranging from packet level metrics to those usually provided only by tools based on Netflow and SNMP. Besides, the tool has a steep knowledge curve and allows one to obtain the desired information quickly. *PaQueT* has been developed as an extension of Borealis Data *Streams* Management System. The first advantage of our approach is the ability to generate measurements in real time, minimizing the volume of data stored; second, the tool can be easily extended to consider several types of network protocols. We have conducted an experimental study to verify the effectiveness of our approach, and to determine its capacity to process large volumes of data.

CAPÍTULO 1

INTRODUÇÃO

Com a popularização da Internet, problemas nas redes de computadores são bastante freqüentes hoje em dia. As reclamações dos usuários envolvem disponibilidade da rede, acesso lento durante os horários de pico, problemas de *download* e de acessos em geral. Mesmo os administradores de rede mais experientes precisam ter uma visão do estado da rede como um todo antes de resolver tais problemas. A melhor forma de obter estas informações é através do monitoramento. Existem várias ferramentas disponíveis para este propósito. Porém, é difícil encontrar uma única ferramenta que atenda a todas as necessidades de uma empresa. Com isso é comum o cenário em que se faz necessário o uso de várias ferramentas diferentes para efetuar o monitoramento de um único sistema como mostra a Figura 1.1. Isto faz com que o usuário tenha que aprender a utilizar cada uma dessas ferramentas, além de ter que tratar com os seus respectivos resultados, os quais são gerados cada um com suas peculiaridades e em diferentes formatos. Uma das soluções mais freqüentes é o uso de *scripts* implementados especificamente para cada cenário. Mas isto nem sempre é uma tarefa fácil, além de normalmente serem de difícil reuso e portabilidade.

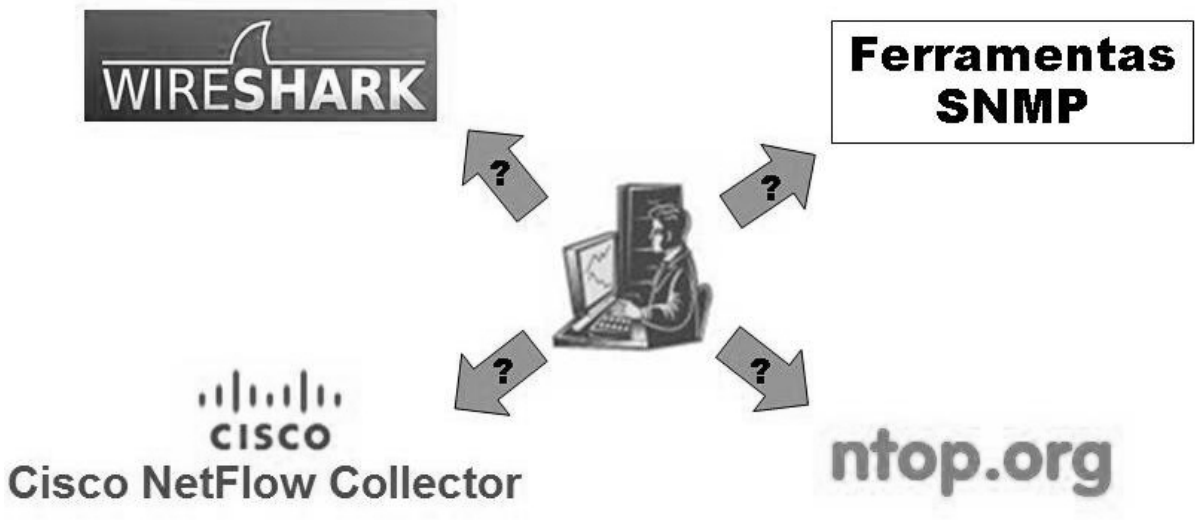

Figura 1.1: Necessidade do uso de várias ferramentas para o monitoramento de uma única rede.

Nos últimos anos, diversos Sistemas Gerenciadores de *Streams* de Dados (SGSD) foram propostos na literatura [16, 4, 7, 14, 8, 3] para prover as funcionalidades dos Sistemas Gerenciadores de Banco de Dados (SGBD) tradicionais sobre fluxos contínuos de dados. Estes fluxos de dados (*streams*) podem ser, por exemplo, os pacotes trafegando em uma rede, ou dados de uma rede de sensores, ou de um sistema de monitoramento de chamadas. A característica principal destes sistemas é o grande volume de dados, o que impossibilita que eles sejam armazenados em sua totalidade para serem processados posteriormente. Assim, os SGSDs são sistemas que, além de outras facilidades, possuem uma linguagem de alto nível para expressar consultas, que são processadas à medida que os dados fluem pelo sistema.

Este trabalho descreve o uso do SGSD Borealis [3] para implementar uma ferramenta de monitoramento de redes, chamada *Packet Query Tool* (*PaQueT*). Ao contrário de outras ferramentas existentes para este propósito [32], que possuem um elenco pré-definido de métricas que podem ser geradas, a *PaQueT* é uma ferramenta de propósito geral, que permite que o administrador de uma rede defina as consultas de acordo com as suas necessidades específicas. Para isto, a *PaQueT* captura todos os pacotes de uma rede, particiona-os de acordo com um esquema pré-definido, e direciona esta informação para o SGSD. O administrador pode então utilizar o SGSD para executar as consultas e obter as informações desejadas. Estas consultas são expressas em uma linguagem de alto nível, que possui operações semelhantes à *Structured Query Language* (SQL), a linguagem de consultas padrão dos SGBDs relacionais. Uma vantagem desta abordagem é a facilidade de reuso, o que permite que as soluções sejam facilmente modificadas a fim de aperfeiçoá-las e adaptá-las conforme a necessidade. Além disso, independente do grau de granularidade da informação desejada, do tipo de informação ou dos detalhes da rede, o usuário faz uso de uma única ferramenta. A *PaQueT* utiliza a mesma linguagem para todos os dados tratados pelo sistema, facilitando o seu uso e unificando os seus resultados como ilustra a Figura 1.2.

Contribuições. As principais contribuições deste trabalho são:

- o desenvolvimento da ferramenta *PaQueT* que além de permitir uma análise detalhada de uma rede, é customizável pelo próprio usuário do sistema sem a necessidade de interferência de um desenvolvedor;

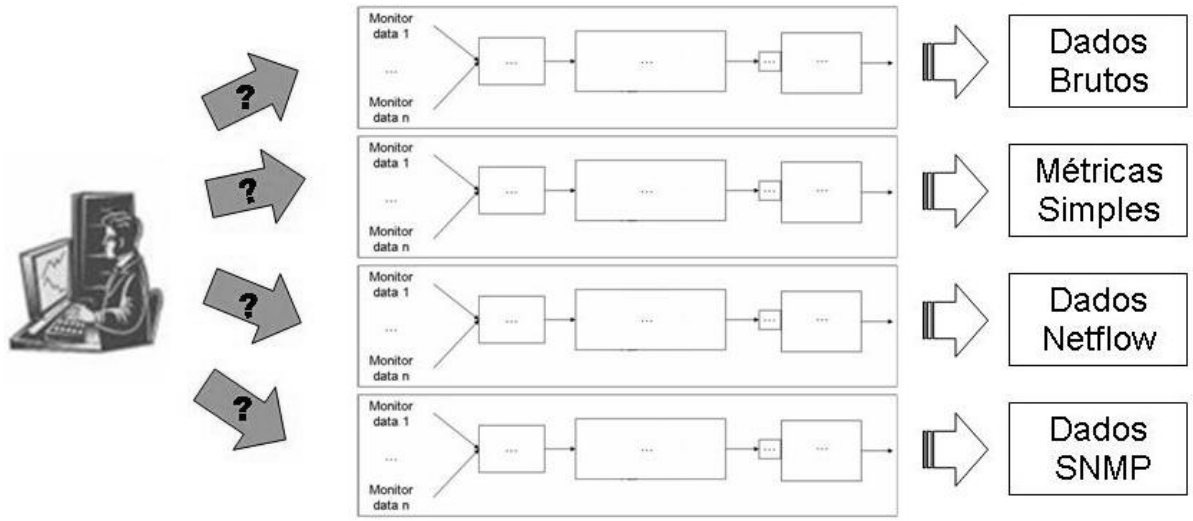

Figura 1.2: A mesma linguagem para expressar quaisquer tipos de monitoramento.

- validação da funcionalidade e desempenho da ferramenta de forma absoluta e também comparativa com resultados de trabalhos semelhantes.

Organização. O restante desta dissertação está organizada da seguinte forma: no Capítulo 2 são apresentados os SGSDs, suas aplicações e alguns trabalhos relacionados. O capítulo seguinte descreve diferentes tipos de monitoramento de redes, apontando seus respectivos objetivos e peculiaridades. O próximo capítulo mostra as características do Borealis, especificando detalhes e limitações do sistema como um todo, focando principalmente em sua arquitetura e funcionalidade. No Capítulo 5 são descritas a arquitetura e os detalhes da *PaQueT*. Também é mostrada a aplicabilidade da ferramenta no escopo de SLA. O Capítulo 6 descreve os experimentos feitos e mostra os resultados obtidos com a *PaQueT*. Neste capítulo ainda são descritas as primeiras experiências da ferramenta quando executada em um ambiente distribuído. Por fim, no Capítulo 7, são apresentados alguns trabalhos futuros, que incluem funcionalidades adicionais e melhorias para a ferramenta proposta neste documento.

CAPÍTULO 2

SISTEMAS GERENCIADORES DE STREAMS DE DADOS

Os Sistemas Gerenciadores de *Streams* de Dados (SGSD) foram propostos para prover as funcionalidades de um Sistema Gerenciador de Banco de Dados (SGBD) sobre fluxos contínuos de dados, fornecendo respostas em tempo real. Uma discussão sobre processamento de *streams* em tempo real pode ser encontrada em [35]. A principal diferença entre os SGBDs e os SGSDs consiste em como os dados e as consultas persistem no sistema [22]. O primeiro contém informação estática e consultas dinâmicas, enquanto o último tem o comportamento inverso. Ou seja, os bancos de dados tradicionais normalmente executam consultas diferentes sobre o mesmo conjunto de dados. Já os SGSDs executam as mesmas consultas sobre dados que chegam ao longo do tempo. A Tabela 2.1 mostra de forma resumida uma comparação mais abrangente dos dois tipos de sistemas.

Característica	**SGBD**	**SGSD**
Dados	Persistente	Transiente
Consulta	Transiente	Persistente
Atualização de dados	Modificação	Inserção
Resultado da consulta	Exato	Aproximado
Avaliação da consulta	Arbitrária	Única
Plano de consulta	Fixo	Adaptativo

Tabela 2.1: Quadro comparativo geral entre SGBD e SGSD.

Outra diferença é a atualização dos dados, que nos sistemas tradicionais consiste de operações para modificar, adicionar e excluir registros de tabelas já existentes. Apesar de alguns SGSDs planejarem revisões de registros, uma forma de modificar os dados de entrada, a operação mais comum e útil em SGSDs é a de alimentar os dados inserindo novas entradas.

Devido à grande quantidade de dados e por se tratar normalmente de consultas em tempo real, é possível que em momentos de pico os SGSDs possam gerar resultados não tão precisos como nos sistemas tradicionais. Neste caso, são processados o maior número de registros possível, podendo ser usados cálculos probabilísticos para tentar minimizar eventuais erros den-

tro das janelas de tempo especificadas nas consultas.

A avaliação da consulta em um sistema tradicional se faz através de acessos arbitrários aos dados, enquanto nos SGSDs, tal avaliação tem que ser feita uma única vez à medida que os dados vão chegando [22]. Já o plano de consulta deve ser fixo nos sistemas tradicionais, mas nos SGSDs deve ser adaptativo a fim de tentar otimizar os resultados das consultas que estão sendo feitas concomitantemente.

Os SGSDs surgiram com o propósito de preencher algumas lacunas dos bancos de dados tradicionais. Tendo em vista o objetivo de complementação, é possível também realizar a integração dos dois sistemas, tirando proveito das vantagens de cada um deles. Com isso, é possível e útil armazenar os resultados obtidos com o SGSD para consultá-los posteriormente através de um SGBD.

2.1 Aplicações

As consultas sobre fluxo contínuo de dados podem ser utilizadas em vários domínios de aplicação. Alguns dos principais domínios são brevemente descritos a seguir.

Monitoramento de redes.

O monitoramento de redes pode ser efetuado tanto em redes de pequeno porte, com alguns poucos usuários, quanto na Internet que, segundo o *The ISC Internet Domain Survey*, possuía em janeiro de 2008 mais de 540 milhões de domínios utilizados por mais de um bilhão de usuários [28]. Esta contagem é feita através da tentativa de identificar todos os *hosts* da Internet através de uma busca completa no DNS.

A utilização de redes, de uma maneira geral, é inevitável para boa parte da população. Com isso, a qualidade delas se torna imprescindível e um fator crucial em algumas profissões. No entanto, da forma como as redes foram planejadas e concretizadas, não foi previsto um padrão para se efetuar monitoramentos.

Existem ferramentas gratuitas e comerciais com diferentes objetivos para efetuar monitoramentos de diversos tipos. Porém, a diversidade das redes, a quantidade de dados e a velocidade na qual eles trafegam atingem números gigantescos, exigindo soluções diferenciadas para cada

problema. Nos próximos capítulos é possível obter mais detalhes sobre monitoramento de redes, aplicação na qual os SGSDs tem se mostrado bastante eficientes e versáteis.

Mercado de ações.

Na bolsa de valores é preciso trocar informações sobre as ações entre seus vendedores e compradores de forma bastante dinâmica. O mercado financeiro também necessita que análises sejam feitas de forma rápida para descobrir as correlações entre as ações, identificar tendências e prever valores futuros.

O Traderbot [2] é um mecanismo de busca que avalia informações financeiras em tempo real que tem como objetivo atender tais necessidades. Esta ferramenta é um exemplo de como as características de um SGSD se adequam também para este tipo de mercado [6]. O sítio do Traderbot mostra alguns exemplos de consultas contínuas sobre *streams* que são utilizadas pelos seus clientes.

Redes de sensores.

Cada sensor de uma rede de sensores é um aparelho funcionando à bateria que mede determinados tipos de informação tais como temperatura, ruído, luz e informações sismográficas. A limitação de energia é uma das principais características destas redes. Portanto, é importante fazer com que o tempo de vida útil de cada sensor seja o maior possível, aumentando a autonomia da rede como um todo.

Os sensores normalmente geram uma grande quantidade de pequenos dados em forma de tuplas. A forma como os dados gerados por cada um destes sensores devem ser tratados é de fundamental importância pois a otimização nesta etapa pode fazer uma grande diferença no seu desempenho. Em [17] é mostrado através de simulações como é possível obter resultados satisfatórios fazendo uso do processamento de *streams*. Foi possível aumentar o tempo de vida de uma rede de sensores, além de reduzir o atraso comum neste tipo de aplicação. O uso do processamento de *streams* se mostrou eficaz para projetar redes de sensores desde que não necessitem de precisão nos resultados.

Monitoramento de chamadas telefônicas.

O mercado de telecomunicações possui a necessidade de monitorar um grande volume de

informações provenientes de chamadas telefônicas. As empresas do ramo fazem uso de *streams* para identificar fraudes em seus sistemas, determinar padrões de uso dos consumidores e também para aplicar tarifação das chamadas [37].

2.2 Sistemas Gerenciadores Conhecidos e Trabalhos Relacionados

Os trabalhos de pesquisa envolvendo SGSDs são recentes, e a maioria dos sistemas desenvolvidos ainda são protótipos. Dentre eles podem ser citados o Aurora, o Medusa, o TelegraphCQ, o STREAM e o Gigascope.

O protótipo do projeto Aurora [4] foi desenvolvido em 2002 para uma companhia de defesa americana, conseguindo resolver facilmente problemas antes considerados difíceis pela empresa. Ele foi desenvolvido em Java e as consultas são feitas em uma interface gráfica na qual operadores são conectados a outros operadores ou podem simplesmente fornecer resultados. Estes operadores podem ter como fonte de dados a saída de outros operadores ou fontes de dados externas. Como uma das características principais deste projeto pode ser citado o dispersor de carga. Através de índices de QoS fornecidos por cada um dos operadores, e índices especificados pelo usuário como entrada do sistema, o Aurora é capaz de otimizar as consultas de forma a descartar mais ou menos dados de entrada conforme necessário. Esta otimização impacta também na precisão dos resultados. O Aurora foi o precursor do Medusa.

O sistema Medusa [8], sucessor do Aurora, foi na verdade apenas uma etapa intermediária para o desenvolvimento do Borealis, sistema de segunda geração descrito a seguir. Apesar de ser distribuído, ele provê apenas duas características adicionais em relação ao Aurora: gerenciamento de carga e alta disponibilidade. Ele foi responsável por identificar e alavancar algumas das novas características do Borealis.

O TelegraphCQ [14] foi uma das ferramentas mais citadas e utilizadas nas pesquisas da área. O sistema começou com o projeto Telegraph em 2000, e se baseou em dois protótipos criados pelo grupo, o CACQ e o PSoup. Ambos tinham por objetivo adicionar características de compartilhamento de processamento ao Telegraph, uma arquitetura para fluxo de dados adaptativo. O desenvolvimento do TelegraphCQ pode tirar proveito da experiência e dos protótipos que já haviam sido criados pelo grupo. Seu principal diferencial é a adaptabilidade, ou seja, os com-

ponentes de gerenciamento de dados do sistema foram especificados de forma a serem capazes de rapidamente evoluir e se ajustar a mudanças bruscas no conteúdo e disponibilidade de dados, características da rede e do sistema, contexto e necessidades do usuário [14]. A linguagem de consulta utilizada pelo TelegraphCQ é uma extensão de SQL, limitando algumas funções e definindo outras, como janelas de tempo. Estas janelas especificam o período de tempo que deve ser considerado para a realização da consulta. A implementação foi feita de forma a estender a arquitetura e a implementação do PostgreSQL.

O projeto STREAM [7] foi sem dúvida um dos precursores e um dos primeiros grupos a gerar um protótipo para testes. As pesquisas foram oficialmente encerradas no início de 2006. Seus estudos focaram em otimização de consultas, gerenciamento de recursos, e aproximações estáticas e dinâmicas no processamento de consultas. Foi definida uma nova linguagem denominada CQL que, assim como no TelegraphCQ, nada mais é do que uma extensão de SQL. Disponibiliza também uma ferramenta gráfica para acompanhar o monitoramento das consultas.

O Gigascope [16] é uma ferramenta comercial desenvolvida especificamente para monitoramento de redes de alta velocidade. Apesar de não haver nenhuma versão disponibilizada, foi o grupo que apresentou os resultados mais significativos em se tratando de monitoramento de redes, mostrando inclusive vantagens sobre ferramentas específicas como o Netflow [15].

Alguns estudos foram feitos sobre estes protótipos, e os resultados apresentados são encorajadores. Um exemplo é o estudo de caso feito sobre o SGSD TelegraphCQ, descrito em [30]. O objetivo deste trabalho era comparar as funcionalidades fornecidas por este SGSD com aquelas existentes na T-RAT [40], uma ferramenta para analisar a dinâmica de uma rede. Este estudo serviu de inspiração para a implementação da ferramenta proposta neste trabalho. Outros estudos de caso que demonstram a possibilidade de utilização do SGSD Borealis são: um jogo com suporte a múltiplos usuários [6] e sua utilização em um ambiente com diversas peculiaridades, como uma rede de sensores [5].

O objetivo deste capítulo foi o de mostrar o que são e como funcionam os SGSDs. Além disso foi apresentada uma breve amostra das áreas de aplicação, assim como os estudos feitos nesta área. No próximo capítulo são apresentados detalhes do monitoramento de redes, um dos principais domínios de aplicação dos SGSDs.

CAPÍTULO 3

MONITORAMENTO DE REDES

O monitoramento de redes tem como foco principal garantir o bom funcionamento de redes de todos os tamanhos e tipos. Neste capítulo são mostrados os objetivos do monitoramento e algumas ferramentas utilizadas para este fim, além de descrever classificações da forma como eles podem ser realizados.

3.1 Objetivos e Ferramentas de Monitoramento

As redes são baseadas em um modelo de serviço de otimização, fazendo com que a simplicidade na troca de pacotes e a flexibilidade no roteamento ofereçam capacidade para conectar milhões de usuários com um desempenho aceitável [27]. No entanto, novas aplicações e serviços passaram a ser disponibilizados nos últimos anos. Assim, é necessário que seja garantido um limiar mínimo aceitável no desempenho da rede para que as aplicações funcionem como esperado. Para isso foi necessário prover meios de garantir não apenas uma rede otimizada, mas uma rede de boa qualidade. A grande questão é definir exatamente o que é uma boa qualidade, ou seja, quais são os requerimentos mínimos de cada aplicação. Através do monitoramento é possível, por exemplo, dimensionar melhor e identificar problemas de configuração existentes que possam estar prejudicando uma rede.

Um bom ponto de partida são as 33 métricas definidas pelo grupo de trabalho IPPM [34]. Estas métricas são subdivididas em 7 subconjuntos, cada um descrito em um documento separado:

- RFC 2678 - métricas para medir conectividade;
- RFC 2679 - métricas de atraso unidirecional;
- RFC 2680 - métricas para perda unidirecional de pacotes;
- RFC 2681 - métricas de atraso no trajeto completo;
- RFC 3357 - exemplos de métricas para padrão de perda unidirecional;

- RFC 3393 - métrica de variação de atraso de pacote IP;
- RFC 3432 - métricas para streams periódicas.

Além desses, outros tipos de monitoramento podem ser necessários para atender determinados requisitos, tais como: utilização da banda disponível, visão geral sobre o fluxo de dados, identificação de possíveis congestionamentos e falhas de equipamentos, além de requisitos de segurança como identificação de intrusos.

Um método comum de análise de redes é através da captura de pacotes que estão trafegando em um dado momento, usando ferramentas como o *tcpdump* [31] ou a biblioteca *pcap* [13]. Este método provê uma grande quantidade de dados detalhados para que análises possam ser feitas posteriormente. Porém, a necessidade de armazenamento causada pelo excesso de dados é bastante significativa. Com isso, muitas vezes a captura tem que ser feita por pequenos períodos de tempo, por amostragem, ou até mesmo pela captura parcial dos pacotes como alguns dados do cabeçalho. Este tipo de cenário normalmente armazena as informações em servidores com centenas de milhares de arquivos que são analisados posteriormente por *scripts* desenvolvidos especialmente para um determinado sistema. Além disso, em contrapartida ao armazenamento de informações detalhadas, existem outros problemas ocasionados por este método, como por exemplo, a segurança da informação [16].

Outra possibilidade para efetuar o monitoramento é a utilização de ferramentas prontas. Existe uma grande variedade deste tipo de ferramenta [32], tanto comerciais quanto gratuitas, além de algumas com código aberto. Porém, o grande problema destas ferramentas é a falta de flexibilidade. Normalmente elas já provêem um determinado conjunto de informações que nem sempre são as mais apropriadas para atender as necessidades da rede em questão. No caso das ferramentas com código aberto, alterá-las nem sempre é uma tarefa fácil, exigindo a contratação de profissionais e bastante tempo para tentar adequar a ferramenta para atingir os objetivos que uma determinada rede demanda. Em relação as ferramentas comerciais, além do custo para a compra das ferramentas, muitas vezes não é permitido alterá-las, sendo preciso negociar com o vendedor a adequação específica para um cliente, exigindo investimento e um tempo de espera significativos.

Há ainda um método intermediário que é usar informações capturadas diretamente nos rote-

adores. Tanto o Netflow [15], quanto ferramentas baseadas em SNMP [33] permitem este tipo de captura. Nestes casos o problema de armazenamento e gerenciamento dos dados não é tão grande, mas em contrapartida, como os dados são agregados, as vezes não é possível obter a informação necessária.

Para tentar resolver os problemas encontrados nos métodos acima, algumas soluções para SGSDs começaram a ser desenvolvidas e analisadas como é o caso do Gigascope [16]. Os resultados apresentados são bastante favoráveis, mas ainda apresentam alguns dos problemas dos outros métodos, como a falta de flexibilidade. Neste documento é apresentada uma proposta e um protótipo de ferramenta para solucionar boa parte destes problemas usando o SGSD Borealis.

3.2 Tipos de Monitoramento

Para realizar um monitoramento é preciso primeiramente definir o local no qual ele será efetuado. Existem duas possibilidades:

Dentro da rede. Este tipo de monitoramento é interessante por exemplo para provedores de serviço, especialmente aqueles que gerenciam o coração da rede. Estes provedores precisam detectar quaisquer problemas com o *backbone*. Nestes casos os pontos de captura devem ser colocados entre dois roteadores centrais. Neste cenário é possível capturar um grande número de informações essenciais, porém, consumindo boa parte dos recursos do sistema como um todo.

Limiares da rede. Os pontos de captura neste tipo de monitoramento são colocados nas periferias de cada rede, como por exemplo o roteador de acesso a um determinado domínio, ou até mesmo em um cliente específico. Uma possível aplicação para estes casos é a efetivação de tarifação do cliente de acordo com contratos pré-estabelecidos do serviço a ser fornecido. Uma das motivações é o fato de que qualquer tráfego a ser monitorado terá o mesmo desempenho que o tráfego nos clientes.

Outra forma bastante comum de classificar os monitoramentos é se eles acontecem de forma *online* ou *offline*. No primeiro caso os dados são analisados no momento em que estão trafegando, como é o caso dos *firewalls*. O segundo tipo analisa posteriormente os dados armaze-

nados, podendo gerar estatísticas, identificar invasões ou até mesmo fazer a tarifação de um provedor de serviço.

A última classificação a ser apresentada nesta seção é sobre a técnica a ser utilizada durante o monitoramento, as quais são descritas a seguir.

3.3 Monitoramento Passivo

Este tipo de monitoramento analisa pacotes que estão trafegando na rede. Toda a análise é feita sobre o tráfego observado. A grande vantagem é que o desempenho da rede não é prejudicado, pois não há inserção de tráfego extra. No entanto, os dados a serem coletados podem ser excessivamente grandes, na ordem de *Terabytes*, principalmente quando se trata de monitoramento *offline*. O desafio é conseguir restringir a quantidade de dados coletados, conseguindo armazenar todas as informações necessárias.

Outro problema é a forma como os pacotes são monitorados. É preciso fazer uso do chamado modo promíscuo que consiste na cópia de pacotes interceptados durante a transmissão. Isso pode gerar problemas de violação de privacidade, mesmo que se utilize criptografia ou se apliquem filtros no conteúdo das mensagens, pois ainda assim será possível saber de onde vem e para onde vão cada um dos pacotes.

3.4 Monitoramento Ativo

O monitoramento ativo é feito através da inserção de pacotes de teste na rede. Nesta técnica apenas estes pacotes são submetidos a análise. Com isso, a necessidade de armazenamento é significativamente menor. No entanto, ele ocupa recursos da rede, aumentando a banda utilizada. O desafio neste caso é conseguir definir a quantidade de informação a ser inserida na rede, suficiente para obter as métricas desejadas, mas sem prejudicar o seu desempenho, e principalmente, sem atingir a escassez de recursos.

Normalmente este tipo de monitoramento estima a capacidade de um link, a banda disponível, além de poder definir métricas com precisão como o atraso de pacotes. Outro ponto importante desta técnica é a sincronização entre o ponto no qual o pacote é inserido e o ponto no

qual ele será capturado. Para isto pode ser usado um protocolo para sincronização dos relógios de computadores, conhecido como NTP [26], ou até mesmo *hardwares* como o GPS em casos em que a precisão é de extrema importância.

Até aqui foram apresentados os SGSDs e suas aplicações, assim como os objetivos do monitoramento de redes. Os dois tópicos englobam o escopo da ferramenta implementada. Antes de descrever a *PaQueT*, é apresentado no próximo capítulo todos os detalhes e peculiaridades do Borealis, o SGSD utilizado durante este estudo.

CAPÍTULO 4

O SISTEMA GERENCIADOR DE STREAMS DE DADOS BOREALIS

O Borealis é o SGSD que foi escolhido para implementar a *PaQueT*, ferramenta proposta neste trabalho. Esse sistema possui características próprias e inovadoras tais como *registros de revisão*, *viagem no tempo* e *linhas de controle* [6], além de ser um sistema distribuído. Tal como em qualquer banco de dados distribuído [25], o Borealis também permite integração dos dados e compartilhamento de recursos e também implementa mecanismos de tolerância a falhas, processamento distribuído, escalabilidade, e balanceamento e dispersão de carga [6]. As características distribuídas do sistema são de extrema importância em termos de desempenho. Outra vantagem disto é que muitas aplicações para as quais os SGSDs foram projetados possuem entrada distribuída, o que facilita a captura dos dados.

Neste capítulo será descrito o Borealis, apontando detalhes em termos de funcionalidade. Na Seção 4.1 será apresentada uma breve descrição de características de banco de dados tradicionais distribuídos que foram incorporadas ao Borealis. Na seção seguinte são abordados detalhes da arquitetura do Borealis. Por fim uma breve explicação sobre como desenvolver uma nova aplicação para o Borelais.

4.1 Características Distribuídas

Existem diversos SGBDs no mercado que provêem as vantagens e características de um sistema distribuído, as quais foram incorporadas ao Borealis, tais como:

- *Alta disponibilidade (tolerância a falhas)*: serviços e recursos devem estar sempre disponíveis de maneira ininterrupta. Isto se faz através de redundâncias do sistema que devem ser ativadas no caso de detecção de falhas. O tempo de latência deve ser mínimo para que não seja notada a troca de responsabilidades entre os serviços e os recursos.
- *Processamento distribuído ou paralelo*: melhora o desempenho de grandes tarefas.

- *Escalabilidade*: permite que o sistema aumente ou diminua de complexidade de forma modular, fazendo com que quaisquer mudanças não impactem no seu funcionamento.
- *Balanceamento de carga*: o carregamento do sistema em termos de processamento sofre diversos picos no decorrer do seu uso. A distribuição mais igualitária possível deste processamento é fundamental para que não haja sobrecarga em nenhum dos nós. Em caso de falha de um dos nós, isto também se torna importante na divisão das tarefas entre os demais nós.

A pesquisa na área de SGBDs distribuídos é bastante antiga. Grande parte dos sistemas abertos e comerciais já possuem implementações e estratégias distribuídas. Porém, até pouco tempo, todos os desenvolvimentos no que se refere à parte distribuída eram feitos de forma que a capacidade de distribuição do processamento fosse implementado diretamente no banco, sendo que suas configurações não pudessem ser modificadas pelos administradores e/ou pelos usuários.

Com a evolução e a popularização do mercado computacional, sistemas distribuídos se tornaram mais comuns e necessários em todas as áreas, não só para banco de dados. Com isso também surgiu uma nova abordagem: a de se utilizar bancos de dados tradicionais que não estão preparados para distribuição em ambientes distribuídos. Esta técnica é recente e ainda vem sendo pesquisada. No projeto *ClusterMiner* [25] é feito um levantamento de diferentes maneiras como isso pode ser feito, englobando cluster de banco de dados, a própria internet e também Grids, que podem ser vistos como um misto das duas primeiras abordagens. O objetivo do *ClusterMiner* é avaliar processamentos paralelos disponíveis no mercado com uma implementação de cluster de banco de dados.

Seja qual for a maneira de se usar SGBDs distribuídos, existem três objetivos principais que devem ser sempre atingidos[38]: integração de informação, compartilhamento de recursos e melhora de desempenho. Todos os estudos aplicados aos SGBDs apresentados até agora, podem também ser utilizados para os SGSDs, em especial o Borealis. Alguns exemplos são verificados na Seção 6.5.

4.2 Funcionalidade

Apesar do processamento de *streams* ser uma área de pesquisa bastante recente, tendo seu início no começo desta década, vários grupos conseguiram gerar protótipos e atingir resultados importantes na área de SGSDs. Os sistemas gerenciadores de *streams* de dados de primeira geração foram os primeiros protótipos desenvolvidos, que tinham como foco definir vantagens e validar a utilidade de se criar um novo tipo de sistema gerenciador de processamento de *streams* ao invés de utilizar os antigos sistemas gerenciadores de banco de dados tradicionais. A implementação desses sistemas foi feita para processamento centralizado, sem possibilidade de integração. Surgiu então o Borealis [3], o sistema escolhido para o desenvolvimento da ferramenta proposta neste trabalho, o qual é considerado um SGSD de segunda geração. De acordo com as pesquisas feitas este é o único protótipo disponível de SGSD distribuído. A escolha foi feita primeiramente por ser um sistema bem documentado, por disponibilizar todo seu código fonte, por ter suporte disponível da equipe de desenvolvimento, além de se tratar de um dos sistemas gerenciadores de *streams* de dados de segunda geração mais consolidados atualmente.

O Borealis foi baseado em dois outros sistemas desenvolvidos pelo mesmo grupo: o Aurora [4] e o Medusa [8]. A equipe que gerou este sistema teve inspiração e experiência significativas com as duas ferramentas precursoras desenvolvidas pelo grupo [3]. O Borealis herda a funcionalidade de processamento de *streams* do Aurora, sendo a sua parte central. Já a funcionalidade das características de sistemas distribuídos vieram do amadurecimento do desenvolvimento do Medusa. Porém, o Borealis não é apenas uma junção dos dois sistemas anteriores e sim um aproveitamento do que já havia sido feito, tendo sido os dois sistemas modificados e estendidos de forma a prover um sistema ainda mais abrangente.

Os grupos de desenvolvimento geralmente mostram em seus trabalhos aplicações práticas dos protótipos desenvolvidos e não foi diferente com o Borealis. O estudo de caso escolhido foi o cenário de um jogo de primeira pessoa com múltiplos jogadores. A descrição da demonstração pode ser encontrada em [6], assim como os detalhes da avaliação dos resultados dos testes de tolerância a falhas, balanceamento de carga, alta disponibilidade e afins. Todos os testes tiveram resultados satisfatórios com o que estava sendo proposto. Foi feito também outro experimento

já citado sobre uma rede de sensores que também obteve ótimos resultados [5].

Dois aspectos fundamentais mostram porque o Borealis pode ser considerado um SGSD de segunda geração. Em primeiro lugar pelas características de um sistema distribuído implementadas pelo Borealis. Entre elas podem ser citadas a distribuição de carga, o dispersor de carga, a alta disponibilidade e a disseminação e o processamento paralelo.

O segundo aspecto que faz parte da evolução do Borealis são os registros de revisão, viagem no tempo e linhas de controle. Os registros de revisão são correções nos dados previamente informados devido a detecção de eventuais correções que se façam necessárias. A viagem no tempo é a possibilidade de se fazer consultas em dados históricos. Pode ser considerada uma maneira mais amigável e flexível de consultar dados já processados do que simplesmente efetuar a consulta sobre dados previamente armazenados como fazem a maioria dos SGSDs. Por fim, as linhas de controle têm por objetivo permitir a modificação de parâmetros de operadores, e de operadores em si, com o objetivo de, assim como a viagem no tempo, permitir a atualização das consultas *on the fly*. Cada uma dessas características, as quais são detalhadas em [3], permitem disponibilizar requisitos fundamentais para os SGSDs, tais como revisão dinâmica do resultado das consultas, modificação dinâmica das consultas, além de otimização flexível e de larga escala.

Na Figura 4.1 são mostrados três dos quatro componentes do sistema: os *Nós*, os *Aplicativos Clientes* e as *Fontes de Dados*. Além deles, também há o *Catálogo Distribuído*, o qual contém as informações gerais do sistema como um todo, tendo como função principal viabilizar o melhor aproveitamento possível dos recursos disponíveis. O funcionamento completo do Catálogo Distribuído e suas respectivas subdivisões, assim como sua definição e implementação podem ser encontrados em [9]. Todos os demais componentes são detalhados em [11].

A interação entre os componentes pode ser vista ainda na Figura 4.1. Cada uma das fontes de dados (1) é encaminhada aos nós responsáveis por processá-las. Os dados de entrada de cada um dos nós (2) podem tanto ser tais fontes de dados, como os próprios resultados gerados por algum outro nó. Por fim, a informação final (3) é passada para o Aplicativo Cliente o qual poderá disponibilizar o resultado inclusive para outras consultas. Este último, além de disponibilizar os dados finais, pode também executar tarefas tais como redefinir consultas e associar operadores

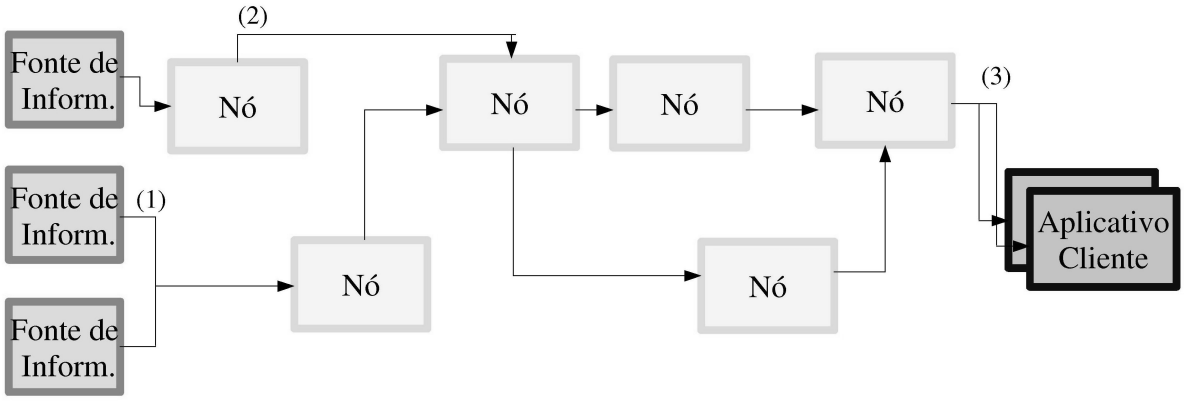

Figura 4.1: Arquitetura do SGSD Borealis.

a determinados nós.

O sistema também pode ser dividido em módulos que foram desenvolvidos na implementação atual para atender as especificações do projeto. Entre eles podemos citar:

- Ferramenta de Processamento de Streams;
- Coordenador;
- Gerenciador de Carga;
- Dispersor de Carga;
- Módulo de tolerância a falhas;
- Mecanismo de revisão de processamento.

Na Figura 4.2 é apresentada a arquitetura de um nó Borealis [11]. Cada nó contém todos os componentes necessários para poder funcionar como um SGSD independente.

O principal componente é o *Processador de Consultas*, responsável através do módulo *Admin* por gerenciar as requisições e alocá-las de acordo no diagrama de consultas. Em alguns casos é responsável também por enviar alguns operadores para outros processadores de consulta. O *DataPath* é um roteador das *streams*, tanto de entrada como de saída, enquanto o *Catálogo Local* é o responsável por armazenar uma imagem atualizada do estado dos operadores, consultas e *streams* registrados no Processador de Consultas em questão. Já o *Nó Aurora* pode ser definido como sendo o processador de *streams* propriamente dito.

Além do Processador de Consultas, cada nó ainda deve conter uma camada de comunicação que deve ser independente para que funcione da mesma maneira entre nós locais e remotos.

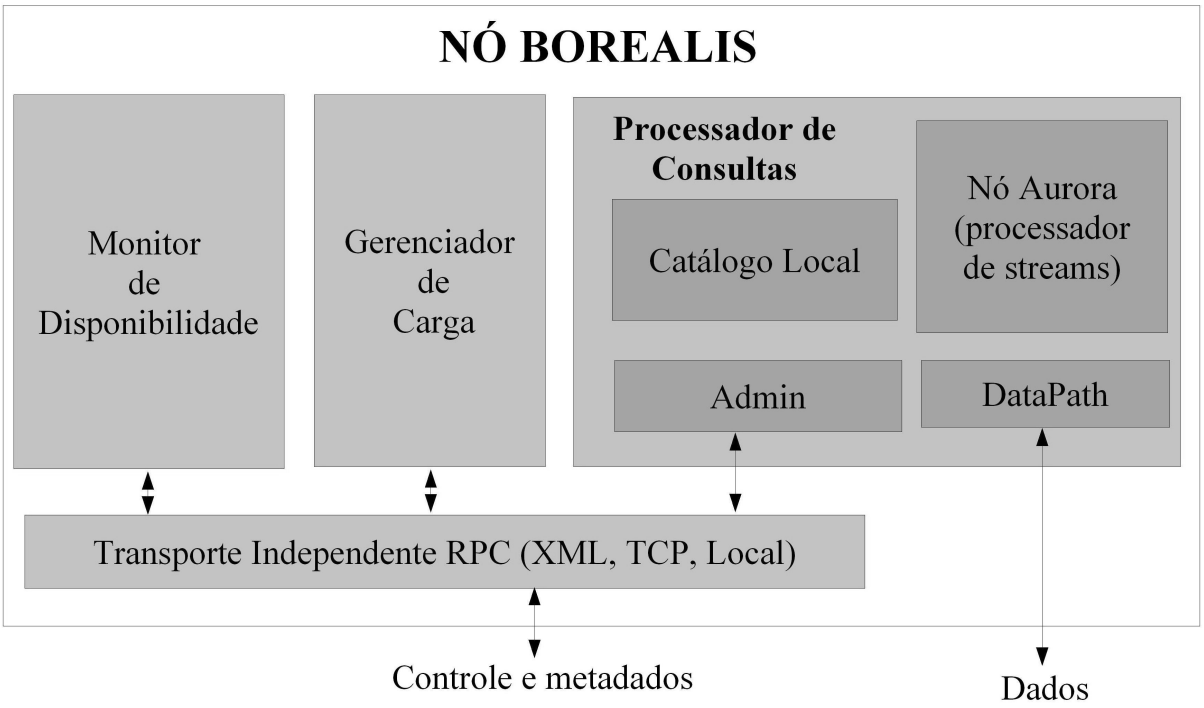

Figura 4.2: Arquitetura de *software* de um nó Borealis.

Cada nó também contém informações sobre a disponibilidade dos demais nós através do *Monitor de Disponibilidade*. E por último, um *Gerenciador de Carga* existe para balancear melhor os recursos disponíveis entre os nós.

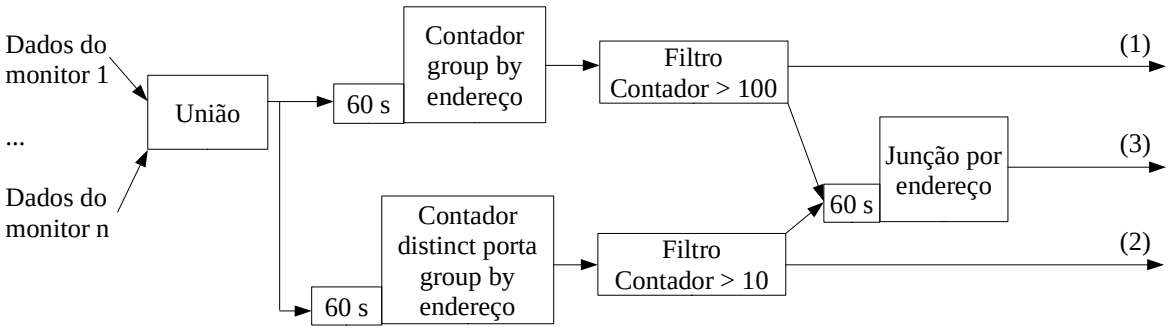

Figura 4.3: Exemplo de diagrama de consulta para monitoramento de redes.

No que se refere a forma de efetuar consultas, o Borealis não usa uma linguagem de consulta. É utilizado um diagrama de consultas em forma de caixas e flechas, nos quais os primeiros representam os operadores, enquanto as últimas representam o fluxo da informação. Os operadores suportados pelo Borealis [10] incluem operadores de álgebra relacional, tais como seleção, projeção, junção, união, assim como operadores de agregação como contagem, soma e

média. Operadores criados pelo usuário também podem ser definidos. Alguns operadores, como os de agregação, precisam que seja definida uma janela de tempo, a qual definirá a freqüência com que novos resultados serão fornecidos. Estas janelas de tempo podem ser definidas tanto em unidades de tempo como em quantidade de registros recebidos.

A Figura 4.3 ilustra como uma consulta para monitoramento de redes pode ser feita. Neste caso, os resultados apresentados são os endereços que iniciaram mais do que 100 conexões (1) ou endereços que se conectaram por mais de 10 portas diferentes (2) no último minuto. Além disso, também é feita uma junção entre os resultados mostrando os endereços que tanto iniciaram várias conexões quanto se conectaram por diversas portas (3).

4.3 Desenvolvendo uma Nova Aplicação

Nesta seção são descritas as etapas para a construção de uma nova aplicação para o Borealis [10]. Primeiramente os documentos XML contendo tanto os esquemas de entrada e saída como as consultas são dados como entrada para o *Marshal*, uma ferramenta do Borealis responsável por gerar código para serializar a informação a ser trocada entre as aplicações e o Borealis. A aplicação é um programa desenvolvido em C++. No caso de uma aplicação de monitoramento de redes similar ao utilizado na *PaQueT*, o programa é o responsável pela captura dos dados e a adequação do mesmo ao esquema de entrada.

O código gerado pelo *Marshal* provê interfaces de funções em C++ para receber e enviar os dados do Borealis, sendo que a implementação da definição dessas funções é de responsabilidade do desenvolvedor. Em ambos os casos eles são definidos como *streams* de acordo com os esquemas de entrada e saída respectivamente previamente informados. Após a compilação tanto do aplicativo quanto do programa gerado pela ferramenta *Marshal*, a consulta pode então ser executada pelo SGSD Borealis.

Duas grandes desvantagens de usar o Borealis diretamente como uma ferramenta de monitoramento de redes são: primeiro, a necessidade de se definir tanto o esquema de entrada quanto de saída de todas as consultas registradas no sistema; e a segunda e principal desvantagem é o fato de que novas consultas exigem pedaços de código a serem desenvolvidos em C++ e a conseqüente necessidade de sua compilação.

O próximo capítulo descreve a *PaQueT* em detalhes. Ela é, na verdade, uma extensão mais elaborada do SGSD Borealis que permite ao usuário definir consultas arbitrárias sobre conexões com *streams* cujos esquemas são definidos dinamicamente. Isto caracteriza uma das principais vantagens da *PaQueT* que é a flexibilidade.

CAPÍTULO 5

UMA FERRAMENTA GENÉRICA PARA MONITORAMENTO DE REDES

Neste capítulo é descrita a *PaQueT*, uma ferramenta genérica de monitoramento de redes, implementada utilizando o SGSD Borealis. Ao contrário de outras ferramentas existentes, na *PaQueT* as métricas retornadas pelo sistema são arbitrárias e definidas pelo usuário através de consultas definidas sobre os pacotes que trafegam pela rede. Tais consultas podem ser feitas através de uma ferramenta gráfica ou através de arquivos XML.

A interação do administrador de rede com a *PaQueT* é similar a como os usuários interagem com os SGBDs. Isto é, assim que o esquema de entrada for definido, consultas podem ser diretamente submetidas ao sistema, sem a necessidade de definição do esquema de saída ou de recompilação do aplicativo. Esta facilidade permite que a ferramenta seja usada por técnicos sem conhecimento nenhum de programação. Além disso, como será detalhado posteriormente, o usuário da *PaQueT* utiliza exatamente a mesma linguagem de consulta para monitorar a rede em diferentes níveis de granularidade.

No restante deste capítulo são descritos a arquitetura da *PaQueT*, o esquema dos pacotes, a linguagem de consulta, além de alguns detalhes relevantes da implementação. Por fim, é mostrada a aplicabilidade da ferramenta em um dos assuntos mais comentados da área de monitoramento: os acordos de níveis de serviço, ou simplesmente, SLAs.

5.1 Arquitetura

A Figura 5.1 mostra uma visão geral da arquitetura da *PaQueT*. A ferramenta consiste basicamente de três módulos: o SGSD Borealis propriamente dito, o *IP Tool* e o *Dynamic Stream Interface*. De forma similar aos SGBDs, o Borealis requer que o esquema dos dados seja previamente definido para que eles possam ser processados. Assim, a *PaQueT* possui um conjunto de esquemas pré-definidos chamado de *Packet Schema*, os quais descrevem a estrutura dos pacotes

que trafegam pela rede. O *Packet Schema* pode consistir somente de pacotes TCP e UDP, como mostra a Figura 5.2, ou pode ser estendido para outros protocolos suportados pelo IPv4, e/ou outros formatos como Netflow [15], Sflow [29], e SNMP [33].

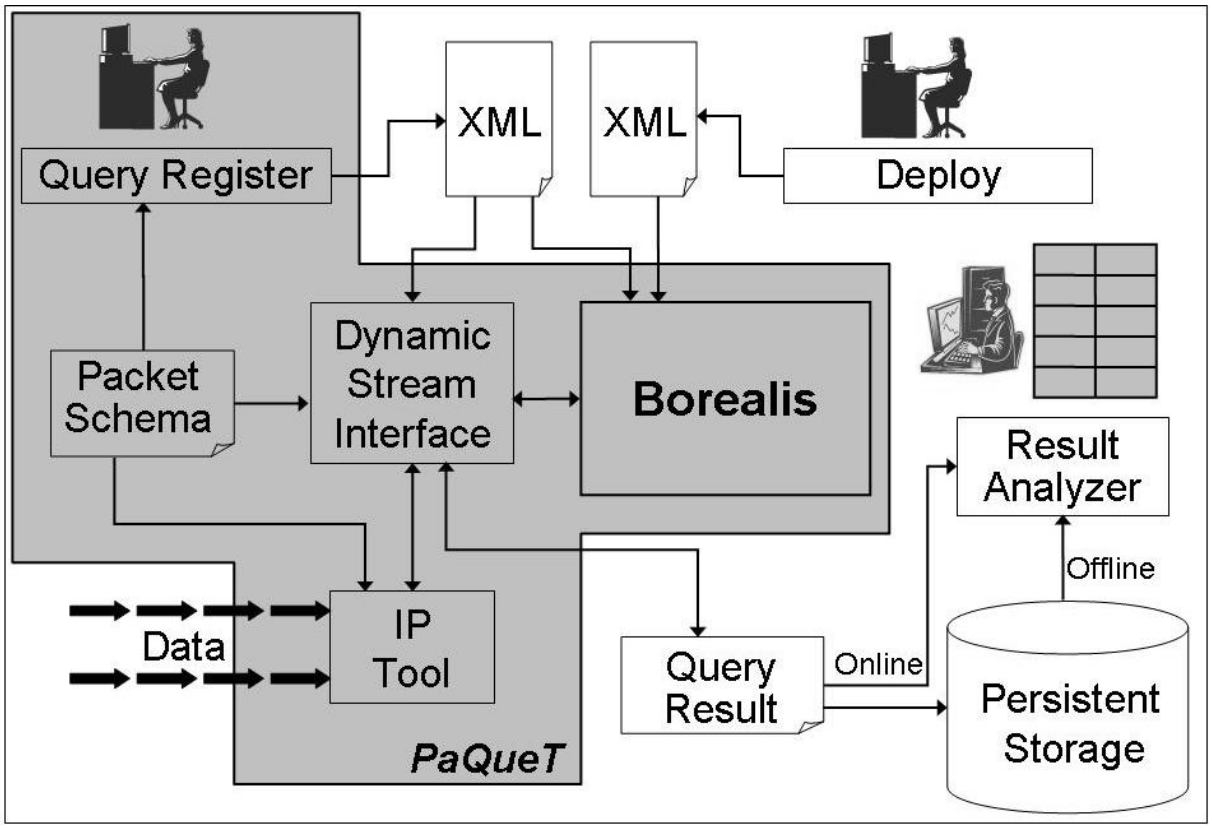

Figura 5.1: Arquitetura da *PaQueT*.

O *IP Tool* é o módulo responsável pela captura dos pacotes e sua decomposição nos campos previamente definidos no *Packet Schema*. Estes dados já decompostos são então enviados para o *Dynamic Stream Interface* para que as consultas possam ser encaminhadas para o Borealis a fim de serem processadas. Os dados provenientes do *IP Tool*, consistem apenas do subconjunto de todos os campos utilizados nos esquemas de entradas das consultas registradas na ferramenta. Este subconjunto é dinamicamente definido pelo *Dynamic Stream Interface* à medida que novas consultas vão sendo registradas.

A *PaQueT* não é limitada a um tipo específico de rede ou de dados de entrada, mas depende apenas da informação disponível nas fontes, as quais são representadas pelo *IP Tool*. Assim como todos os demais componentes da arquitetura, é possível ter diversas versões do *IP Tool* rodando simultaneamente. Para cada versão devem ser adicionados ao *Packet Schema* os novos

campos que passarão a ser disponibilizados. Portanto, para estender a ferramenta, basta implementar novas versões do *IP Tool* para dar suporte à captura de determinada informação. A versão usada no protótipo é totalmente baseada na biblioteca *Pcap*.

O *Dynamic Stream Interface* recebe como entrada um documento XML que contém a definição de uma ou mais consultas. Ele é o responsável por registrar a consulta no Borealis, inferir o esquema de saída e requisitar os campos de entrada para o *IP Tool* que disponibiliza as informações para cada consulta. O esquema de saída consiste de um registro de campos inferidos pelo *Dynamic Stream Interface* o qual se baseia na definição da consulta para determiná-lo.

Para utilizar a ferramenta, o usuário do sistema, normalmente o administrador da rede, registra as consultas (*Query Register*) para obter as informações desejadas. Opcionalmente, é possível também especificar um arquivo de distribuição dos recursos (*Deploy*), o qual deve conter informações sobre as responsabilidades de cada nó em um sistema distribuído. Ambas as especificações, de consulta e de recursos, podem ser feitas através de arquivos XML ou por uma interface gráfica, chamada Borgui, que é fornecida juntamente com o Borealis. A ferramenta gráfica irá simplesmente traduzir as consultas para os arquivos XML que devem ser dados como entrada ao Borealis. Para registrar uma consulta na *PaQueT* é necessário que o usuário tenha familiaridade com os esquemas definidos no *Packet Schema*, isto é, é necessário saber o nome e o significado de cada um dos campos para poder expressar corretamente as métricas desejadas.

Os resultados das consultas podem ser armazenados em uma tabela de um banco de dados tradicional (*Persistent Storage*) ou em um arquivo, ou até mesmo serem enviados para o *Result Analyzer* para que possam ser gerados relatórios a partir dos dados obtidos. Desta forma, os relatórios podem ser vistos à medida que os dados vão sendo gerados, ou podem ser obtidos do armazenamento persistente posteriormente. Com o resultado das consultas, o administrador é capaz de fazer diagnósticos sobre a rede e otimizar sua configuração. Outra facilidade fornecida pela *PaQueT* é a possibilidade de disparar eventos de acordo com o resultado de uma consulta, como por exemplo aquelas que detectam anomalias na rede.

```
<schema name="TuplaPacote">
    <field name="tempo"       type="timestamp"/>
    <field name="ether_dhost" type="string"   size="6"/>
    <field name="ether_shost" type="string"   size="6"/>
    <field name="ether_type"  type="string"   size="1"/>
    <field name="ip_vhl"      type="string"   size="1"/>
    <field name="ip_tos"      type="string"   size="1"/>
    <field name="ip_len"      type="int"/>
    <field name="ip_id"       type="string"   size="2"/>
    <field name="ip_off"      type="string"   size="2"/>
    <field name="ip_ttl"      type="string"   size="1"/>
    <field name="ip_p"        type="string"   size="1"/>
    <field name="ip_sum"      type="int"/>
    <field name="ip_src"      type="string"   size="4"/>
    <field name="ip_dst"      type="string"   size="4"/>
    <field name="tcp_sport"   type="int"/>
    <field name="tcp_dport"   type="int"/>
    <field name="tcp_seq"     type="long"/>
    <field name="tcp_ack"     type="long"/>
    <field name="tcp_off"     type="int"/>
    <field name="tcp_flags"   type="string"   size="1"/>
    <field name="tcp_win"     type="string"   size="2"/>
    <field name="tcp_sum"     type="int"/>
    <field name="tcp_urp"     type="string"   size="2"/>
    <field name="udp_sport"   type="int"/>
    <field name="udp_dport"   type="int"/>
    <field name="udp_len"     type="int"/>
    <field name="udp_sum"     type="int"/>
</schema>
```

Figura 5.2: Esquema dos dados de entrada da *PaQueT*.

5.2 O Esquema dos Pacotes

A *PaQueT* define o esquema dos pacotes sobre os quais o usuário pode definir suas consultas. Este esquema é definido em XML como apresentado na Figura 5.2. Ele consiste de uma seqüência de elementos para cada campo (`field`) do cabeçalho dos pacotes, onde cada campo possui um nome (`name`), um tipo (`type`), e opcionalmente um tamanho (`size`). Para simplificar, o esquema mostrado contém informações apenas sobre os protocolos da camada de transporte TCP e UDP. No entanto, a *PaQueT* pode ser estendida para dar suporte também a outros protocolos da camada de rede IPv4 ou outros que se façam necessários. O nome de

cada um dos campos foi escolhido de acordo com a definição que pode ser encontrada em [31]. É possível também definir mais de um esquema de entrada para a *PaQueT* permitindo que o usuário possa combinar informações de diferentes fontes e granularidades.

Um outro ponto que deve ser discutido é sobre a forma de representação dos esquemas. Um exemplo bastante simples é o da Figura 5.2 que define um esquema que cobre pacotes IP tanto de protocolo UDP quanto TCP. Alternativamente, poderiam ser criados dois esquemas distintos, um somente para pacotes TCP e outro somente para pacotes UDP, evitando o desperdício de campos que nunca são utilizados por serem mutuamente exclusivos. Esta discussão já é bastante conhecida no que se refere a SGBDs e pode ser aproveitada também neste cenário.

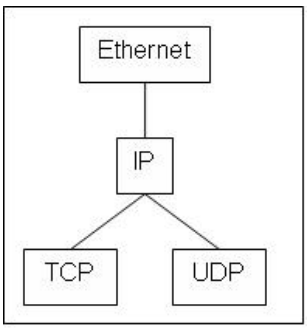

Figura 5.3: Modelo com a hierarquia dos pacotes.

A Figura 5.3 pode representar o modelo do exemplo anterior. Este relacionamento é conhecido como especialização. Há diversas opções para o mapeamento de um grupo de subclasses que, juntas formam uma especialização (ou, alternativamente, são generalizadas em uma superclasse)[20]. Apenas algumas dessas opções são apresentadas nesta seção.

Considerando que cada especialização possui m subclasses S_1,S_2,...,S_m e a superclasse C(generalizada), em que os atributos (Atr) de C são k,a_1,a_2,...a_n e k é a chave primária (PK) é possível:

1. *relações múltiplas - superclasse e subclasse.* Criar uma relação L para C com os atributos $Atr(L) = k, a_1, a_2, ...a_n$ e chave primária $PK(L) = k$. Criar uma relação L_i para cada subclasse $S_i, 1 \leq i \leq m$, com os atributos $Atr(L_i) = k \cup$ atributos deS_i e $PK(L_i) = k$. Essa opção funciona para qualquer especialização (total ou parcial, disjuntas ou sobrepostas).

2. *relações múltiplas - somente relações de subclasse.* Criar uma relação L_i para cada subclasse $S_i, 1 \leq i \leq m$, com os atributos $Atr(L_i) =$ atributos de$S_i \cup k, a_1, a_2, ...a_n$ e chave primária $PK(L_i) = k$. Essa opção funciona somente para as especializações cujas subclasses são totais (toda entidade de uma superclasse deve pertencer a pelo menos uma subclasse).
3. *relação única com um atributo tipo.* Criar uma única relação L com os atributos $Atr(L) = k, a_1, a_2, ...a_n \cup$ atributos de $S_i \cup ... \cup$ atributos de$S_m \cup t$ e chave primária $PK(L) = k$. O atributo t é chamado de atributo tipo, que indica a subclasse à qual cada tupla pertence, se pertencer a alguma. Essa opção funciona para as especializações cujas subclasses são disjuntas.

A decisão na escolha da opção 3 foi influenciada pela implementação da *PaQueT*. A forma como o *IP Tool* e o *Dynamic Stream Interface* se comunicam favorecem o uso de um único esquema para representar uma especialização, como é o caso da Figura 5.2, visto que apesar de ser transparente para os usuários os atributos específicos de outras especializações não são utilizados na prática. Os detalhes desta integração são descritos na seqüência.

Na próxima seção é apresentada a linguagem de consulta do SGSD Borealis para ilustrar como o esquema definido pode ser utilizado para expressar consultas no sistema.

5.3 Uma Interface para Construção de Consultas.

O SGSD Borealis possui uma ferramenta chamada *Borealis Graphical User Interface* (Borgui), que é uma interface gráfica para o usuário construir suas consultas. Na Borgui, as consultas são expressas através de diagramas compostos de caixas, que representam os operadores, e flechas, que representam o fluxo da informação. Os operadores que podem ser usados na *PaQueT*, são os mesmos definidos para o Borealis já descritos na Seção 4.2, os quais são baseados nos conceitos da álgebra relacional e do SQL. É possível definir também uma janela de tempo, ou seja, o intervalo que determina a periodicidade em que um novo valor é gerado. Como já mencionado anteriormente, a janela de tempo pode ser definida tanto por unidade de tempo, como pela quantidade de dados recebidos.

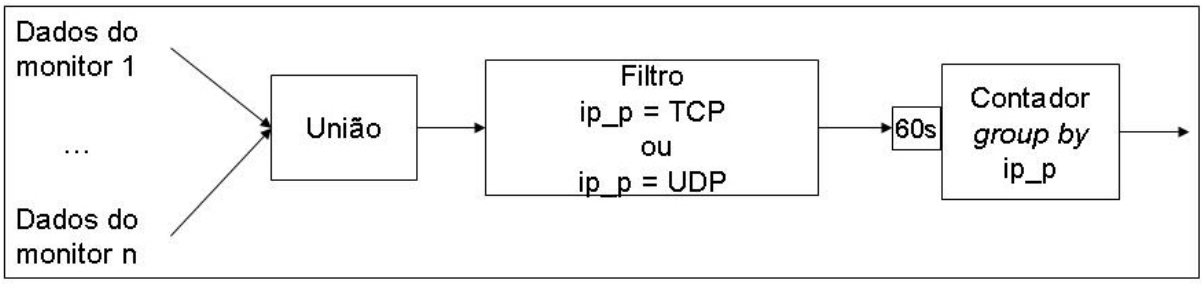

Figura 5.4: Exemplo de um diagrama de consulta.

Para exemplificar como uma consulta é expressa no sistema, considere a Figura 5.4. A consulta retorna a quantidade de pacotes UDP e TCP que passaram pela interface de rede a cada intervalo de 60 segundos. Para isso, todos os pacotes são capturados pelo operador de `união`, criando um único fluxo de dados que é direcionado para o `filtro` (operador de seleção). Este operador passa para o próximo operador somente os pacotes cujo protocolo (`ip_p`) seja UDP ou TCP. Finalmente, o operador de agregação faz a contagem de quantos pacotes de cada tipo passaram pela interface a cada 60 segundos e retorna estes valores.

Para que uma consulta seja processada pelo Borealis, ela é primeiramente traduzida para um arquivo XML. Assim, uma forma alternativa de expressar uma consulta é através de um arquivo XML diretamente, sem a utilização da linguagem visual. A Figura 5.5 apresenta a consulta ilustrada na Figura 5.4 expressa em XML, porém sem o operador de união. Ou seja, ela representa a mesma consulta sobre apenas um ponto de monitoramento, já que o operador de união só precisa ser aplicado para criar um único fluxo de pacotes provenientes de múltiplos pontos.

Este exemplo de consulta mostra como elas podem ser facilmente construídas pelo usuário do sistema. Além disso, como todos os campos dos pacotes podem ser utilizados nas consultas, a *PaQueT* pode ser utilizada inclusive para o monitoramento de conteúdo, desde que isso não viole os direitos dos usuários da rede.

5.4 Implementação

Para construir a *PaQueT*, além de ser definido o esquema de entrada, foi também desenvolvido em C++ o aplicativo *IP Tool*, responsável pela captura dos pacotes e sua quebra de

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">
<borealis>
    <input  stream="Pacote"          schema="TuplaPacote"/>
    <schema name="TuplaPacote">
        <field name="tempo"        type="int"/>
        <field name="ip_p"         type="string"  size="4"/>
    </schema>
    <query name="NumPacotesUdpTcp">
        <box name="Filtro"  type="filter">
            <in stream="Pacote"/>
            <out stream="Filtro"/>
            <parameter name="expression.0"        value="ip_p='tcp' || ip_p='udp'"/>
            <parameter name="pass-on-false-port"  value="0"/>
        </box>
        <box name="Contagem"  type="aggregate">
            <in stream="Filtro"/>
            <out stream="Agregacao"/>
            <parameter name="aggregate-function.0"                 value="count()"/>
            <parameter name="aggregate-function-output-name.0"  value="numPacotes"/>
            <parameter name="window-size-by"                       value="VALUES"/>
            <parameter name="window-size"                          value="60"/>
            <parameter name="advance"                              value="60"/>
            <parameter name="order-by"                             value="FIELD"/>
            <parameter name="order-on-field"                       value="tempo"/>
            <parameter name="group-by"                             value="ip_p"/>
            <parameter name="independent-window-alignment"     value="1"/>
            <parameter name="drop-empty-outputs"                   value="1"/>
        </box>
    </query>
</borealis>
```

Figura 5.5: Consulta para contagem de pacotes por protocolo.

acordo com os respectivos cabeçalhos e campos. Para implementar o *IP Tool*, foi utilizada a biblioteca *Pcap* [13] para a captura e decomposição dos pacotes. O *Dynamic Stream Interface* é o responsável por decidir quais campos serão encapsulados para compor o fluxo de entrada do SGSD para o processamento das consultas registradas. Para evitar tráfego desnecessário, apenas as informações necessárias chegam ao Borealis. O código fonte dos dois módulos está disponível no Apêndice B.

Existem várias formas de comunicação que poderiam ser utilizadas entre o *IP Tool* e o *Dynamic Stream Interface*. A comunicação interprocessos oferece um grande leque de opções,

cada uma com suas peculiaridades. Foram analisadas três formas dentre as opções que atendiam as necessidades, e são comuns dentro do conceito produtor/consumidor: a utilização de *sockets*, o uso de *named pipes* e a integração dos dois aplicativos removendo a necessidade de comunicação interprocessos.

Os *sockets* tem a vantagem de poderem funcionar dentro da rede enquanto os *named pipes* permitem apenas a comunicação em um mesmo computador. Em termos de desempenho, os dois métodos são equivalentes. No entanto, pelo fato da *PaQueT* ser uma ferramenta de monitoramento de rede, a versão final foi implementada usando *named pipes*. Foi testada também apenas para verificação de desempenho a integração das responsabilidades das duas ferramentas em uma. Como não houve melhora significativa, esta opção foi também descartada.

A forma de comunicação entre os dois módulos é bastante simples. Cada *IP Tool* deve ser iniciado antes do *Dynamic Stream Interface*. Após carregar a consulta, o *Dynamic Stream Interface* envia para o *IP Tool* o número do processo e os campos que precisa para atender as consultas. A partir desse momento, enquanto o *pipe* do processo em questão se mantiver, as informações serão enviadas ao Borealis assim que forem capturadas. O Borealis irá então processar os dados de entrada e gerar os resultados de acordo com as consultas registradas. Por não ser foco do estudo, a ferramenta em seu estado atual apenas escreve na saída padrão os resultados de cada consulta como mostra a figura 5.6.

5.5 Aplicabilidade

Um novo tipo de documento vem sendo utilizado a fim de especificar requisitos e garantir o QoS fornecido pelos prestadores de serviço, principalmente no que diz respeito a Tecnologia da Informação. Os acordos de nível de serviço, ou SLA (*Service Level Agreement*) como são conhecidos, estão sendo amplamente utilizados como um contrato entre consumidores e prestadores de serviço. O objetivo é definir detalhadamente a qualidade mínima dos serviços a serem prestados, especificando penalidades que devem ser aplicadas no caso de não se atingir os níveis de qualidade acordados.

Uma ferramenta como a *PaQueT* atinge grande parte das necessidades do mercado de monitoramento de redes. Porém, a *PaQueT* foi inicialmente idealizada para funcionar como uma

```
Saida: ConsultaNumPacotes
Tempo: 60
Contagem: 31527
Saida: ConsultaNumPacotes
Tempo: 120
Contagem: 23478
Saida: ConsultaNumPacotes
Tempo: 180
Contagem: 26875
Saida: ConsultaNumPacotes
Tempo: 240
Contagem: 19834
Saida: ConsultaNumPacotes
Tempo: 300
Contagem: 25456
...
```

Figura 5.6: Resultados escritos na saída padrão de uma consulta para contar o número de pacotes a cada sessenta segundos.

ferramenta que ajudasse no cumprimento de um SLA, o qual pode ser considerado como uma maneira de formalizar e documentar todas as negociações estabelecidas entre o provedor de serviços e o consumidor.

Um SLA adequado deve conter diversas informações, tais como: serviços a serem disponibilizados, suporte a ser fornecido, desempenho esperado, disponibilidade, antecedência de aviso em caso de manutenção, relatórios a serem fornecidos com as informações do funcionamento do serviço de acordo com as métricas definidas, gerenciamento e recuperação do serviço em caso de falhas, decisão em caso de assuntos não previstos no contrato, penalidades previstas e quaisquer outras informações que se fizerem necessárias.

Segundo WOZEN [39], no Brasil, aproximadamente 59% das empresas já adotaram o conceito de SLA para a garantia do nível de serviços em TI. Já nos Estados Unidos, por exemplo, uma pesquisa realizada pela *PricewaterhouseCoopers* revelou que 85% das empresas já utilizam uma ferramenta para gestão automatizada de SLA, sendo que, deste universo, 76% são empresas do segmento de telecomunicações. A expectativa é que esta mesma tendência se repita no Brasil, com a adoção de SLAs pelas grandes operadoras do setor.

Existem duas tarefas bastante complexas cuja garantia de qualidade é de fundamental im-

portância para o sucesso de um SLA. Primeiro, é a geração do contrato propriamente dito. O nível de detalhamento é alto e os termos a serem utilizados devem ser precisos. O contrato deve ser o mais abrangente possível, a fim de atender e beneficiar tanto o consumidor quanto o prestador de serviço. Porém, apesar de ser uma tarefa difícil, os contratos, principalmente os que dizem respeito ao mesmo tipo de serviço prestado, normalmente são similares mudando apenas os níveis especificados e algumas características próprias de cada conjunto de prestador de serviço e consumidor. Com isso, ferramentas de auxílio para geração de SLA através de modelos foram desenvolvidas, sendo o SLA Toolkit [1] uma das mais comumente indicadas.

Englobando o SLA há o *Service Level Management* (SLM - gerenciamento dos níveis de serviço). As métricas que são definidas em um SLA precisam ser monitoradas de alguma forma para garantir que a qualidade especificada esteja sendo cumprida. Porém, o mercado de ferramentas automatizadas para tal fim ainda tem muito que evoluir. Muitas vezes, os relatórios feitos com os resultados obtidos pela prestação de serviços não são precisos já que necessitam de intervenção humana para serem gerados. No entanto, existe uma gama bastante grande de *softwares* que estão direcionando seus esforços neste sentido e a *PaQueT* também pode ser facilmente adaptada para suprir essas necessidades. No Apêndice A estão as consultas utilizadas nos experimentos do próximo capítulo, demonstrando apenas parte das métricas que podem ser obtidas facilmente através da *PaQueT*.

5.6 Resumo

Nesse capítulo foram mostrados os detalhes da ferramenta, desde sua arquitetura até sua implementação. Foi também exemplificado como os pacotes e as consultas devem ser manipulados. Por fim foi mostrado como uma ferramenta como a *PaQueT* pode ser importante em assuntos recorrentes atuais como é o caso do SLA. No próximo capítulo é feita a validação da ferramenta através de experimentos de monitoramento de redes realizados em computadores comuns.

CAPÍTULO 6

ESTUDO EXPERIMENTAL

Para explorar as funcionalidades disponíveis no Borealis, determinar a precisão dos resultados, e avaliar a carga de trabalho no sistema imposta pela *PaQueT*, foi realizado um estudo experimental. O objetivo do estudo foi principalmente o de validar a *PaQueT* como uma ferramenta de monitoramento de redes. Este capítulo apresenta o ambiente utilizado para executar os experimentos, os resultados dos primeiros testes obtidos com uma versão inicial da *PaQueT* [23], e em seguida dois experimentos para determinar a aplicabilidade e o impacto na ferramenta utilizando equipamentos bastante simples: o primeiro foi um teste de carga para determinar a capacidade de processamento da ferramenta a medida que o tráfego aumenta; o segundo se refere ao impacto no consumo de CPU conforme novas consultas vão sendo registradas.

6.1 Descrição do Ambiente

Os estudos iniciais foram executados em um computador com processador Intel Celeron 1.46 GHz com 512 MB de memória RAM e placa de rede Intel 100. Estes experimentos foram realizados apenas através de uma simulação com uma ferramenta de geração de pacotes randômicos. Os demais experimentos foram executados em dois computadores. O primeiro, responsável pela execução do *Borealis*, é um processador AMD Athlon 3000+ com 1 GB de memória RAM e uma placa de interface de rede 3COM 10/100/1000. O segundo computador, responsável pela execução do *IP Tool* e do *Dynamic Stream Interface*, é um processador Pentium 1.5 GHz com 512 MB de memória RAM e uma placa de rede Intel 100.

Durante os experimentos, o programa *top* do Linux foi utilizado para avaliar o tempo de CPU e o uso de memória física dos processos relativos a cada uma das ferramentas avaliadas. A taxa de atualização configurada foi de 3 segundos. Através destas informações foi possível obter o consumo de memória física e de tempo de CPU de cada um dos processos.

6.2 Estudos Iniciais

O objetivo dos primeiros experimentos foi não só mostrar a funcionalidade e a flexibilidade da ferramenta, como também determinar se os resultados gerados pela *PaQueT* eram precisos como os apresentados pelo *Ntop* [19] e *Wireshark* [12], que são ferramentas populares de monitoramento de redes. O objetivo da escolha das duas ferramentas foi a de tentar abranger tanto ferramentas de análise de protocolos de baixo nível, representadas pelo *Wireshark*, quanto ferramentas com foco nas estatísticas geradas, representadas pelo *Ntop*.

Existem diversas ferramentas de monitoramento de redes disponíveis. Uma lista bastante extensa destes sistemas pode ser encontrada em [32]. O *Wireshark* é uma ferramenta que possibilita verificar o conteúdo dos pacotes de diversos protocolos, além de permitir a aplicação de filtros e a visualização de estatísticas sobre os dados obtidos. Apesar de dar suporte a dezenas de protocolos, ela não possui muitas opções de sumarização das informações. Como alternativa, os resultados podem ser exportados para outros formatos, podendo ser analisados por outros aplicativos.

O *Ntop* é uma ferramenta para analisar o uso de uma rede, de forma similar ao que faz o comando *top* do Linux. Ela possui uma interface bastante amigável, sendo possível visualizar seus resultados pela *web*. Ela também dá suporte a diversos protocolos e interfaces de rede, e utiliza o conceito de *plugins* para adicionar novas funcionalidades à ferramenta. Vários gráficos mostrando o tráfego na rede podem ser gerados, e eles podem ser customizados pelo usuário, porém dentro de um escopo pré-definido.

Nestes experimentos, a *PaQueT* foi configurada para fazer a captura dos pacotes de modo promíscuo e para gerar os resultados em um arquivo. Porém, existem duas variações que poderiam ser utilizadas. A primeira refere-se à forma de captura. Se não houver suporte para o modo promíscuo, o monitoramento poderia ser feito em cada um dos pontos da rede isoladamente. Tal mudança requer apenas uma alteração na consulta, com a inclusão de um operador de união para capturar os pacotes de todos os pontos monitorados, como apresentado na Figura 5.4. Outra modificação que poderia ser feita é na forma de apresentação dos resultados. Ao invés de mostrá-los na tela e armazená-los em um arquivo, eles poderiam também ter sido inseridos em um banco de dados tradicional. Esta alteração também requer apenas a adição do

operador *table* no final da consulta, o qual recebe como parâmetro comandos SQL.

Para mostrar a funcionalidade e a flexibilidade da ferramenta foram escolhidas duas métricas para realizar os experimentos. O primeiro para registrar o número de pacotes por protocolo, o mesmo daquele descrito no diagrama da Figura 5.4 e no arquivo XML da Figura 5.5. O objetivo é fazer a contagem total de pacotes UDP e TCP que passaram pela rede durante uma janela de tempo de 60 segundos, sendo que as janelas que não contêm pacotes dos tipos monitorados são descartadas. A segunda métrica foi calcular a taxa de transmissão por IP. Este tipo de monitoramento é bastante útil para determinar os principais responsáveis do consumo da banda de uma rede. A consulta é feita de forma similar à anterior. Ela consiste de um agrupamento dos pacotes por IP do transmissor para fazer a soma do número total de bytes dos pacotes transmitidos. O objetivo é identificar a taxa de *upload* de cada usuário na rede. O mesmo poderia ser feito para obter a taxa de *download*.

O número de pacotes gerados em ambos os experimentos foi baixo, cerca de 210 mil pacotes durante um período de pouco mais de uma hora. Isto porque o objetivo dos experimentos foi de apenas validar os resultados para estudar a viabilidade para dar continuidade ao projeto da *PaQueT*. Os resultados obtidos pela *PaQueT* nos dois experimentos foram equivalentes àqueles gerados tanto pelo *Wireshark* quanto pelo *Ntop*.

Os resultados apresentados pelos estudos iniciais foram bastante positivos e indicaram que, apesar do SGSD Borealis ser um sistema de propósito geral para o processamento de *streams*, permite que seja desenvolvido uma extensão apenas para realizar o monitoramento de redes. Com esses resultados foi decidido continuar o desenvolvimento da *PaQueT*, tendo como diferencial a flexibilidade de customização da *PaQueT* sem necessidade de conhecimentos mais avançados de programação. Um ponto a ser ressaltado é a não utilização de recursos para armazenamento. Ao contrário das ferramentas consideradas no estudo experimental, que armazenam todas as informações necessárias para gerar todas as suas métricas, que são pré-definidas, na *PaQueT* a filtragem é realizada durante o monitoramento. Ela se baseia nas consultas registradas no sistema, diminuindo significativamente o volume de dados gerados e conseqüentemente a quantidade de armazenamento de informação.

6.3 Teste de Carga

Para este experimento, a consulta para contar o número de pacotes TCP e UDP por segundo foram executadas em três cenários distintos. A diferença entre eles consiste na forma como os pacotes são gerados e capturados. No primeiro cenário, denominado como *Arquivo*, os pacotes foram carregados a partir da leitura um arquivo de *dump* local. O segundo, denominado como *TcpReplay*, utilizou o tráfego gerado pela execução repetida do mesmo arquivo de *dump*, através do uso da ferramenta TcpReplay [36] em velocidade máxima; no terceiro e último cenário, a *PaQueT* foi executada em uma rede em ambiente real usando o Netcat[21] para gerar *burst* de tráfego TCP entre dois computadores; este cenário foi denominado de *Ambiente real*.

A Tabela 6.1 mostra os resultados dos três experimentos. Os valores correspondem a média de 10 execuções de cada um deles. O tamanho médio dos pacotes foi de 256 kB, porém durante os estudos a variação deste valor não demonstrou diferença durante nos resultados. A primeira linha da tabela mostra o tráfego em *megabits* por segundo (Mbps) enquanto a segunda linha representa o número de pacotes processados em *kilo* pacotes por segundo (kpps). Um ponto que chama atenção é a diferença da capacidade no caso do cenário *Arquivo*, enquanto nos outros dois os valores são similares. Na verdade, durante os experimentos do *TcpReplay*, a capacidade total na execução em velocidade máxima era praticamente a mesma quando configurada para executar em velocidades mais baixas. Após a análise dos resultados a conclusão foi a de que o uso da biblioteca *Pcap* é a principal limitação da *PaQueT*. Isto é, a capacidade de processamento da ferramenta foi limitada pelo número de pacotes que a *Pcap* é capaz de capturar. Para verificar basta comparar a segunda e a terceira linha da Tabela 6.1. A quarta linha mostra o número de pacotes que a *Pcap* informou ter descartado.

Tabela 6.1: Teste de carga da *PaQueT*.

Experimento	**Arquivo**	**TcpReplay**	**Ambiente real**
Tráfego (Mbps)	-	47	100[a]
PaQueT (kpps)	45	26	22
Recebidos pela Pcap (kpps)	-	26	22
Descartados pela Pcap (kpps)	-	35	12

[a] Taxa nominal

A limitação da Pcap explica também a enorme diferença nos resultados do cenário de *Ar-*

quivo, pois neste caso ao invés de fazer a captura dos dados, apenas estava sendo feita a leitura de um arquivo de *dump*. Portanto, é possível concluir que a *PaQueT* pode processar de forma precisa todos os pacotes capturados pela *Pcap*, mesmo a uma taxa de 45 kpps. Foi verificado também que na verdade, este é o limiar superior que o Borealis é capaz de processar. Aumentando um pouco esta taxa foi verificado que o Borealis não é capaz de processar todos os pacotes.

É importante ressaltar que os experimentos foram executados em um computador simples, sem fazer uso das características distribuídas do SGSD Borealis, como é o caso do *Dispersor de Carga*. Considerando o tamanho padrão do MTU, o qual especifica o tamanho máximo do pacote que uma rede pode transmitir, a *PaQueT* poderia dar suporte a transmissões de até 520 Mbps. Este resultado superou todas as expectativas, visto que tanto o SGSD *Borealis* quanto a *PaQueT* são apenas protótipos. Ainda mais se for considerado o fato de que a maioria dos protocolos de rede não possui taxas de transmissão tão altas, é possível dizer que a *PaQueT* é candidata em potencial para dar suporte a qualquer tipo de protocolo.

Em [18], é enfatizada a importância de modificar o pensamento em termos de velocidade para número de pacotes por segundo. Também é ressaltado o fato de que a capacidade de captura da *Pcap* depende de vários fatores, incluindo a plataforma na qual está sendo executada, a placa de rede utilizada assim como o seu respectivo *driver*. Durante os experimentos alguns parâmetros foram modificados com o objetivo de melhorar o desempenho, porém nenhum deles obteve resultados significativos. Melhorar o desempenho da *Pcap* ou usar *hardwares* especializados de alto custo são possibilidades que não estão no escopo deste trabalho. Até porque outras ferramentas de monitoramento baseadas em *Pcap*, como *TCPDump*[31], *Ntop* [19], e *Wireshark* [12],também apresentam a mesma limitação. Sendo assim isto não pode ser considerada uma desvantagem da ferramenta proposta. Os resultados gerados por essas ferramentas são bastante similares apesar de não poderem ser diretamente comparadas já que possuem modelos de monitoramento diferenciados. O *Ntop* armazena apenas um resumo das informações capturadas e as processa apenas quando o usuário deseja visualizar as métricas pré-definidas. O *Wireshark* armazena toda a informação capturada sem nenhum tipo de tratamento dos dados, o que possibilita obter quaisquer informação para posterior processamento. No entanto, o con-

sumo de recursos de armazenamento se torna muito alto. Já a *PaQueT* pode ser usada para gerar métricas arbitrárias em tempo real com um custo de armazenamento praticamente nulo.

O custo de execução da *PaQueT* nos três cenários descritos foi considerado baixo. O consumo de memória se manteve constante em 5% enquanto o consumo de CPU teve uma média de utilização de 15%, sofrendo alguns picos de consumo. No entanto, na próxima seção é avaliado o impacto de se adicionar novas consultas à ferramenta, visto que nesta seção foi utilizada uma única consulta bastante simples.

6.4 O Efeito do Aumento do Número de Consultas

Além de determinar a capacidade de tráfego processado pela *PaQueT* foi feito um experimento também para determinar o impacto no consumo de recursos do aumento do número de consultas sendo simultaneamente processadas.

Foram utilizadas consultas para obter métricas comuns do monitoramento de redes. Os detalhes de cada consulta são apresentadas no Apêndice B, sendo que algumas delas são listadas a seguir:

- Contagem do número de pacotes UDP e TCP;
- Taxa de transmissão IP;
- Média do tamanho dos pacotes trafegando pela rede;
- Número de bytes trocados por cada conexão.

Apesar de cada consulta possuir uma complexidade diferente, os resultados da Figura 6.1 mostram que o consumo de CPU não cresce linearmente com o número de consultas. Com 5 consultas registradas, a *PaQueT* consumiu cerca de 12% da capacidade de CPU; com 10 consultas, o consumo chegou a cerca de 22%, enquanto que com 20 consultas registradas o uso de CPU foi de aproximadamente 38%. Isto se deve ao fato de que o SGSD Borealis aplica técnicas de otimização no conjunto total de consultas, permitindo a reutilização dos resultados parciais que são comuns às consultas.

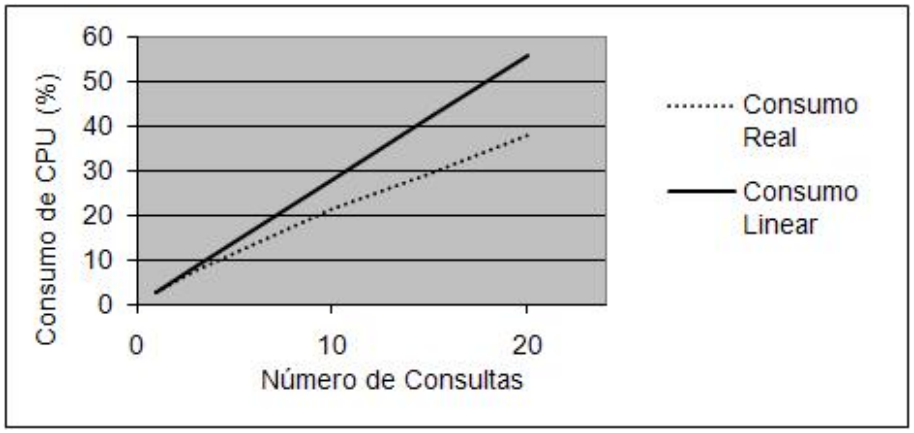

Figura 6.1: Número de consultas X consumo de CPU.

6.5 *PaQueT* em um Ambiente Distribuído

Apesar do desempenho ter superado as expectativas rodando o Borealis em apenas um processador, ele ainda pode ser melhorado fazendo uso dos módulos de Dispersor de Carga e Gerenciador de Carga. O objetivo destes dois módulos é de otimizar o desempenho do SGSD em momentos de grande atividade. Embora não fizesse parte do escopo deste estudo, foram feitos alguns testes preliminares da *PaQueT* utilizando as características distribuídas do Borealis. O objetivo dos testes foi o de mostrar as vantagens de se dar continuidade ao desenvolvimento e aperfeiçoamento da *PaQueT*.

Na versão atual do Borealis, existe um único coordenador que deve ser executado para poder gerenciar todas as decisões de otimização e coordenar os componentes locais que agem de forma passiva. Em cada nó, os componentes locais aguardam informações do coordenador, tanto para as responsabilidades do Gerenciador de carga, que consiste na realocação dos operadores das consultas, quanto nas responsabilidades do Dispersor de Carga, que descarta alguns dos pacotes que estão gerando sobrecarga. A utilização do Dispersor de Carga precisa ser consciente e bastante otimizada, visto que afeta diretamente a precisão dos resultados finais.

Esses testes preliminares, consistiram basicamente da execução de alguns dos testes descritos nas seções anteriores, porém executados em mais de um computador. Durante os testes foi verificado que os dois módulos respeitam as duas principais diretivas para otimização de sobrecarga: a primeira que é identificar tais situações e a segunda que é a capacidade de distribuir a carga entre os nós, e caso isso não seja possível, descartar algumas das informações para que os

resultados continuem sendo gerados mesmo sem garantia de precisão.

O coordenador do Gerenciador e do Dispersor de Carga possui uma série de parâmetros que podem ser configurados de acordo com cada sistema. Independentemente dos valores configurados, o seu funcionamento consiste na geração constante de estatísticas sobre a carga do sistema e no momento em que o limite superior é atingido, os operadores para distribuição e descarte dos pacotes são inseridos nas consultas. Este último experimento juntamente com os demais permitiu mostrar todo o potencial de uma ferramenta como a *PaQueT*.

CAPÍTULO 7

CONCLUSÃO

Através dos estudos experimentais, foi possível validar a *PaQueT* como uma ferramenta genérica de monitoramento de redes. Por ser uma ferramenta de alto nível, é possível construir diferentes consultas sem a necessidade da ajuda de desenvolvedores para a implementação de programas específicos para cada cenário. Esta abordagem permite fácil reutilização e adaptação de consultas previamente existentes. Além disso, ela permite que apenas os dados sumarizados sejam armazenados, caso se deseje consultá-los posteriormente.

O estudo experimental demonstrou a eficácia da ferramenta proposta, tanto em termos de funcionalidade, como em desempenho [24]. Além disso, uma das grandes vantagens da *PaQueT* é evitar o desperdício de armazenamento. Outro ponto importante é o fato de todas as consultas do Borealis passarem por um processo de otimização como nos bancos de dados tradicionais, permitindo melhorar seu desempenho de forma proporcional ao número de consultas registradas. Ou seja, quanto maior o número de consultas, melhor será a sua otimização, visto que alguns resultados parciais de uma consulta podem ser reaproveitados nas demais.

Outro ponto que deve ser ressaltado é quanto a abrangência das validações feitas com este estudo de caso. A escolha do Borealis como o SGSD a ser utilizado para implementação da ferramenta foi feita de acordo com a disponibilidade de SGSDs no momento da decisão. O Borealis poderia ser substituído na *PaQueT* por qualquer outro SGSD, bastando adaptar o módulo *Dynamic Stream Interface* para as necessidades do novo sistema. Além disso, não só uma ferramenta de monitoramento de redes como a *PaQueT* poderia ter sido desenvolvida, mas também outros aplicativos que atendessem os mercados descritos na seção 2.1, como é o caso das redes de sensores e do monitoramento de chamadas telefônicas.

7.1 Trabalhos Futuros

Existem diversas melhorias que podem ser feitas para complementar a *PaQueT*. Entre elas está a implementação de um analisador de resultados, bem como uma ferramenta para análise dos dados dos pacotes. Desta forma seria possível gerar estatísticas sobre os sites acessados, facilitando por exemplo uma melhor configuração de *firewalls*. Além disso, o suporte a outros protocolos da camada de transporte ou de rede pode ser facilmente adicionado através da definição das estruturas dos pacotes e extensão do esquema de entrada. Uma implementação também de grande utilidade seria conseguir manter as consultas registradas após reiniciar a ferramenta. Da forma como a ferramenta está implementada hoje, o usuário tem que registrar todas as consultas toda vez que ela for reinicializada. Outra grande contribuição seria a utilização da ferramenta *PaQueT* no dia-a-dia de um administrador de redes para que um profissional da área pudesse indicar as deficiências encontradas. Testes específicos para avaliar o impacto no desempenho da ferramenta fazendo uso dos módulos que distribuem o processamento e controlam os momentos de sobrecarga poderiam demonstrar ainda mais a eficiência da ferramenta. Por fim, a interface com o usuário é implementada também como um protótipo. Várias melhorias poderiam ser feitas para tornar o uso da ferramenta mais amigável, caso o usuário prefira não usar diretamente os arquivos em XML.

REFERÊNCIAS

[1] The sla toolkit. Disponível em: `http://www.service-level-agreement.net/`, 2003.

[2] Traderbot. `www.traderbot.com`, 2007.

[3] Daniel. J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çentintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, e Stan Zdonik. The design of the borealis stream processing engine. *Proceedings of the 2nd Conference on Classless Inter-Domain Routing (CIDR'05)*, páginas 277–289, 2005.

[4] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Jeong-Hyon Hwang, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, e Stan Zdonik. Aurora: A data stream management system. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, 2003.

[5] Daniel J. Abadi, Wolfgang Lindner, Samuel Madden, e Jörg Schuler. An integration framework for sensor networks and data stream management systems. *Proceedings of 30th International Conference on Very Large Data Bases (VLDB'04)*, páginas 1361–1364, 2004.

[6] Yanif Ahmad, Bradley Berg, Ugur Çetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, e Stan Zdonik. Distributed operation in the borealis stream processing engine. *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*, páginas 882–884, 2005.

[7] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, e Jennifer Widom. Stream: The stanford data stream management system. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[8] Magdalena Balazinska, Hari Balakrishnan, e Michael Stonebraker. Load management and high availability in the medusa distributed stream processing system. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, páginas 929–930, 2004.

[9] Bradley Berg. A distributed catalog for the borealis stream processing engine. `www.cs.brown.edu/research/borealis/public/`, 2006.

[10] Borealis Team. *Borealis Application Programmer's Guide*, 2006.

[11] Borealis Team. *Borealis Developer's Guide*, 2006.

[12] Cace Technologies. *Wireshark*, 2007.

[13] Tim Carstens. Programming with pcap. `www.tcpdump.org/pcap.htm`, 2002.

[14] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, e Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, páginas 269–280, 2003.

[15] Cisco Systems Inc. *Introduction to Cisco IOS Netflow - A Technical Overview*, 2006.

[16] Chuck Cranor, Theodore Johnson, Oliver Spatscheck, e Vladislav Shkapenyuk. The gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[17] André L. L. de Aquino, Carlos M. S. Figueiredo, Eduardo F. Nakamura, Luciana S. Buriol, Antonio A. F. Loureiro, Antonio Otávio Fernandes, e Claudionor J. N. Jr Coelho. Data stream based algorithms for wireless sensor network applications. *AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, páginas 869–876, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Luca Deri. Improving passive packet capture: Beyond device polling. Relatório técnico, Information Society Technologies, 2007.

[19] Luca Deri e Stefano Suin. Effective traffic measurement using ntop. *IEEE Communications Magazine*, 38(5):138–143, 2000.

[20] Ramez Elmasri e Shamkant B. Navathe. *Sistemas de Banco de Dados*. Pearson - Addison Wesley, 2005.

[21] The GNU Netcat project. *Netcat*, 2007.

[22] Nick Koudas e Divesh Srivastava. Data stream query processing: A tutorial. *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03)*, páginas 1149–1149, 2003.

[23] Natascha Petry Ligocki e Carmem Satie Hara. Uma ferramenta de monitoramento de redes usando sistemas gerenciadores de streams de dados. *Workshop de Gerência e Operações de Redes e Serviços (WGRS'07)*, 2007.

[24] Natascha Petry Ligocki, Christian Lyra, e Carmem Satie Hara. A flexible network monitoring tool based on a data stream management system. *IEEE Symposium on Computers and Communications (ISCC'2008)*, 2008.

[25] Alexandre A. B. Lima, Marta L. Q. Matoso, e Cláudio Esperança. Efficient processing of heavy-weight queries in database clusters. Relatório Técnico 001, UFRJ, 2003.

[26] David L. Mills. Network time protocol distribution. `www.ntp.org`, 2008.

[27] Dimitrios Miras, Amela Sadagic, Ben Teitelbaum, Jason Leigh, Magda El Zarki, e Haining Liu. *A Survey on Network QoS Needs of Advanced Internet Applications*. Internet2 - QoS Working Group, 2002.

[28] Pandia Search Engine News. The size of the world wide web. Disponível em: `http://www.pandia.com/`, 2007.

[29] P. Phaal, S. Panchen, e N. McKee. RFC 3176: InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, Setembro de 2001.

[30] Thomas Plagemann, Vera Goebel, Andrea Bergamini, Giacomo Tolu, Guillaume Urvoy-Keller, e Ernst W. Biersack. Using data stream management systems for traffic analysis - a case study. *Proceedings of the 5th International Workshop on Passive and Active Network Measurement (PAM'04)*, páginas 215–226, 2004.

[31] SANS Institute. *TCP/IP and tcpdump. Pocket reference guide.*, 2007.

[32] Network monitoring tools. `www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html`, 2007.

[33] William Stallings. *SNMP, SNMPv2 and RMON - Practical Network Management, 2nd Ed.* Addison-Wesley, 1996.

[34] E. Stephan. Rfc4148: Ip performance metrics (ippm) metrics registry. `http://tools.ietf.org/html/rfc4148`, 2005.

[35] Michael Stonebraker, Ugur Çetintemel, e Stan Zdonik. The 8 requirements of real-time stream processing. *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, 2005.

[36] TcpReplay. *TcpReplay Online Manual*, 2007.

[37] Traderbot Technologies. Data stream systems, 2002.

[38] André Ormastroni Victor e Marta L. Q. Mattoso. Distributed query routing in a cluster of autonomous databases. Relatório Técnico 002, UFRJ, 2003.

[39] Wozen. O que é service level agreement. Disponível em: `http://www.ccs.com.br/`, 2006.

[40] Yin Zhang, Lee Breslau, Vern Paxson, e Scott Shenker. On the characteristics and origins of internet flow rates. *Proceedings of Conference on Applications, technologies, architec-*

tures, and protocols for computer communications (ACM SIGCOMM'02), páginas 309 – 322, 2002.

APÊNDICE A

CONSULTAS

```
========================================
  Número de pacotes por tipo: UDP e TCP
========================================
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">
<borealis>
   <input stream="Pacote" schema="TuplaPacote"/>

   <schema name="TuplaPacote">
      <field name="time" type="int"/>
      <field name="ip_p" type="int"/>
   </schema>

   <query name="NumPacotesUdpTcp">
      <box name="Filtro" type="filter">
         <in stream="Pacote"/>
         <out stream="Filtro"/>

         <parameter name="expression.0"
                     value="ip_p==6 || ip_p=17"/>
         <parameter name="pass-on-false-port" value="0"/>
      </box>
      <box name="Contagem" type="aggregate">
         <in stream="Filtro"/>
         <out stream="Agregacao"/>

         <parameter name="aggregate-function.0"
                                        value="count()"   />
         <parameter name="aggregate-function-output-name.0"
                                        value="numPacotes"/>
         <parameter name="window-size-by" value="VALUES"    />
         <parameter name="window-size"    value="60"        />
         <parameter name="advance"        value="60"        />
         <parameter name="order-by"       value="FIELD"     />
         <parameter name="order-on-field" value="time"      />
         <parameter name="group-by"       value="ip_p"      />
         <parameter name="drop-empty-outputs" value="1"     />
      </box>
   </query>
</borealis>

==============================
  Taxa de transmissão por IP
```

```
================================
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">

<borealis>
   <input   stream="Pacote"    schema="TuplaPacote" />

   <schema name="TuplaPacote">
      <field  name="time"      type="int" />
      <field  name="ip_len"    type="int"/>
      <field  name="ip_src"    type="string"  size="15" />
   </schema>

   <query name="Traffic" >
      <box name="Soma"   type="aggregate" >
         <in    stream="Pacote" />
         <out   stream="Aggregate" />

         <parameter   name="aggregate-function.0"
                                  value="sum(ip_len)" />
         <parameter   name="aggregate-function-output-name.0"
                                  value="traffic" />
         <parameter   name="window-size-by"   value="VALUES" />
         <parameter   name="window-size"      value="60"     />
         <parameter   name="advance"          value="60"     />
         <parameter   name="order-by"         value="FIELD"  />
         <parameter   name="order-on-field"   value="time"   />
         <parameter   name="group-by"         value="ip_src" />
         <parameter   name="independent-window-alignment"
                                              value="1" />
         <parameter   name="drop-empty-outputs" value="1"    />
      </box>
   </query>
</borealis>

==================================
  Descarte de metade dos pacotes
==================================
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">

<borealis>
   <input   stream="Pacote"   schema="TuplaPacote"  />

   <schema name="TuplaPacote">
      <field  name="time"     type="int" />
      <field  name="ip_len"   type="int" />
      <field  name="ip_src"   type="string"  size="15" />
```

```
   </schema>
   <box name="Descarte" type="random_drop">
      <in stream="Pacote" />
      <out stream="Resultado" />

      <parameter name="drop_rate" value="0.5" />
      <parameter name="max_batch_size" value="1000" />
</box>

==========================================
  Média do tamanho dos pacotes trafegando
==========================================
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">

<borealis>
   <input   stream="Pacote"  schema="TuplaPacote" />

   <schema name="TuplaPacote">
      <field  name="time"    type="int" />
      <field  name="ip_len"  type="int" />
      <field  name="ip_src"  type="string"  size="15" />
   </schema>

   <query>
      <box name="Soma" type="aggregate" >
         <in stream="Pacote" />
         <out stream="TotalPorMinuto" />

         <parameter name="aggregate-function.0"
                                    value="count()"    />
         <parameter name="aggregate-function-output-name.0"
                                    value="contagem"   />
         <parameter name="aggregate-function.1"
                                    value="sum(ip_len)"/>
         <parameter name="aggregate-function-output-name.1"
                                    value="total"      />
         <parameter name="window-size-by" value="VALUES" />
         <parameter name="window-size" value="60"        />
         <parameter name="advance" value="60"            />
         <parameter name="order-by" value="FIELD"        />
         <parameter name="order-on-field" value="time"   />
      </box>
      <box name="Media" type="map">
         <in  stream="TotalPorMinuto" />
         <out stream="MediaPorMinuto" />

         <parameter name="expression.0" value="time" />
```

```
         <parameter name="output-field-name.0" value="time" />
         <parameter name="expression.1" value="contagem/60" />
         <parameter name="output-field-name.1" value="count"/>
         <parameter name="expression.2" value="total*8/60"  />
         <parameter name="output-field-name.2" value="media"/>
      </box>
   </query>
</borealis>

==================================
  Bytes trocados em cada conexão
==================================
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">

<borealis>
   <input   stream="Pacote"     schema="TuplaPacote" />

   <schema name="TuplaPacote">
      <field  name="time"      type="int"  />
      <field  name="ip_len"    type="int"  />
      <field  name="ip_src"    type="string"  size="15" />
      <field  name="ip_dst"    type="string"  size="15" />
      <field name="tcp_sport"  type="int"  />
      <field name="tcp_dport"  type="int"  />
      <field name="tcp_seq"    type="long" />
      <field name="tcp_ack"    type="long" />
   </schema>

   <query name="Query" >
      <box name="filtro" type="filter">
         <in  stream="Pacote"   />
         <out stream="Esquerda" />
         <out stream="Direita"  />

         <parameter name="expression.0"
                          value="tcp_seq &gt; tcp_ack" />
         <parameter name="pass-on-false-port" value="1" />
      </box>
      <box name="box1" type="aggregate" >
         <in stream="Esquerda" />
         <out stream="SomaEsquerda" />

         <parameter name="aggregate-function.0"
                                  value="sum(ip_len)" />
         <parameter name="aggregate-function-output-name.0"
                                  value="bytes" />
         <parameter name="aggregate-function.1"
```

```
                                value="count()" />
   <parameter name="aggregate-function-output-name.1"
                                value="contagem" />
   <parameter name="window-size-by" value="VALUES" />
   <parameter name="window-size" value="10" />
   <parameter name="advance" value="10" />
   <parameter name="order-by" value="FIELD" />
   <parameter name="order-on-field" value="time" />
   <parameter name="group-by"
        value="ip_src,tcp_sport,ip_dst,tcp_dport" />
</box>
<box name="box2" type="aggregate" >
   <in stream="Direita" />
   <out stream="SomaDireita" />

   <parameter name="aggregate-function.0"
                                  value="sum(ip_len)" />
   <parameter name="aggregate-function-output-name.0"
                                  value="bytes" />
   <parameter name="aggregate-function.1"
                                  value="count()" />
   <parameter name="aggregate-function-output-name.1"
                                  value="contagem" />
   <parameter name="window-size-by" value="VALUES" />
   <parameter name="window-size" value="10" />
   <parameter name="advance" value="10" />
   <parameter name="order-by" value="FIELD" />
   <parameter name="order-on-field" value="time" />
   <parameter name="group-by"
         value="ip_src,tcp_sport,ip_dst,tcp_dport" />
</box>
<box name="Total" type="join" >
   <in stream="SomaEsquerda " />
   <in stream="SomaDireita " />
   <out stream="Todos" />

   <parameter name="predicate"
               value="left.ip_src == right.ip_dst and
                  left.tcp_sport == right.tcp_dport and
                  left.ip_dst == right.ip_src and
                  left.tcp_dport == right.tcp_sport and
                  left.time == right.time" />
   <parameter name="left-buffer-size" value="1"      />
   <parameter name="left-order-by" value="VALUES"    />
   <parameter name="left-order-on-field" value="time"/>
   <parameter name="right-buffer-size" value="1"     />
   <parameter name="right-order-by" value="VALUES"   />
   <parameter name="right-order-on-field" value="time"/>
```

```
        <parameter name="out-field-name.0"
                                    value="esq_ip_src"  />
        <parameter name="out-field.0" value="left.ip_src" />
        <parameter name="out-field-name.1"
                                    value="esq_src_port"/>
        <parameter name="out-field.1" value="left.tcp_sport"/>
        <parameter name="out-field-name.2"
                                    value="esq_bytes"   />
        <parameter name="out-field.2" value="left.bytes"  />
        <parameter name="out-field-name.3"
                                    value="dir_ip_src"  />
        <parameter name="out-field.3" value="right.ip_src"/>
        <parameter name="out-field-name.4"
                                    value="dir_src_port"/>
        <parameter name="out-field.4"
                                  value="right.tcp_sport"/>
        <parameter name="out-field-name.5"
                                    value="dir_bytes"   />
        <parameter name="out-field.5" value="right.bytes" />
        <parameter name="out-field-name.6"
                                    value="time" />
        <parameter name="out-field.6" value="right.time"  />
      </box>
      <box name="resultbox" type="map" >
          <in stream="Todos" />
          <out stream="Resultado" />

          <parameter name="expression.0" value="esq_ip_src"  />
          <parameter name="output-field-name.0"
                                      value="esq_ip_src"  />
          <parameter name="expression.1" value="esq_src_port"/>
          <parameter name="output-field-name.1"
                                      value="esq_src_port"/>
          <parameter name="expression.2" value="dir_ip_src"  />
          <parameter name="output-field-name.2"
                                      value="dir_ip_src"  />
          <parameter name="expression.3" value="dir_src_prt" />
          <parameter name="output-field-name.3"
                                      value="dir_src_port"/>
          <parameter name="expression.4" value="time" />
          <parameter name="output-field-name.4"
                                      value="time" />
          <parameter name="expression.5"
                              value="esq_bytes + dir_bytes" />
          <parameter name="output-field-name.5" value="bytes"/>
      </box>
   </query>
</borealis>
```

APÊNDICE B

CÓDIGO FONTE

B.1 *IP Tool*

```
//============================
// Name        : IPTool.h
// Version     : 1.0
//============================

#ifndef IPTOOL_H_
#define IPTOOL_H_

#include <map>
#include <vector>
#include <string>
#include <fstream>
#include <arpa/inet.h>

// default snap length (maximum bytes per packet to capture)
const int SNAP_LEN = 1518;
// ethernet headers are always exactly 14 bytes
const int SIZE_ETHERNET = 14;
// Ethernet addresses are 6 bytes
const int ETHER_ADDR_LEN = 6;

class IPTool {
public:
   IPTool(const std::string &itf);
   ~IPTool();

   static void *open_connection(void *data);
   static void *add_listeners(void *data);

private:
   std::string interface_;
   std::vector<int> listeners;
   std::vector<int> wantedFields;
   std::map<std::string,int> fieldMap;
   std::ofstream ofs;

   static void sigint_handler(int sig);
   int getWantedFields(std::string str);

   // Ethernet header
```

```
 struct sniff_ethernet {
    char  ether_dhost[ETHER_ADDR_LEN];
    char  ether_shost[ETHER_ADDR_LEN];
    u_short ether_type;
 };

 // IP header
 struct sniff_ip {
    u_char  ip_vhl;
    u_char  ip_tos;
    u_short ip_len;
    u_short ip_id;
    u_short ip_off;
    u_char  ip_ttl;
    u_char  ip_p;
    u_short ip_sum;
    struct  in_addr ip_src,ip_dst;
 };
 #define IP_HL(ip) (((ip).ip_vhl) & 0x0f)

 // TCP header
 struct sniff_tcp {
    u_short tcp_sport;
    u_short tcp_dport;
    u_int tcp_seq;
    u_int tcp_ack;
    u_char  tcp_off;
    u_char  tcp_flags;
    u_short tcp_win;
    u_short tcp_sum;
    u_short tcp_urp;
 };
 #define TH_OFF(th) (((th).tcp_off & 0xf0) >> 4)

struct sniff_udp {
    u_short udp_sport;
    u_short udp_dport;
    u_short udp_len;
    u_short udp_sum;
};

struct sniff_packet {
    struct sniff_ethernet ether_header;
    struct sniff_ip ip_header;
    struct sniff_tcp tcp_header;
    struct sniff_udp udp_header;
};
```

```
   void got_packet(u_char *args,
                   const struct pcap_pkthdr *header,
                   const u_char *packet);
   void pack_send_fields(const struct sniff_packet *headers);
};
#endif

//============================
// Name        : IPTool.cpp
// Version     : 1.0
//============================

#include "IPTool.h"
#include <pcap.h>
#include <fcntl.h>
#include <signal.h>
#include <iostream>
#include <sstream>

namespace
{
   const time_t time0 = time(NULL);
   const char * IPTOOL = "/tmp/IPToolRegister";

   struct thread_args
   {
      IPTool* ipt;
      thread_args(IPTool* t) :ipt(t) {}
   };
}

namespace packet
{
   const int time          = 0x00000001;

   // destination host address
   const int ether_dhost   = 0x00000002;
   // source host address
   const int ether_shost   = 0x00000004;
   // IP? ARP? RARP? ...
   const int ether_type    = 0x00000008;

   // version << 4 | header length >> 2
   const int ip_vhl        = 0x00000010;
   // type of service
   const int ip_tos        = 0x00000020;
   // total length
   const int ip_len        = 0x00000040;
```

```
    // identification
    const int ip_id         = 0x00000080;
    // fragment offset field
    const int ip_off        = 0x00000100;
    // time to live
    const int ip_ttl        = 0x00000200;
    // protocol
    const int ip_p          = 0x00000400;
    // checksum
    const int ip_sum        = 0x00000800;
    // source and destination addresses
    const int ip_src        = 0x00001000;
    const int ip_dst        = 0x00002000;
    // source and destination ports
    const int tcp_sport     = 0x00004000;
    const int tcp_dport     = 0x00008000;
    // sequence number
    const int tcp_seq       = 0x00010000;
    // acknowledgement number
    const int tcp_ack       = 0x00020000;
    // data offset, rsvd
    const int tcp_off       = 0x00040000;
    const int tcp_flags     = 0x00080000;
    // window
    const int tcp_win       = 0x00100000;
    // checksum
    const int tcp_sum       = 0x00200000;
    // urgent pointer
    const int tcp_urp       = 0x00400000;

    // source and destination ports
    const int udp_sport     = 0x00800000;
    const int udp_dport     = 0x01000000;
    // datagram length
    const int udp_len       = 0x02000000;
    // checksum
    const int udp_sum       = 0x04000000;
}

IPTool::IPTool(const std::string &itf) : interface_(itf)
{
    signal(SIGINT, &IPTool::sigint_handler);

    fieldMap["time"]        = packet::time;
    fieldMap["ether_dhost"] = packet::ether_dhost;
    fieldMap["ether_shost"] = packet::ether_shost;
    fieldMap["ether_type"]  = packet::ether_type;
    fieldMap["ip_vhl"]      = packet::ip_vhl;
```

```
   fieldMap["ip_tos"]      = packet::ip_tos;
   fieldMap["ip_len"]      = packet::ip_len;
   fieldMap["ip_id"]       = packet::ip_id;
   fieldMap["ip_off"]      = packet::ip_off;
   fieldMap["ip_ttl"]      = packet::ip_ttl;
   fieldMap["ip_p"]        = packet::ip_p;
   fieldMap["ip_sum"]      = packet::ip_sum;
   fieldMap["ip_src"]      = packet::ip_src;
   fieldMap["ip_dst"]      = packet::ip_dst;
   fieldMap["tcp_sport"]   = packet::tcp_sport;
   fieldMap["tcp_dport"]   = packet::tcp_dport;
   fieldMap["tcp_seq"]     = packet::tcp_seq;
   fieldMap["tcp_ack"]     = packet::tcp_ack;
   fieldMap["tcp_off"]     = packet::tcp_off;
   fieldMap["tcp_flags"]   = packet::tcp_flags;
   fieldMap["tcp_win"]     = packet::tcp_win;
   fieldMap["tcp_sum"]     = packet::tcp_sum;
   fieldMap["tcp_urp"]     = packet::tcp_urp;
   fieldMap["udp_sport"]   = packet::udp_sport;
   fieldMap["udp_dport"]   = packet::udp_dport;
   fieldMap["udp_len"]     = packet::udp_len;
   fieldMap["udp_sum"]     = packet::udp_sum;
}

IPTool::~IPTool()
{
   for (std::vector<int>::iterator it = listeners.begin();
                          it != listeners.end(); it++)
      close(*it);
}

void IPTool::sigint_handler(int sig)
{
   exit(0);
}

int IPTool::getWantedFields(std::string str)
{
   int wanted = 0;

   std::replace(str.begin(), str.end(), ',', ' ');
   std::istringstream iss(str.c_str());
   if (iss) {
      std::string field;
      while (iss >> field) {
         std::map<std::string,int>::iterator
                                  it = fieldMap.find(field);
         if (it!=fieldMap.end())
```

```
            wanted+=it->second;
         else
            std::cout << "Field not found: " <<
                                       field << std::endl;
      }
   }
   return wanted;
}

void *IPTool::open_connection(void *data)
{
   thread_args *args= static_cast<thread_args*>(data);
   IPTool *ipt = args->ipt;

   char errbuf[PCAP_ERRBUF_SIZE]= {0};
   pcap_t *handle;
   struct pcap_pkthdr *header;
   const u_char *packet;

   if (!strcmp(ipt->interface_.c_str(), "file")) {
      while (true) {
         handle = pcap_open_offline("dump", errbuf);
         if (handle == NULL)
            fprintf(stderr, "Couldn't open file %s:\n",errbuf);
         else {
            while (pcap_next_ex(handle, &header, &packet) > 0)
               ipt->got_packet(0, header, packet);
            pcap_close(handle);
         }
         sleep(1);
      }
   }
   else {
      handle = pcap_open_live(ipt->interface_.c_str(),
                                    BUFSIZ, 1, 128, errbuf);
      if (handle == NULL)
         fprintf(stderr, "Couldn't open device %s:\n",errbuf);
      else {
         while (pcap_next_ex(handle, &header, &packet) > 0)
            ipt->got_packet(0, header, packet);
         pcap_close(handle);
      }
   }
   delete args;
   pthread_exit(NULL);
}

void IPTool::add_listeners()
```

```
{
   std::string str;

   while (true) {
      std::ifstream ifs(IPTOOL);
      ifs >> str;
      std::cout << str << std::endl;
      if (access(str.substr(0, str.find(";")).c_str(),
                                          F_OK) == -1)
          mkfifo(str.substr(0, str.find(";")).c_str(), 0666);
      int fd = open(str.substr(0, str.find(";")).c_str(),
                                          O_WRONLY);
      if (fd != -1) {
         listeners.push_back(fd);
         wantedFields.push_back(getWantedFields(
                           str.substr(str.find(";")+1)));
      }
      else
         std::cout << "Could not open pipe: " << std::endl;
   }
}

void IPTool::got_packet(u_char *args,
      const struct pcap_pkthdr *header, const u_char *packet)
{
   struct sniff_packet *headers = {0};
   headers = new sniff_packet;
   int size_ip = 0;
   int size_tcp = 0;
   int size_udp = 0;

   memcpy(&(headers->ether_header), packet,
                              sizeof (struct sniff_ethernet));
   memcpy(&(headers->ip_header), packet + SIZE_ETHERNET,
                              sizeof (struct sniff_ip));
   size_ip = IP_HL(headers->ip_header)*4;
   if (size_ip < 20) // Invalid IP header length
      return;

   switch(headers->ip_header.ip_p)
   {
      case IPPROTO_TCP:
         memcpy(&(headers->tcp_header), packet +
           SIZE_ETHERNET + size_ip, sizeof (struct sniff_tcp));
         size_tcp = TH_OFF(headers->tcp_header)*4;
         if (size_tcp < 20) // Invalid TCP header length
            return;
         else
```

```
            break;
      case IPPROTO_UDP:
         memcpy(&(headers->udp_header), packet +
           SIZE_ETHERNET + size_ip, sizeof (struct sniff_udp));
         size_udp = headers->udp_header.udp_len*4;
         if (size_udp < 20) // Invalid UDP header length
            return;
         else
            break;
      default:
         break;
//this part can be extended to sniffer/analyse other streams
   }
    pack_send_fields(headers);
}

void IPTool::pack_send_fields
            (const struct sniff_packet *headers)
{
   long timestamp = (long)difftime(time(NULL), time0);
   if ( timestamp < 0 ) timestamp = 0;
   long l = 0;
   long ll = 0;
   for (unsigned i = 0; i < listeners.size(); i++) {
      char data[1024]={0};
      int wanted = wantedFields[i];
      memset(data, 0, sizeof data);
      char *buf = &data[0];

      if (wanted & packet::time) {
         memcpy(buf, &timestamp, sizeof timestamp);
         buf += sizeof timestamp;
      }
      if (wanted & packet::ether_dhost) {
         memcpy(buf, headers->ether_header.ether_dhost,
                  sizeof headers->ether_header.ether_dhost);
         buf += sizeof headers->ether_header.ether_dhost;
      }
      if (wanted & packet::ether_shost) {
         memcpy(buf, headers->ether_header.ether_shost,
                  sizeof headers->ether_header.ether_shost);
         buf += sizeof headers->ether_header.ether_shost;
      }
      if (wanted & packet::ether_type) {
         memcpy(buf, &headers->ether_header.ether_type,
                  sizeof headers->ether_header.ether_type);
         buf += sizeof headers->ether_header.ether_type;
      }
```

```
if (wanted & packet::ip_vhl) {
   memcpy(buf, &headers->ip_header.ip_vhl,
            sizeof headers->ip_header.ip_vhl);
   buf += sizeof headers->ip_header.ip_vhl;
}
if (wanted & packet::ip_tos) {
   memcpy(buf, &headers->ip_header.ip_tos,
            sizeof headers->ip_header.ip_tos);
   buf += sizeof headers->ip_header.ip_tos;
}
if (wanted & packet::ip_len) {
   l = headers->ip_header.ip_len;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::ip_id) {
   memcpy(buf, &headers->ip_header.ip_id,
            2*sizeof headers->ip_header.ip_id);
   buf += 2*sizeof headers->ip_header.ip_id;
}
if (wanted & packet::ip_off) {
   memcpy(buf, &headers->ip_header.ip_off,
            2*sizeof headers->ip_header.ip_off);
   buf += 2*sizeof headers->ip_header.ip_off;
}
if (wanted & packet::ip_ttl) {
   memcpy(buf, &headers->ip_header.ip_ttl,
            sizeof headers->ip_header.ip_ttl);
   buf += sizeof headers->ip_header.ip_ttl;
}
if (wanted & packet::ip_p) {
   l = headers->ip_header.ip_p;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::ip_sum) {
   l = headers->ip_header.ip_sum;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::ip_src) {
   memcpy(buf, inet_ntoa(headers->ip_header.ip_src), 15);
   buf += 15;
}
if (wanted & packet::ip_dst) {
   memcpy(buf, inet_ntoa(headers->ip_header.ip_dst), 15);
   buf += 15;
}
```

```
if (wanted & packet::tcp_sport) {
   l = headers->tcp_header.tcp_sport;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::tcp_dport) {
   l = headers->tcp_header.tcp_dport;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::tcp_seq) {
   l = headers->tcp_header.tcp_seq;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::tcp_ack) {
   ll = headers->tcp_header.tcp_ack;
   memcpy(buf, &ll, sizeof ll);
   buf += sizeof ll;
}
if (wanted & packet::tcp_off) {
   ll = headers->tcp_header.tcp_off;
   memcpy(buf, &ll, sizeof ll);
   buf += sizeof ll;
}
if (wanted & packet::tcp_flags) {
   memcpy(buf, &headers->tcp_header.tcp_flags,
            sizeof headers->tcp_header.tcp_flags);
   buf += sizeof headers->tcp_header.tcp_flags;
}
if (wanted & packet::tcp_win) {
   memcpy(buf, &headers->tcp_header.tcp_win,
            2*sizeof headers->tcp_header.tcp_win);
   buf += 2*sizeof headers->tcp_header.tcp_win;
}
if (wanted & packet::tcp_sum) {
   l = headers->tcp_header.tcp_sum;
   memcpy(buf, &l, sizeof l);
   buf += sizeof l;
}
if (wanted & packet::tcp_urp) {
   memcpy(buf, &headers->tcp_header.tcp_urp,
            2*sizeof headers->tcp_header.tcp_urp);
   buf += 2*sizeof headers->tcp_header.tcp_urp;
}
if (wanted & packet::udp_sport) {
   l = headers->udp_header.udp_sport;
   memcpy(buf, &l, sizeof l);
```

```
            buf += sizeof l;
         }
         if (wanted & packet::udp_dport) {
            l = headers->udp_header.udp_dport;
            memcpy(buf, &l, sizeof l);
            buf += sizeof l;
         }
         if (wanted & packet::udp_len) {
            l = headers->udp_header.udp_len;
            memcpy(buf, &l, sizeof l);
            buf += sizeof l;
         }
         if (wanted & packet::udp_sum) {
            l = headers->udp_header.udp_sum;
            memcpy(buf, &l, sizeof l);
            buf += sizeof l;
         }
         if (write(listeners[i], data, buf - &data[0]) == -1) {
            std::cout << "Unregistering listener "
                      << listeners[i];
            close(listeners[i]);
            listeners.erase(listeners.begin()+i);
            wantedFields.erase(wantedFields.begin()+i);
         }
      }
   }

   int main(int argc, const char *argv[])
   {
      int thr_id;
      pthread_t  p_thread;

      // create a new thread that will capture packets
      if (argc == 2) {
         mkfifo(IPTOOL, 0666);
         class IPTool *ipt = new IPTool(argv[1]);
         thr_id = pthread_create(&p_thread, NULL,
            &IPTool::open_connection, new thread_args(&ipt));
         std::cout << "Waiting listeners... " << std::endl;
         ipt->add_listeners();
         delete ipt;
      }
      else
            std::cout <<
               "USAGE: IPTool [<interface>]...";

      return EXIT_SUCCESS;
   }
```

B.2 *Dynamic Stream Interface*

```
//============================
// Name        : DSI.cpp
// Version     : 1.0
//============================

#include "RegisterQuery.h"
#include <vector>
#include <fstream>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <sys/socket.h>

BOREALIS_NAMESPACE_BEGIN

map<string, CatalogSchema> PaQueTDiagram::_output_schema;

const char *IPTOOL = "/tmp/IPToolRegister";

void sigint_handler(int sig)
{
   exit(0);
}

int main(int argc, const char *argv[])
{
   if (argc < 2 || argc > 3) {
      NOTICE <<
        "USAGE: extension <xml file> [deploy_xml file] ...";
      return -1;
   }

   signal(SIGINT, sigint_handler);

   RegisterQuery  diagram;
   // Relative path of the XML source
   string    xml_file;
   string    deploy_xml;
   char   buf[16] = {0} ;

   sprintf(buf, "/tmp/%d", getpid());
   std::string pidfile = buf;
   xml_file = to_string(argv[1]);
   NOTICE << "extension xml: " << xml_file;
   if (argc == 3) {
      deploy_xml = to_string(argv[2]);
```

```
      NOTICE << "extension depoly xml:  " << deploy_xml;
   }
   if (!diagram.parse_file(xml_file) ||
                          !diagram.infer_schema())
      return -2;

   if (diagram.open(xml_file, deploy_xml))
      WARN << "Could not deploy the network";
   else {
      std::string registerClient = pidfile;
      registerClient += ";";
      std::vector<SchemaField> fields =
                  diagram._input_schema.get_schema_field();
      for(std::vector<SchemaField>::iterator it =
                  fields.begin(); it != fields.end(); it++) {
         registerClient += (*it).get_name();
         registerClient += ",";
      }
      std::ofstream ofs(IPTOOL);
      if (ofs.is_open()) {
         mkfifo(pidfile.c_str(), 0666);
         ofs << registerClient;
         ofs.close();

         diagram.setSource(open(pidfile.c_str(), O_RDONLY));
         // Send the first batch of tuples
         //Queue up the next round
         diagram.sentPacket();
         DEBUG  << "run the client event loop...";
         diagram.runClient();
      }
   }
   diagram.terminateClient();
   return 0;
}
BOREALIS_NAMESPACE_END

//================================
// Name        : RegisterQuery.h
// Version     : 1.0
//================================

#ifndef REGISTERQUERY_H
#define REGISTERQUERY_H

#include  "PaQueTDiagram.h"
#include  "MedusaClient.h"
#include  "TupleHeader.h"
```

```
BOREALIS_NAMESPACE_BEGIN

class RegisterQuery : public PaQueTDiagram
{
public:
   RegisterQuery() { fd = -1; }

   struct  Packet : public TupleHeader
   {
      char _data[1024];
   } __attribute__((__packed__));

   struct  OutputTuple
   {
      string str;
   };

   Status  register_query(string  xml_file );

   // Activate the front-end and subscribe to streams.
   int32  open(string xml_file, string deploy_xml);
   void  setSource(int source);
   void  runClient();
   void  terminateClient();

   // It is called after sendPacket is done and a pause.
   void  sentPacket();
   // Enque a Packet for input.
   void  batchPacket(Packet  *tuple);
   // Send enqued Packet inputs.
   void  sendPacket(uint32  sleep);

private:
   void  connectPacket();
   void  delayPacket();
   // Handler to dispatch tuples received.
   static Status outputHandler(ptr<StreamEvent> event);
   // It is called after a tuple is received.
   static void  receivedOutput
                     (OutputTuple *tuple, string output);
   // Subscribe to output stream.
   void  subscribeOutput();
   // Launch the Borealis front-end.
   static int32  launchDiagram
                        (string  xml, string deploy_xml);

   // Client connections to Borealis nodes
```

```
   MedusaClient      *_client;
   // Event state for input streams.
   ptr<StreamEvent>   _eventPacket;
   int fd;
};
BOREALIS_NAMESPACE_END
#endif

//================================
// Name        : RegisterQuery.cpp
// Version     : 1.0
//================================

#include "RegisterQuery.h"
#include "HeadClient.h"
#include "util.h"

BOREALIS_NAMESPACE_BEGIN

namespace
{
   // Delay between injections
   const uint32  SLEEP_TIME = 1;
  // Number of input tuples per batch
   const uint32  BATCH_SIZE = 10;
   const uint32 MAX_BUFFER = 128*1024;

   const Time t = Time::now();

   const char *HEAD_NODE = "127.0.0.1";
   const char *BOREALIS_NODE = HEAD_NODE;
   const char *TCPUDP_ENDPOINT = "127.0.0.1";
}

//  Subscribe to input and output streams.
int32 RegisterQuery::open(string xml_file, string deploy_xml)
{
   // Open a client to send and receive data.
   _client = new MedusaClient(InetAddress());
   // Subscribe to outputs.
   subscribeOutput();

   //  Launch the front-end with the xml file.
   int32 status = launchDiagram(xml_file, deploy_xml);

   if (status)
      ERROR << "launchDiagram failed ( " << status << " )";
   else // Establish input data path connections.
```

```
      connectPacket();

   return(status);
}

void RegisterQuery::setSource(int source)
{
   fd = source;
}

//  Activate the client I/O event loop.
void  RegisterQuery::runClient()
{
   _client->run();
}

//  Terminate the client I/O event loop.
void  RegisterQuery::terminateClient()
{
   _client->terminate();
   close(fd);
}

void  RegisterQuery::connectPacket()
{
   // Starting to produce events on input stream.
   if (!_client->set_data_path(MAX_BUFFER,
               Util::get_host_address(BOREALIS_NODE), 15000))
      ERROR << "Failed setting data path";
   else {
      DEBUG << "Set data path";
      _eventPacket = ptr<StreamEvent>(
                        new StreamEvent(_input_name));
      _eventPacket->_inject = true;
   }
}

void  RegisterQuery::batchPacket( Packet  *tuple )
{
   // Tuples are buffered in a string
   _eventPacket->insert_bin(string((const char *)tuple,
            sizeof (TupleHeader)+_input_schema.get_size()));
}

void  RegisterQuery::sendPacket( uint32  sleep )
{
   // Transmit data to the node.
   Status  status = _client->fast_post_event(_eventPacket);
```

```
   while (!status) {
      if (status.as_string() == DataHandler::NO_SPACE) {
         WARN << "We dropped a tuple";
         Thread::sleep(Time::msecs(sleep));
         // retry (make this conditional)
         status = _client->fast_post_event(_eventPacket);
      }
      else {
         ERROR << "Connection closed. stopping event stream";
         return;
      }
   }

   // Event loop is activated so the queue can be processed
   if (sleep) {
         (new CallbackTimer(_client->get_loop(),
                     wrap(this, &RegisterQuery::delayPacket)))
                     ->arm(Time::now() + Time::msecs(sleep));
    }
}

//  Resume here after sending a tuple.
void  RegisterQuery::delayPacket()
{
   // Release the previous event.
    _eventPacket.reset();

    // Construct a new Packet input event.
    _eventPacket = ptr<StreamEvent>(
                new StreamEvent(_input_name));
    _eventPacket->_inject = true;

    // Return to the application code.
    sentPacket();
}

// Subscribing to receive output on a fast datapath.
void  RegisterQuery::subscribeOutput()
{
   DEBUG << "Subscribing to receive output.";

   // Setup the subscription request.
   Status  status = _client->set_data_handler(
   InetAddress(Util::form_endpoint(TCPUDP_ENDPOINT,
         DEFAULT_MONITOR_PORT)), wrap(&outputHandler));

   if (status)
```

```
      DEBUG << "Done subscribing to output.";
   else
      ERROR << "Could not subscribe to output.";
}

// Receive output on a fast datapath
Status  RegisterQuery::outputHandler(ptr<StreamEvent>  event)
{
    int32         index;
    uint32        offset = 0;
    OutputTuple   tuple;

    map<string, CatalogSchema>::iterator pos =
               _output_schema.find(event->_stream.as_string());
    if (pos == _output_schema.end())
         NOTICE << string("Unknown output stream ") +
                                    to_string(event->_stream);
    else {
       // For each tuple that was received,
       for (index=0; index < event->_inserted_count; index++) {
           offset += HEADER_SIZE;
           // At the tuple data past the header.
           tuple.str = event->_bin_tuples.substr(offset,
                                    (*pos).second.get_size());
           DEBUG << "DATA:  " << to_hex_string(&tuple,
                                    (*pos).second.get_size());
           receivedOutput(&tuple, event->_stream.as_string());
           offset += (*pos).second.get_size();
       }
    }
    // Signal done with this batch.
    return(true);
}

int32 RegisterQuery::launchDiagram
                (string xml, string deploy_xml)
{
   if (deploy_xml.empty()) {
      int32   status;
      string  command;

      INFO << "Connect with: " << xml;

      command = string() + "BigGiantHead  " + xml;
      status  = system(command.c_str());

      DEBUG << "Ran the Borealis front end:  " << status;
```

```
      return(status);
   }
   else {
      HeadClient  *client;
      RPC<void>  rpc;

      client = (HeadClient *)new HeadClient(
                InetAddress( HEAD_NODE, DEFAULT_HEAD_PORT ));

      client->deploy_xml_file(xml);
      if (!rpc.valid())
      {
         WARN << "Error with query file " << rpc.stat();
         exit(0);
      }

      client->deploy_xml_file(deploy_xml);
      if (!rpc.valid())
      {
         WARN << "Error with deployment file " << rpc.stat();
         exit(0);
      }
      return (int32)0;
   }
}

const char *get_value(const std::string &str, int *offset,
             const std::vector<SchemaField>::iterator &it)
{
   int i = *offset;
   *offset += (*it).get_size();
   return str.substr(i, (*offset)).c_str();
}

// Print the content of received tuples.
void RegisterQuery::receivedOutput
                     (OutputTuple *tuple, string output)
{
   NOTICE << "Output: " << output;

   int i = 0;
   std::vector<SchemaField> fields =
               _output_schema[output].get_schema_field();
   for(std::vector<SchemaField>::iterator it = fields.begin();
                                it != fields.end(); it++) {
      if((*it).get_type() == DataType::INT)
         NOTICE << (*it).get_name() <<  "=" <<
         (int)*((const int32*)(get_value(tuple->str,&i,it)));
```

```
      else if((*it).get_type() == DataType::LONG ||
             (*it).get_type() == DataType::TIMESTAMP)
         NOTICE << (*it).get_name() <<  "=" <<
             (const int64*)(get_value(tuple->str,&i,it));
      else if((*it).get_type() == DataType::SINGLE)
         NOTICE << (*it).get_name() <<  "=" <<
             (const single*)(get_value(tuple->str,&i,it));
      else if((*it).get_type() == DataType::DOUBLE)
         NOTICE << (*it).get_name() <<  "=" <<
             (const double*)(get_value(tuple->str,&i,it));
      else if((*it).get_type() == DataType::STRING)
         NOTICE << (*it).get_name() <<  "=" <<
               get_value(tuple->str,&i,it);
      else
         NOTICE << "For time interval starting at " <<
                to_hex_string(&tuple[i], (*it).get_size());
   }
}

void  RegisterQuery::sentPacket()
{
   unsigned n = 0;
   while( n++ < BATCH_SIZE) {
      std::string str;

      if (fd != -1) {
         Packet  tuple;
         char buf[1024]={0};
         if (read(fd, buf, _input_schema.get_size()) <= 0)
            continue;
         memcpy(tuple._data, buf, _input_schema.get_size());
         batchPacket(&tuple);
      }
      else {
         WARN << "iptool not available";
         char buf[16] = {0};
         sprintf(buf, "/tmp/%d", getpid());
         fd = open(buf, O_RDONLY);
         break;
      }
   }
   sendPacket(SLEEP_TIME);
}
BOREALIS_NAMESPACE_END

//================================
// Name        : PaQueTDiagram.h
// Version     : 1.0
```

```
//================================

#ifndef  PAQUETDIAGRAM_H
#define  PAQUETDIAGRAM_H

#include "Diagram.h"

BOREALIS_NAMESPACE_BEGIN

class PaQueTDiagram : public Diagram
{
public:
   // Determine the schemas for pendieng intermediate streams
   Status  infer_schema();
   CatalogSchema _input_schema;
   string _input_name;

protected:
   static map<string, CatalogSchema> _output_schema;

};

BOREALIS_NAMESPACE_END
#endif

//================================
// Name        : PaQueTDiagram.h
// Version     : 1.0
//================================

#include  "PaQueTDiagram.h"
#include  "parseutil.h"

BOREALIS_NAMESPACE_BEGIN

//  Determine the schemas for pending intermediate streams.
//  Check for cyclic deployment.
Status  PaQueTDiagram::infer_schema()
{
   Status       status = true;
   Boolean      done   = False;
   Boolean      cyclic;
   Boolean      infer;
   BoxMap      *box_map;
   bool     getInput = false;

   set<CatalogBox *> box_infer;
   set<CatalogBox *>::iterator  box;
```

```
BoxMap::iterator map;
CatalogStream::StreamStar::iterator in;
CatalogStream::StreamStar *box_in;

// If there are any boxes the network must be complete
// and acyclic. List all the boxes yet to be inferred
box_map = get_box_map();

for (map = box_map->begin(); map != box_map->end(); map++)
   box_infer.insert(&(map->second));

// do until all streams have schemas
while (( status )  &&  ( !done )) {
   cyclic = False;
   done   = True;

   //  Do over boxes pending inferencing on each node
   DEBUG << "Infer " << box_map->size() << " boxes ...";

   for (box = box_infer.begin();
                        box != box_infer.end(); box++) {
      DEBUG  << "box_name=" << (*box)->get_box_name();

   cyclic = True;
   infer  = True;
   done   = False;
   box_in = (*box)->get_box_in();

   // See if all Ins have schemas
   for (in = box_in->begin(); in != box_in->end(); in++) {
      WARN << "In stream (" +
               to_string((*in)->get_stream_name()) + ")";

      if (!(*in)->get_stream_schema()) {
         infer = False;
         break;
      }

      if (!getInput) {
         getInput = true;
         _input_schema = *(*in)->get_stream_schema();
         _input_name = (*in)->get_stream_name_string();
      }
   }

   // If all Ins have schemas, infer Outs;
   //    enqueue in deployment order
   if (infer) {
```

```
            cyclic = False;
            DEBUG  << "Infer box " << (*box)->get_box_name();
            status = (*box)->infer_box_out(get_schema_map());

            if (status.is_false()) {
               WARN << "WARNING:  The box"
                      << (*box)->get_box_name()
                      << " has an unknown external type ("
                      << (*box)->get_box_type() << ")";
               status = true;
               done   = True;  // Give up on any more inferencing
            }
            else {
               CatalogStream::StreamStar           *box_out;
               CatalogStream::StreamStar::iterator out;
               box_out = (*box)->get_box_out();
               for (out = box_out->begin();
                                 out != box_out->end(); out++) {
                  WARN << "Stream Name: " <<
                           (*out)->get_stream_name_string();
                  CatalogSchema *schema = (*out)->get_schema();
                  _output_schema.insert(make_pair(
                     (*out)->get_stream_name_string(), *schema));
                  WARN << "Schema Size: " << schema->get_size();
                  std::vector<SchemaField> fields =
                                    schema->get_schema_field();
                  for(std::vector<SchemaField>::iterator it =
                        fields.begin(); it != fields.end(); it++)
                     WARN << (*it).as_string();
               }
            }
            box_infer.erase(&*(*box));
            break;
         }
         DEBUG << "Skip box (" << (*box)->get_box_name() << ")";
          // Assert at least one box was resolved each round
         if (cyclic)
            status = "Cyclic network.";
      }
      return( status );
}
BOREALIS_NAMESPACE_END
```