CHRISTIAN LYRA GOMES

ANÁLISE DE TRÁFEGO DE BACKBONES BASEADA EM SISTEMAS GERENCIADORES DE STREAMS DE DADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Dr. Elias Procópio Duarte Junior
Co-orientadora: Profa. Dra. Carmem Satie Hara

CURITIBA

2008

SUMÁRIO

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

CQL - Continuous Query Language

API - *Application Program Interface*

DoS - *Deny of Service*

DDoS- *Distributed Deny of Service*

ISO - *International Standards Organization*

MIB - Management Information Base

OID - Object Identificator

OSI - Open Systems Interconnection

RMON - Remote Network Monitoring

RPC - Remote Procedure Call

SA - Sistema Autônomo

SGBD - Sistema Gerenciador de Banco de Dados

SGSD - Sistema Gerenciador de Streams de Dados

SPE - *Stream Processing Engine*

SNMP - Simple Network Management Protocol

Resumo

A análise de tráfego em um backbone apresenta uma série de desafios. O volume de tráfego, o número de componentes da rede e a distribuição geográfica destes componentes [9] são alguns dos principais fatores que tornam árdua tal tarefa. Os Sistemas Gerenciadores de Streams de Dados (SGSD) foram desenvolvidos para a análise de fluxos ou *streams* de informação em tempo real. Este trabalho descreve a utilização de um SGSD para a monitoração de tráfego em backbones. Uma ferramenta que permite a elaboração de consultas diversas sobre o tráfego de rede foi implementada, validada e avaliada. A ferramenta utiliza o SGBD Borealis e informações de tráfego providas pelo protocolo *Netflow* sendo capaz de analisar o tráfego de forma distribuída e em tempo real. Estudos de caso são apresentados para demonstrar a funcionalidade da ferramenta.

Abstract

Backbone traffic analysis presents a number of challenges, including a huge amount of traffic, a large number of network components as well as their geographic distribution. Stream Process Engines (SPE) are designed to perform stream analysis in real-time. This work describes the use of a SPE for backbone traffic monitoring. A tool was implemented and is described that allows arbitrary queries to be issued about the traffic on the backbone as a whole. The tool is based on the SPE Borealis and obtains backbone traffic information provided by the *Netflow* protocol. The tool was constructed so that several Borealis nodes can be deployed across the backbone in a distributed fashion in order to perform real-time traffic analysis. The architecture of the tool is presented, with case studies designed for validation as well as performance evaluation.

Capítulo 1

Introdução

No gerenciamento de rede, as informações sobre a rede como um todo e seus componentes individualmente fornecem a base para a tomada de decisões relativas à gerência de falhas, desempenho, segurança, configuração e contabilização [2]. Uma parte importante dessas informações são obtidas pela monitoração e análise do tráfego de rede, ou seja, da quantificação e qualificação do tráfego de uma rede.

Nos sistemas de gerência tradicionais, as informações sobre a rede e seus componentes podem ser obtidas através dos processos de *polling* e emissão de alarmes com o auxílio de ferramentas como, por exemplo, aquelas baseadas no protocolo SNMP [33]. As informações obtidas trazem dados diversos mantidos por elementos gerenciados como, por exemplo, quantidade de bytes ou o número de pacotes que passaram por uma determinada interface. Embora de utilização simples, as informações obtidas dessa forma podem não ser suficientes para a execução de todas as tarefas de gerência, e em particular da análise de tráfego.

Os fabricantes de equipamentos de rede têm desenvolvido e incluído em seus produtos recursos que permitem a obtenção de informações mais detalhadas sobre o tráfego de rede. Protocolos como o *Netflow* [21] e *Sflow* [26] são utilizados para se obter informações sobre *fluxos* de pacotes. Um fluxo de pacotes pode ser definido como um conjunto de pacotes com os mesmos protocolos, endereços de origem e endereço de destino [21]. As informações obtidas com o auxílio destes protocolos podem ser armazenadas e analisadas

com ferramentas como o *flow-tools* [19, 31] e o *ntop* [27].

Outra maneira de se obter mais informações sobre o tráfego de rede é através da utilização de programas conhecidos como *sniffers* que são capazes de armazenar e decodificar todo o tráfego de rede direcionado para eles [25]. Embora essa seja uma maneira que permite colher maior número de informações, é necessário fazer com que todo o tráfego de rede seja visto pelos *sniffers*, o que envolve a utilização de equipamentos especiais, ou configurações especiais de rede. Além disso, em uma rede de médio/grande porte a quantidade de dados a serem tratados pode ser muito grande.

A utilização tanto de *sniffers*, quanto de protocolos como *Netflow*, implica em um processamento que geralmente não é feito em tempo real, ou seja, os dados são coletados e armazenados, para serem processados e apresentados posteriormente. Os Sistemas Gerenciadores de Streams de Dados (*SGSD*) foram desenvolvidos visando oferecer funcionalidades semelhantes aos Sistemas Gerenciadores de Banco de Dados (SGBD), mas permitindo a execução de consultas sobre fluxos contínuos de dados *streams*, com a obtenção de respostas em tempo real.

O presente trabalho apresenta uma ferramenta para a análise de tráfego baseada em um SGSD e que permite ao administrador de uma rede definir consultas arbitrárias sobre os dados de tráfego obtidos de forma distribuída em todo um backbone. Para a aplicação de tal ferramenta a um backbone, com múltiplos equipamentos gerando fluxos em localidades diferentes, é utilizada a capacidade de execução de consultas de forma distribuída de um SGSD específico: o Borealis [6]. A ferramenta foi validada e avaliada, e a sua funcionalidade é demonstrada através de três estudos de caso onde ela foi aplicada para a construção de uma matriz de tráfego, detecção de varreduras de rede, e identificação de origens de ataques de negação de serviço.

As principais contribuições deste trabalho incluem a definição de uma arquitetura para análise de tráfego baseado em um SGSD; a implementação de uma ferramenta que permita a análise de tráfego de um backbone de forma genérica e em tempo real; e a avaliação do desempenho do uso do Borealis.

1.1 Organização deste trabalho

O presente trabalho está organizado da seguinte forma: No capítulo 2 são apresentados os conceitos de gerência de redes e os principais protocolos utilizados para análise de tráfego, além de trabalhos relacionados. O capítulo 3 descreve os SGSDs e apresenta o SGSD Borealis. No capítulo 4 a ferramenta para análise de tráfego de redes baseada em um SGSD é apresentada, bem como estudos de caso de sua utilização. As conclusões seguem no capítulo 5. Os códigos e arquivos de configurações utilizados são apresentados no Anexo A.

Capítulo 2

Monitoração de Tráfego de Rede

Esse capítulo inicia com a apresentação de alguns dos principais conceitos de gerência de redes na Seção 2.1. A monitoração de redes baseada no arcabouço SNMP e o protocolo SNMP propriamente dito serão apresentados, de forma resumida, na Seção 2.2. O protocolo *Netflow*, criado especificamente para a análise de tráfego será apresentado na Seção 2.3. O capítulo é encerrado um uma descrição de trabalhos recentes relacionados à análise de tráfego de redes na Seção 2.4

2.1 Conceitos de Gerência de Redes

De acordo com a *International Standards Organization* (ISO), o gerenciamento de redes provê mecanismos para a monitoração, controle e coordenação de recursos e define padrões para a troca de informações entre estes recursos [2]. A gerência de redes pode ser dividida em 5 áreas funcionais, a saber: Gerência de Falhas, Gerência de Configuração, Gerência de Contabilização, Gerência de Desempenho e Gerência de Segurança [2]. Estas áreas funcionais são descritas a seguir.

A *Gerência de Falha* é a área que procura garantir que a rede como um todo e cada dispositivo desse sistema opere de maneira correta. São atribuições dessa área a detecção de falhas, a determinação do local da falha, seu isolamento e correção.

A *Gerência de Configuração* é a área que permite o controle do comportamento de cada dispositivo de rede através de sua configuração. É também a área que cuida da

manutenção, adição e atualização das relações entre componentes e do estado dos próprios componentes durante a operação da rede.

A *Gerência de Contabilização* é responsável pela mensuração da utilização de recursos da rede pelos seus usuários ou grupos de usuários. Por exemplo, em redes corporativas, cada departamento ou centro de custo é cobrado pela utilização que faz da rede, e é essa área de gerência de redes a responsável por essa contabilização.

A *Gerência de Desempenho* envolve a mensuração do desempenho dos componentes da rede, tanto hardware quanto software, e é responsável também pela garantia da qualidade de serviços.

Por fim, a *Gerência de Segurança* é a área envolvida com a proteção da informação, segurança de sistemas e controle de acesso.

Além das áreas expostas acima, a gerência de redes também distingue duas categorias funcionais: monitoramento e controle. O monitoramento é a função que faz o acompanhamento de todas as atividades da rede, e o controle é a função que permite que ajustes sejam feitos visando melhorar o desempenho da rede.

2.2 Breve Descrição do Arcabouço SNMP

O arcabouço de gerência de redes padrão da internet é o *Simple Network Management Protocol* (SNMP). A descrição completa do arcabouço SNMP pode ser encontrada nas RFCs que constituem o padrão STD62 [16], com destaque para as RFCs 3411 [15] e 3416 [17]. Uma breve descrição SNMP será apresentada.

Na gerência de rede baseada em SNMP os principais conceitos são os de *g*erente, agente e objeto gerenciado [23], descritos a seguir. O protocolo SNMP propriamente dito é descrito na sequência.

O *gerente* é a entidade que obtém informações e controla os objetos gerenciados. Esse papel pode ser desempenhado por um único *host* executando um aplicativo de gerência.

Um *agente* executa operações de gerenciamento sobre objetos gerenciáveis, podendo também transmitir ao gerente notificações emitidas por estes objetos. Um *daemon* executado em um *host* ou o software de controle de um comutador são exemplos de agentes.

Um *objeto gerenciado* é a representação de um recurso que está sujeito ao gerenciamento. Tal recurso pode ser um dispositivo de rede ou mesmo uma conexão.

Os *objetos gerenciados* são definidos em termos de seus atributos ou propriedades, as operações a que podem ser submetidos, as notificações que podem emitir, e suas relações com outros objetos. O conjunto de objetos gerenciados dentro de um sistema, juntamente com seus atributos, constituem a *Management Information Base* (MIB).

A sintaxe de cada MIB é definida por um subconjunto da linguagem *ISO Abstract Syntax Notation One* (ISO ASN.1). Cada MIB usa uma arquitetura em árvore definida pela ASN.1 para organizar toda a informação disponível. Assim, cada informação em uma árvore é um *nodo rotulado*. Cada nodo contém um *identificador de objeto* (OID) e uma descrição. Um nodo pode conter outros nodos, e no caso de ser um nodo folha, ele contém também um valor, e é chamado de *objeto* [23].

2.2.1 O Protocolo SNMP

O protocolo SNMP propriamente dito define um conjunto de operações utilizadas na comunicação entre um agente e um gerente. Cada agente mantém uma MIB que reflete o estado dos recursos gerenciados por este agente. Assim um gerente pode monitorar estes recursos lendo os valores dos objetos dessa MIB ou mesmo controlar estes recursos modificando tais valores [33].

O SNMP é tradicionalmente baseado na arquitetura cliente-servidor, e utiliza como protocolo de transporte o UDP. O papel de "cliente" é desempenhado pelo gerente, e o papel de "servidor" pelo agente SNMP. Um gerente pode solicitar informações de um agente através das operações como GET que busca o valor de um objeto específico, GETNEXT, que busca o valor do próximo objeto especificado ou GETBULK que busca os valores de um conjunto de objetos. As solicitações iniciadas pelo gerente são conhecidas como "polling". No entanto, um agente também pode enviar alarmes contendo informações sem que tenha sido perguntado através das operações de TRAP e INFORM, conhecidas simplesmente por "traps". Um gerente pode também enviar solicitações de mudança de configuração para um agente utilizando a operação SET. O protocolo é integralmente

descrito em [16].

2.3 Análise de Tráfego com Netflow

O *Netflow* é uma tecnologia e um protocolo aberto, mas proprietário, desenvolvido pela CISCO [21]. Um roteador com suporte a *Netflow* implementa um agente que é capaz de contabilizar fluxos de pacotes. Um fluxo para o *Netflow* é definido como um *stream* unidirecional de pacotes entre uma dada origem e destino. Mais especificamente, um fluxo é considerado como o conjunto de pacotes com os mesmos campos: endereço IP de origem, endereço IP de destino, porta de origem, porta de destino, protocolo, tipo de serviço e interface de entrada. Além dos campos utilizados para definir o fluxo, o agente é capaz de armazenar dados como o Sistema Autônomo (SA) de origem e o SA de destino do pacote.

O agente *Netflow* mantém um cache para cada fluxo ativo (um *flow record*) no roteador, incrementando o número de pacotes e octetos para cada novo pacote. Essas entradas são então exportadas para um coletor utilizando-se o protocolo *Netflow*. Os agentes *Netflow* utilizados nos roteadores CISCO também são capazes de disponibilizar as informações contidas no seu cache através de uma MIB especial utilizando o protocolo SNMP.

O *Netflow* permite portanto sumarizar e obter estatísticas sobre o tráfego que atravessa um roteador. Como apenas os dados de fluxo são armazenados e não o conteúdo de cada pacote, é possivel armazenar as informações de uma grande quantidade de tráfego. A partir do dados armazenados é possível obter informações como, por exemplo, o número de octetos e fluxos com destino a uma porta específica de um servidor, ou o total de tráfego gerado por um *host*.

O *Netflow* será utilizado como a principal fonte de dados da ferramenta implementada, portanto uma descrição mais detalhada do seu formato é apresentada a seguir.

2.3.1 Formato de Dados do Netflow

O *Netflow* possui diferentes versões, e cada versão traz diferentes tipos de dados, organizados de diferentes maneiras. Uma das versões do *Netflow* mais populares, e usualmente

encontrada nos roteadores CISCO é a versão 5 (*Netflow-v5*).

O *Netflow* utiliza como protocolo de transporte o UDP. Dentro de cada datagrama UDP, um pacote *Netflow* é inserido, contendo um cabeçalho e de 1 a 30 registros de fluxo. O cabeçalho de cada pacote contém campos contendo o número da versão do protocolo utilizado, o número de fluxos (registros) contidos no pacote, além campos com valores de temporizadores e identificadores do agente que gerou o pacote. A Figura 2.1 mostra o formato do cabeçalho de um pacote *Netflow-v5*.

Número de versão		Número de fluxos no pacote
Tempo de uptime do roteador (ms)		
Tempo de exportação (em segundos epoch)		
Tempo de exportação (ns)		
Número de sequência		
Tipo de agente	Id do agente	[enchimento]

Figura 2.1: Formato do cabeçalho de um pacote *Netflow-v5*.

Após o cabeçalho são inseridos os registros de fluxo. Cada registro possui campos contendo os endereços de origem e destino do fluxo, índices SNMP das interfaces de entrada e saída do fluxo no roteador, total de pacotes, total de octetos, valores de temporizadores do roteador, porta de origem e de destino do fluxo, protolo, Sistema Autônomo (AS) de origem e de destino, além de informações sobre flags do pacote. A Figura 2.2 apresenta o formato de um registro, com todos os campos utilizados.

Endereço IP de origem			
Endereço IP de destino			
Endereço IP do próximo roteador (next-hop)			
Índice SNMP da interface de entrada		Índice SNMP da interface de entrada	
Total de pacotes			
Total de octetos			
Tempo de uptime do roteador no ínicio do fluxo (ms)			
Tempo de uptime do roteador no fim do fluxo (ms)			
Porta de origem		Porta de destino	
[Enchimento]	Flags TCP	Protocolo	ToS
AS de origem		AS de destino	
Máscara de origem	Máscara de destino	[Enchimento]	

Figura 2.2: Formato de um registro de fluxo do *Netflow-v5*.

2.4 Trabalhos Relacionados

A área de gerência de redes e análise de tráfego é rica em trabalhos e ferramentas relacionadas à proposta deste trabalho. Em [1] é possível encontrar uma extensa lista destas ferramentas.

Ferramentas que trabalham em nível de pacotes, como o *tcpdump* [25] e o *wireshark* [5], capturando e mostrando esses pacotes são bastante populares entre os administradores de redes. No entanto tais ferramentas não permitem a realização de consultas arbitrárias sobre o tráfego capturado, além disso, usualmente são alimentadas por uma única fonte de dados. Em [4] é descrita uma solução que consiste de coletores distribuídos capazes de receber informações no formato *Netflow* e exportar os dados recebidos no formato *pcap* para que o *wireshark* os analise, no entanto os autores reconhecem que a quantidade de informação pode ser grande e recomendam a filtragem dos dados de interesse nos próprios coletores.

O *ntop* é uma ferramenta descrita em [34] que é capaz de trabalhar tanto em nível de pacotes, quanto em nível de fluxos através de um *plugin* que é capaz de receber informações de tráfego via *Netflow*. Embora seja capaz apresentar vários tipos de relatórios, ela também não permite a utilização de consultas arbitrárias definidas pelo administrador de redes.

Dentre as ferramentas que trabalham com *Netflow*, uma das primeiras a ser distribuída no modelo *open source* foi a *cflowd* [3], em 1998, e que mais tarde deu origem ao *flowscan* [30]. Os autores de [29], afirmam que sistemas que trabalham em nível de pacotes não escalam em redes de altas velocidades, e descrevem um *framework* para monitoração em tempo real de backbones e detecção de ataques baseado em *Netflow*. O *framework* descrito funciona de forma centralizada e depende que os usuários escrevam *plugins* para cada tarefa a ser realizada.

Em [36] é descrito um sistema para análise de tráfego baseado em *Netflow* onde as informações são armazenadas em um banco de dados *Oracle*. A utilização de bancos de dados permite uma grande flexibilidade na realização das consultas, no entanto, tal sistema ainda requer que a informação seja armazenada para depois ser processada, além

de funcionar de forma centralizada.

Os autores da ferramenta *SMART*, descrita em [8], argumentam que as ferramentas tradicionais que utilizam *Netflow* trabalham armazenando informações em disco para depois processá-las, o que impede o processamento de grandes volumes de tráfego de maneira eficiente. Afirmam também que Sistemas Gerenciadores de Streams de Dados (SGSD, descritos no Capítulo 3) não foram feitos especificamente para o monitoramente de tráfego com *Netflow*. O *SMART* utiliza o armazenamento e processamento dos dados em memória. No entanto, os resultados obtidos pelo *SMART*, que foi capaz de processar 30 mil fluxos por segundo, é similar ao obtido no presente trabalho (que utiliza um SGSD).

Em [32, 14] é descrito o sistema *Gigascope*, que funciona como um SGSD específico para o monitoramento de redes de alta velocidade. Embora os resultados sejam os mais significativos em se tratando de monitoramento de redes, o *Gigascope* é uma ferramenta comercial e de código fechado, utilizada pela AT&T.

Motivados em investigar a utilização de um SGSD *open source* para análise de tráfego, os autores de [18] descrevem um estudo de caso que utilizou o SGSD TelegraphCQ. Os resultados obtidos, em termos de quantidade de tráfego analisado, foram modestos, mas serviram de motivação para a realização de outros trabalhos, como o da ferramenta *PaQueT* descrita em [20]. A *PaQueT* utiliza o SGSD Borealis e permite que o administrador de redes efetue consultas arbitrárias em nível de pacote sobre o tráfego capturado em uma interface de rede, ou sobre um arquivo no formato *pcap*.

Assim como a *PaQueT*, o presente trabalho também utiliza o SGSD Borealis para a análise de tráfego. Enquanto que *PaQueT* permite a monitoração de *um* segmento de rede, a ferramenta apresentada no presente trabalho permite a análise de tráfego de um backbone, tendo como características a possibilide de utilizar múltiplas fontes de dados, processamento distribuído em múltiplos nodos Borealis, separação das funções de captura, controle e visualização, além de trabalhar em nível de fluxo (utilizando principalmente o *Netflow*).

Embora não tenha sido utilizado no presente trabalho, uma alternativa ao uso do *Netflow* é o *SFlow*, que é uma tecnologia para monitoramento de tráfego definida pela

RFC 3176 [26]. Nela é definido um mecanismo de amostragem de pacotes em um agente *SFlow*, uma MIB e um formato de dados utilizado na comunicação entre o agente e um coletor *SFlow*. Como no agente *SFLow* não é necessário manter informações de fluxos, a implementação do agente é bastante simplificada. Além disso, a especificação foi projetada para atender de forma precisa o monitoramento de interfaces a velocidade de Gigabits por segundo ou mais. O *SFlow* utiliza amostragem estatística de pacotes, no entanto, como o coletor que recebe as amostras conhece a razão de amostragem, é possível reconstruir com razável precisão o tráfego amostrado [28, 35].

Capítulo 3

Sistemas Gerenciadores de Streams de Dados

Os Sistemas Gerenciadores de Streams de Dados e suas características são apresentados neste capítulo, que inclui também definições básicas e um breve histórico da sua evolução. Um SGSD específico, o Borealis, é apresentado.

3.1 Características de um SGSD

Os Sistemas Gerenciadores de Streams de Dados (SGSD) [12] surgiram das necessidades de uma nova classe de aplicações que buscam monitorar eventos em tempo-real, tais como análise de dados de sensores, monitoramento de tráfego, análise em tempo real de ações da bolsa de valores e monitoramento de campo de batalhas. Dado o volume de dados que tais aplicações geram e a necessidade de se efetuar consultas sobre esses dados indicam que elas se beneficiariam do uso de um SGBD. No entanto, os sistemas tradicionais de bancos de dados, os SGBDs, não são capazes de atender todos os requisitos dessa classe de aplicações, que possuem em comum a necessidade de processamento em tempo real, além do processamento de entradas de dados alimentadas continuamente e em grande volume. As características dos SGSDs e as diferenças entre os SGSDs e os SGBDs serão apresentadas a seguir.

Em geral, os SGBDs possuem uma quantidade de dados armazenados e a cada mo-

mento consultas transientes são feitas sobre esses dados. Já nos SGSDs os dados não são armazenados e sim fornecidos por fontes externas, como sensores. Um conjunto de consultas fixas são registradas, e são continuamente aplicadas aos novos dados que chegam. A atualização dos dados também é diferente nos dois sistemas. Enquanto em um SGBD é possível modificar, adicionar ou excluir registros, em um SGSD é possível apenas inserir novos dados.

Tendo em vista o volume de dados processados, a necessidade de resultados em tempo real e o fato de a importância do resultado obtido decair rapidamente com o tempo, nos SGSDs se admitem resultados aproximados, e o sistema pode até mesmo descartar dados classificados como menos importantes [12]. Tal característica contrasta com o objetivo de um SGBD, que deve gerar resultados precisos. Algumas estratégias como o uso de cálculos probabilísticos e a revisão de resultados podem ser utilizadas para aumentar a precisão da resposta obtida dentro de uma determinada janela de tempo.

A avaliação de uma consulta também difere nos dois sistemas. Em um SGBD a consulta se faz através do acesso arbitrário aos dados. Já em um SGSD a avaliação tem que ser feita de uma vez, à medida que os dados vão chegando [7]. O plano de consulta também difere nos dois sistemas: deve ser fixo nos SGBDs, mas nos SGSDs deve ser adaptativo. Desta forma, um SGSD tenta otimizar os resultados de consultas, e o plano de consulta pode ser modificado para aproveitar resultados, parciais ou não, de outras consultas sendo feitas no mesmo instante.

A Tabela 3.1 resume as principais características dos SGSDs e SGBDs.

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

Tabela 3.1: Comparação das características de SGBD e SGSD.

Os primeiros protótipos de SGSDs foram desenvolvidos no inicio desta década, como

por exemplo o projeto Aurora [12], desenvolvido em 2002, o projeto TelegraphCQ [13], que foi baseado no projeto Telegraph de 2000, e o projeto STREAM [11] de 2004. Esses primeiros esforços tinham como foco o projeto de novos operadores e novas linguagens, assim como a construção de engenhos de alto desempenho operando de forma centralizada.

Os SGSDs de segunda geração, como o Medusa [24] e o Borealis [6], foram construídos com base na experiência adquirida com os sistemas de primeira geração, e evoluíram para se tornarem sistemas distribuídos e com recursos como tolerância a falhas, balanceamento de carga e processamento paralelo.

3.2 O SGSD Borealis

O Borealis é um SGSD distribuído de segunda geração desenvolvido por um grupo de pesquisadores de três universidades: Brandeis University, Brown University e MIT [6]. O Borealis estende e modifica funcionalidades presentes nos projetos Aurora [12] e Medusa [24].

De uma maneira simplificada, um sistema Borealis recebe um conjunto de streams como entrada, e continuamente processa os dados do stream, agregando, filtrando e correlacionando-os de forma a produzir uma saída de interesse de outra aplicação. No Borealis, um operador que processe um stream pode estar distribuído por várias máquinas físicas, chamadas de nodos processadores, ou simplesmente nodos.

Um sistema Borealis é composto pelos seguintes componentes: o *catálogo distribuído*, *nodos*, *aplicações clientes*, e *fontes de dados*, descritos a seguir.

O *catálogo distribuído* guarda informações do sistema como um todo. Esta informação inclui o diagrama de consultas, ou seja, o conjunto de todos os operadores e todos os streams, e a informação de distribuição, que especifica o designação de operadores aos nodos processadores.

Os *nodos* são os responsáveis pelo processamento dos streams. Cada nodo executa um pedaço do diagrama de consulta e armazena informações sobre esse pedaço em um catálogo local. Os *nodos* também executam as tarefas necessárias para permitir a gerência de carga e garantir a tolerância a falhas.

As *aplicações clientes* interagem com o Borealis produzindo dados para o sistema,

ou consumindo os resultados produzidos pelo sistema. Essas aplicações podem também criar ou modificar pedaços do diagrama de consulta. O sistema Borealis vem com duas aplicações clientes por padrão. Um cliente com uma interface gráfica para a modificação de um diagrama de execução, e uma ferramenta de monitoramento capaz de mostrar as condições de carga e designação.

As *fontes de dados* são aplicações cliente que produzem streams de dados e as enviam para o processamento pelos nodos.

A Figura 3.1 apresenta os componentes de um sistema Borealis com *fontes de dados*, *nodos* e *aplicações clientes*.

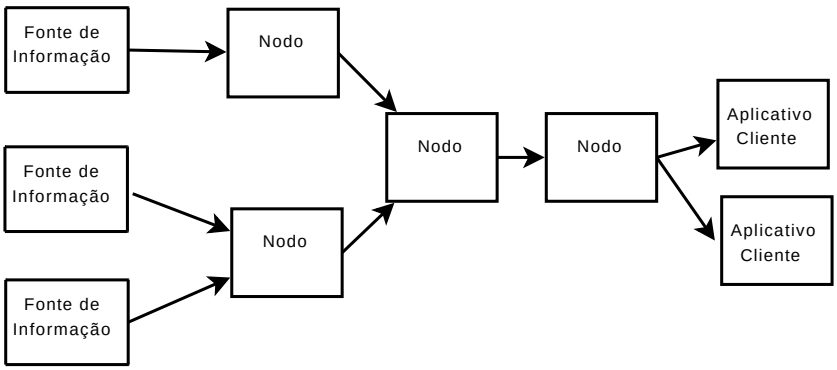

Figura 3.1: Componentes do sistema Borealis.

3.2.1 Arquitetura de um Nodo Borealis

Cada nodo de processamento do Borealis contém todos os componentes necessários para funcionar como um SGSD independente. A arquitetura de um nodo Borealis é mostrada na Figura 3.2. O principal componente de um nodo é o *processador de consultas*, que é o componente onde o processamento de streams propriamente dito é realizado. O *processador de consultas* é composto por: uma interface *Admin* que é responsável por receber requisições, que por sua vez podem modificar a parte local do diagrama de consultas e em alguns casos solicitar o envio de operadores para outros nodos; um *catálogo local* que mantém informações sobre o pedaço local do diagrama de consultas; um *DataPath*, responsável por gerenciar a entrada de streams e saída de resultados; e um *nodo Aurora*

que é o processador de streams, responsável por instanciar operadores e escalonar sua execução.

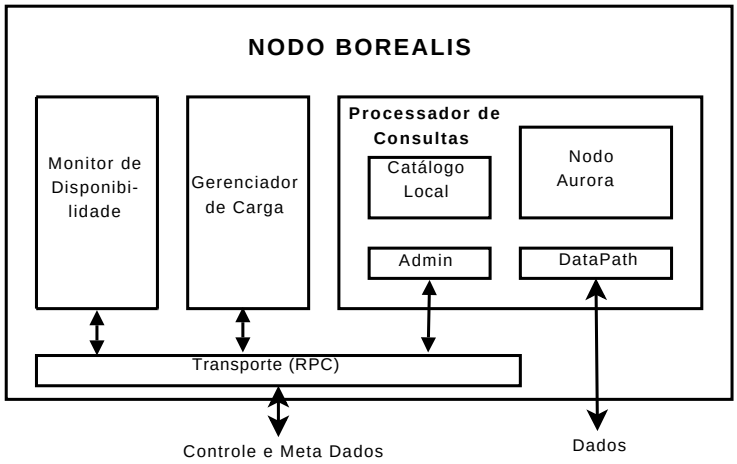

Figura 3.2: Nodo Borealis.

Além do *processador de consultas* cada nodo Borealis possui módulos para a comunicação e colaboração com outros nodos. Esse módulos são o *monitor de disponibilidade*, cuja função é o monitoramento do estado de outros nodos, e o *gerenciador de carga* que utiliza a informação de carga local e remota para prover o balanceamento de carga entre os nodos. Para efetuar o envio e o controle de mensagens entre componentes e entre nodos é utilizado um módulo que atua como uma camada provendo o transporte baseado em RPC.

3.2.2 Consultas

O Borealis processa fluxos de informações filtrando, correlacionando, agregando e transformando entradas em saídas que serão consumidas por outras aplicações. Dentro de cada nodo Borealis fluxos de entrada são transformados em saídas através da passagem destes por um conjunto de operadores (também chamados de "caixas"). Uma consulta é definida em termos das entradas, saídas e um conjunto de caixas. Dois tipos de operadores estão presentes no Borealis: os operadores *stateless* e os *stateful*. A seguir serão apresentados os principais operadores do Borealis.

3.2.2.1 Operadores *Stateless*

Os operadores do tipo *stateless* são aqueles que efetuam sua função uma tupla por vez sem a retenção de nenhum estado entre as tuplas. O Borealis possui três operadores deste tipo: *Filter*, *Map* e *Union*, descritos a seguir. Uma representação gráfica de cada um deles é mostrada na Figura 3.3.

O operador do tipo *Filter*, como o próprio nome indica, funciona como um filtro deixando passar ou não uma tupla tendo em vista algum predicado. Por exemplo, se cada tupla é composta por um valor X e um Y, é possível aplicar um filtro que só deixe passar as tuplas nas quais o valor X é maior que 10. O operador *Filter* pode funcionar também como a instrução *case* da linguagem C, direcionando a tupla para uma saída diferente de acordo com o casamento de um determinado valor de predicado.

O operador do tipo *Map* transforma as tuplas de entrada em tuplas de saída aplicando um conjunto de funções aos atributos da tupla. Utilizando o exemplo anterior, um operador do tipo *Map* poderia multiplicar o valor do atributo Y por 8, transformando assim octetos em bits.

O operador *Union* funciona mesclando um conjunto de fluxos de entrada em um único stream de saída (todos os fluxos devem ter o mesmo esquema). Este operador pode ser utilizado quando se tem múltiplas fontes produzindo o mesmo tipo de dado e se deseja processá-los como se fossem um só.

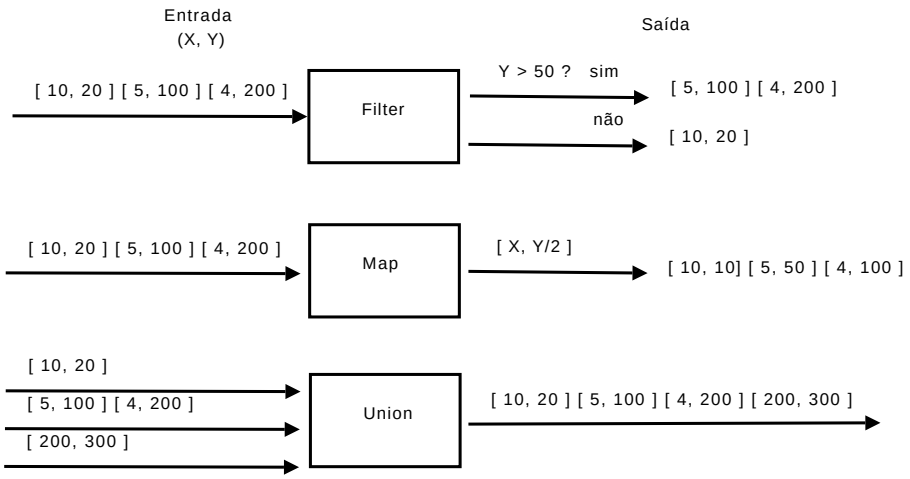

Figura 3.3: Operadores *Stateless*

3.2.2.2 Operadores *Stateful*

Ao contrário dos operadores *stateless*, os operadores *stateful* processam algum tipo de computação sobre grupos de tuplas de entradas e não sobre tuplas isoladas. O tamanho desse grupo pode ser definido em termos do número de tuplas, ou de algum atributo das tuplas. Usualmente esse tipo de operador é aplicado sobre uma janela de dados que se movem com o tempo (por exemplo, tuplas dentro de um intervalo de tempo de 10s). Embora o Borealis apresente um certo número de operadores stateful, o principal operador, e o único descrito aqui será o operador *Aggregate*.

O operador *Aggregate* aplica uma função de agregagação como média, máximo, ou contador. A função é computada sobre os valores de um atributo das tuplas de entrada. Antes da função ser aplicada, o operador *Aggregate* pode opcionalmente particionar o fluxo de entrada de acordo com um ou mais de seus atributos. Por exemplo, se as tuplas contém os atributos [protocolo, octetos] o operador *Aggregate* poderia contabilizar o total de octetos por protocolo.

3.2.2.3 Diagramas de Consultas

No Borealis uma consulta toma a forma de *dataflow*, um fluxo de dados. Para expressar uma consulta sobre fluxos, os usuários ou aplicações combinam os operadores em um diagrama de "caixas e flechas", chamado de "diagrama de consulta". Para informar ao Borealis o diagrama de consulta é utilizado um arquivo XML contendo a descrição do diagrama, no entanto é também possível representá-lo graficamente na forma de um diagrama propriamente dito, como o da Figura 3.4.

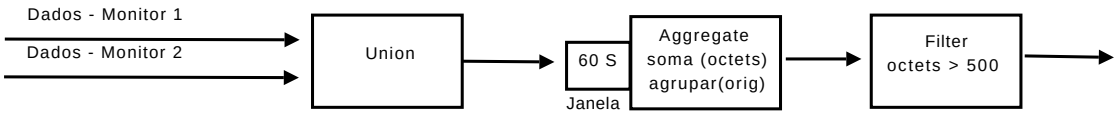

Figura 3.4: Exemplo de um diagrama de consulta.

3.2.3 Redes Borealis

O conjunto de entradas, saídas, consultas e nodos Borealis formam uma rede Borealis. A configuração de uma rede Borealis é feita através de um arquivo XML que define onde as entradas serão recebidas, para onde as saídas devem ser enviadas, e o que cada nodo Borealis executará. O processo de configuração da rede Borealis é chamado de *deploy*.

É possível utilizar apenas um arquivo XML contendo todas as configurações necessárias para se configurar uma rede Borealis, no entanto, o procedimento mais utilizado é separar as definições de fluxos de entrada e saída, bem como o diagrama de consulta em um arquivo XML, e as informações de configuração de que nodo executa o que em outro arquivo. O exemplo da Figura 3.5 apresenta um arquivo de *deploy*.

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">
<deploy>
    <publish    stream="flow"   endpoint="192.168.0.1:15000" />
    <subscribe  stream="result" endpoint="192.168.0.2:4444" />
    <node  query="contagem"     endpoint="192.168.0.1:15000" />
</deploy>
```

Figura 3.5: Arquivo de *deploy*

O arquivo de exemplo define que o nodo 192.168.0.1 receberá como stream de entrada o fluxo “flow”e também será responsável por executar a consulta “contagem”. O resultado dessa consulta será enviado para o nodo “192.168.0.2”e receberá o nome de “result”. O esquema de entrada, de saída e a consulta foram previamente definidos em outro arquivo XML (o diagrama de consulta).

Capítulo 4

Uma Ferramenta para Análise de Tráfego de Redes Baseada em SGSD

Neste capítulo é apresentada a ferramenta construída bem como sua avaliação e validação. Três estudos de caso de aplicação da ferramenta são também apresentados.

4.1 A Arquitetura da Ferramenta Proposta

A ferramenta implementada permite que um administrador de rede efetue consultas arbitrárias sobre os dados de tráfego obtidos de forma distribuída em todo um backbone. Para a aplicação de tal ferramenta a um backbone, com múltiplos equipamentos gerando fluxos em localidades diferentes, utiliza-se a capacidade de execução de consultas de forma distribuída do SGSD Borealis [6].

A arquitetura da ferramenta proposta utiliza nodos Borealis localizados próximos às fontes de dados, e é constituída por três conjuntos de aplicações responsáveis pela aquisição de dados, controle do sistema e apresentação.

Antes que um determinado fluxo de dados possa ser tratado por um nodo Borealis, ele deve ser processado para que contenha o formato de entrada esperado pelo Borealis. Uma vez convertido, o resultado é enviado para o nodo Borealis usando o protocolo de comunicação do Borealis. Os resultados gerados pelos nodos Borealis devem ser recebidos por uma aplicação que seja capaz de entender o protocolo de comunicação do Borealis a

fim de apresentá-los ao usuário.

A Figura 4.1 apresenta uma visão geral da arquitetura. O sistema pode ser composto de vários nodos Borealis, vários nodos "alimentadores"(*aquisição*) e vários nodos de *visualização*. Além destes, é necessário também um mecanismo de *controle* que permita a coordenação destes nodos. Cada conjunto de aplicações pode ser executada em um nodo distinto, ou podem ser agregadas em um mesmo nodo, e serão detalhadas a seguir.

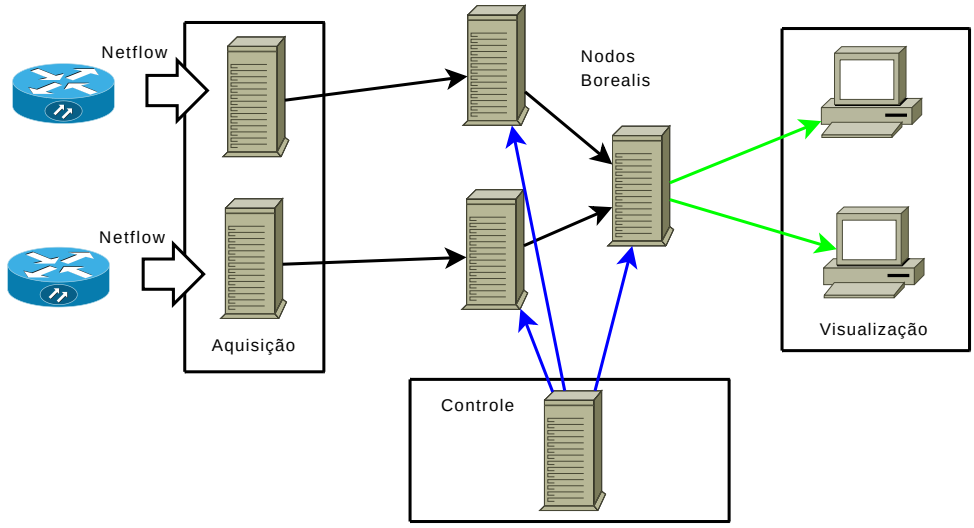

Figura 4.1: A arquitetura de análise de tráfego baseada no SGSD Borealis.

4.1.1 Aplicações para Aquisição de Dados

O módulo de aquisição de dados é responsável por fazer a interface entre as fontes de dados e os nodos Borealis. Dois grupos de tarefas são executadas por este módulo: a aquisição e conversão dos dados; além do envio dos dados para os nodos Borealis.

A ferramenta proposta utiliza os dados obtidos através do protocolo Netflow, no entanto também é possivel utilizar outras fontes de dados, desde que as informações providas possam ser transformadas ao formato de entrada esperado. De forma a manter uma certa independência da fonte de dados, a aquisição e conversão dos dados é executada por uma aplicação e o envio dos dados para os nodos Borelis é feito por outra aplicação. A comunicação destas duas aplicações se dá por meio de memória compartilhada.

Para suprir a função de fonte de dados utilizando o protocolo Netflow foi utilizada uma aplicação distribuída como software livre chamada de *New Netflow Collector* (*nnfc*) [22]. O *nnfc* é um coletor de fluxos Netflows, isto é, ele é capaz de receber fluxos de Netflow enviados por um roteador, decodificá-los e armazená-los. Um *plugin* foi escrito para o *nnfc* para que os fluxos de Netflow recebidos e decodificados pudessem ser enviados através de um mecanismo de IPC (*Inter Process Communication*) para a aplicação de envio.

A aplicação que faz o envio propriamente dito de informações para o Borealis recebeu o nome de *flowsender*, e foi implementado em C++ utilizando-se a API (*Application Programming Interface*) e bibliotecas do Borealis. O *flowsender* é capaz de ler registros de fluxo das filas em memória e enviá-los para um nodo Borealis. A fila a ser lida bem como o endereço do host e porta de destino do nodo Borealis são especificados através de parâmetros passados na linha de comando.

Para fins de testes e validação uma pequena aplicação para a geração de carga sintética foi desenvolvida. O *dummysender* é capaz de gerar registros de fluxo em três modos: a partir de um arquivo texto; aleatoriamente; ou registros com conteúdo fixo definidos no código. O número de registros a serem gerados pode ser definido via linha de comando, bem como a fila em memória onde os fluxos devem ser armazenados. Como os fluxos a serem gerados são conhecidos *a priori* torna-se trivial a comparação com os resultados gerados pelo Borealis.

4.1.1.1 Esquema de Entrada

O esquema de entrada do Borealis foi definido com base em um subconjunto dos dados fornecidos pelo Netflow. O esquema de entrada definido no formato XML encontra-se na Figura 4.2. Cada registro de fluxo contém as informações descritas na Tabela 4.1.

```
<?xml version="1.0"?>
<!DOCTYPE borealis SYSTEM "borealis.dtd">
<borealis>
    <input   stream="flow"     schema="FlowSchema"     />
    <schema name="FlowSchema">
        <field name="time"      type="int" />
        <field name="id"        type="string" size="15" />
        <field name="src"       type="string" size="15" />
        <field name="sport"     type="int" />
        <field name="dst"       type="string" size="15" />
        <field name="dport"     type="int" />
        <field name="prot"      type="int" />
        <field name="iif"       type="int" />
        <field name="oif"       type="int" />
        <field name="src_as"    type="int" />
        <field name="dst_as"    type="int" />
        <field name="pkts"      type="long" />
        <field name="octets"    type="long" />
    </schema>
</borealis>
```

Figura 4.2: Definição do esquema de entrada

campo	Descrição
time	Timestamp do momento de geração do fluxo.
id	Identificador do roteador que observou o fluxo. Normalmente é utilizado o endereço IP do roteador como identificador.
src	Endereço IP de origem do fluxo.
sport	Porta de origem do fluxo.
dst	Endereço IP de destino do fluxo.
dport	Porta de destino do fluxo.
prot	Protocolo do fluxo (tcp, udp, icmp).
iif	Interface de entrada do fluxo. É o índice de interface por onde o fluxo entrou no roteador.
oif	Interface de saída do fluxo. É o índice da interface por onde o fluxo deixou o roteador.
src_as	Número do Sistema Autônomo de origem do fluxo.
dst_as	Número do Sistema Autônomo de destino do fluxo.
pkts	Número de pacotes contabilizados no fluxo.
octets	Número de octetos contabilizados no fluxo.

Tabela 4.1: Campos dos registros de fluxo de entrada.

Embora a definição do esquema de entrada se baseie principalmente no formato do Netflow, ele ainda permanece genérico o suficiente para que outras fontes de dados, como por exemplo o Sflow, possam ser utilizadas, bastando para tanto escrever uma aplicação capaz de transformar os dados para o esquema de entrada. Até mesmo fontes que não possuam todos os dados necessários podem ser utilizadas, desde que uma certa perda na precisão das consultas seja admitida. Por exemplo, uma fonte de dados que não seja

capaz de fornecer endereços de origem e destino do fluxo, poderia ser utilizada. Consultas como o número de pacotes ou octetos poderiam ser feitas sem prejuízo para a precisão do resultado, embora não fosse possível efetuar consultas que utilizassem filtros baseados no IP de origem e/ou destino. Tendo em vista a vasta gama de dados que podem ser obtidas do *Netflow*, justifica-se a escolha deste formato.

4.1.2 Aplicações de Controle

São duas as aplicações de controle: o *BigGiantHead* e o *BigMouth*. O *BigGiantHead* é uma aplicação que acompanha o Borealis e é utilizada para controlar os nodos Borealis. O *BigGiantHead* pode ser utilizado de duas maneiras: transiente ou persistente. No modo transiente, ele é chamado por uma aplicação cliente do Borealis e finalizado tão logo tenha lido o arquivo XML que contém a definição dos fluxos e consultas, e tenha feito o *deploy* dessas consultas, ou seja, tenha informado aos nodos Borealis o que cada um deve executar. No modo persistente, o *BigGiantHead* funciona como um *daemon*, e é capaz de se comunicar tanto com nodos Borealis quanto com outras aplicações clientes, permitindo o registro de fluxos e consultas.

Na arquitetura proposta, o *BigGiantHead* é executado em modo persistente. O usuário do sistema utiliza uma aplicação cliente, o *BigMouth*, para informar ao *BigGiantHead* o que o sistema como um todo deve executar. A Figura 4.3 mostra a interação das aplicações. O funcionamento do *BigMouth* é bastante simples, bastando informar via linha de comando o endereço e porta de destino do *BigGiantHead* e um arquivo XML contendo as informações de entradas, consultas e saídas. A comunicação entre os dois é feita pela rede, o que permite que sejam executados em máquinas distintas.

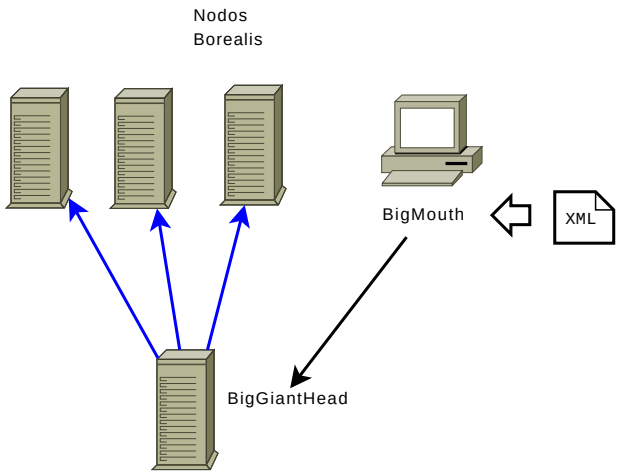

Figura 4.3: Interação das aplicações de controle.

4.1.3 Aplicação de Apresentação

As informações processadas pelos nodos Borealis são enviadas utilizando o protocolo de comunicação do Borealis. A aplicação cliente do Borealis deve ser capaz de decodificá-las para poder exibí-las ao usuário. Na forma mais simples de utilização do Borealis, a aplicação que envia dados e recebe os resultados é, comumente, a mesma. No entanto, dadas as características da arquitetura proposta, a aplicação de apresentação deve ser distinta, o que permite a sua execução em um nodo diferente dos nodos que fazem a aquisição e envio dos fluxos.

Como o esquema de saída não é fixo, isto é, ele depende da consulta a ser realizada, a aplicação de visualização deve ser capaz de inferí-lo. Por este motivo a aplicação foi chamada de *ureceiver* (universal receiver). Para executar o *ureceiver* é necessário informar via linha de comando a porta onde a aplicação deve aceitar conexões dos nodos Borealis para receber os resultados, bem como o arquivo XML contendo a consulta a ser feita, para que a inferência do esquema de saída seja realizada.

Ao ser executado, o *ureceiver* aguarda conexões dos nodos Borealis, e mostra a cada intervalo de tempo definido na consulta, os resultados gerados pelo Borealis.

O código das ferramentas está listado no Anexo A.

4.2 Validação e Avaliação

Para validar e avaliar o desempenho da ferramenta proposta três cenários de testes foram executados, e serão descritos a seguir.

4.2.1 Teste 1: Único Nodo Borealis e Carga Sintética

O primeiro teste executado teve por objetivo validar a ferramenta. Uma carga sintética foi gerada e o resultado retornado foi comparado com o resultado esperado. O teste permitiu também avaliar o desempenho do nodo Borealis. A topologia montada para o primeiro teste é mostrada na Figura 4.4.

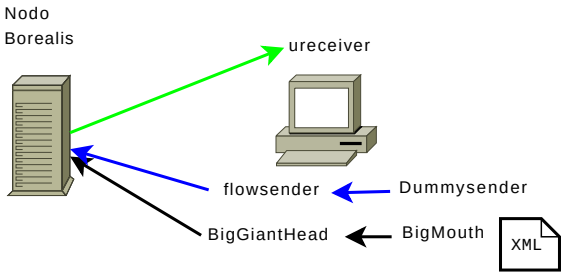

Figura 4.4: Teste 1: validação

O ambiente de testes foi composto por duas máquinas: o nodo Borealis e o nodo cliente. A máquina utilizada como nodo Borealis possuia uma CPU Athlon XP 2600+ e 1 GB de memória. A máquina cliente, onde foi executado o *BigGiantHead*, *dummysender*, *flowsender* e *ureceiver*, possuia uma CPU Celeron 900 e 2GB de memória. Ambas as máquinas estavam conectadas em uma rede local *Ethernet* através de um switch de 100Mbps.

Para verificar se as respostas produzidas pelo sistema eram corretas foram gerados registros de fluxos a velocidades pré-determinadas, variando de mil a 40 mil registros por segundo em intervalos de mil registros. Uma consulta simples que contabilizava o número de pacotes de todos os registros a cada janela de 10 segundos foi empregada. Como o contéudo de cada registro era igual e previamente conhecido, a verificação da precisão da resposta foi trivial.

Os teste foram repetidos 10 vezes e os resultados apresentados representam o conjunto de resultados obtidos. Foi observado que o ponto de saturação do sistema se situou em torno dos 35 mil registros por segundo. Ao se atingir essa taxa de geração de fluxos, o nodo Borealis atingia uma condição de erro e parava de contabilizar pacotes. O resultado obtido é compativel com um experimento similar descrito em [20]. Para todas as velocidades testadas, abaixo do ponto do saturação, o sistema produziu respostas corretas.

Durante os testes, a utilização de CPU e da rede foram medidas no nodo Borealis. A Figura 4.5 mostra a utilização de CPU no nodo Borealis para cada velocidade de transmissão de registros.

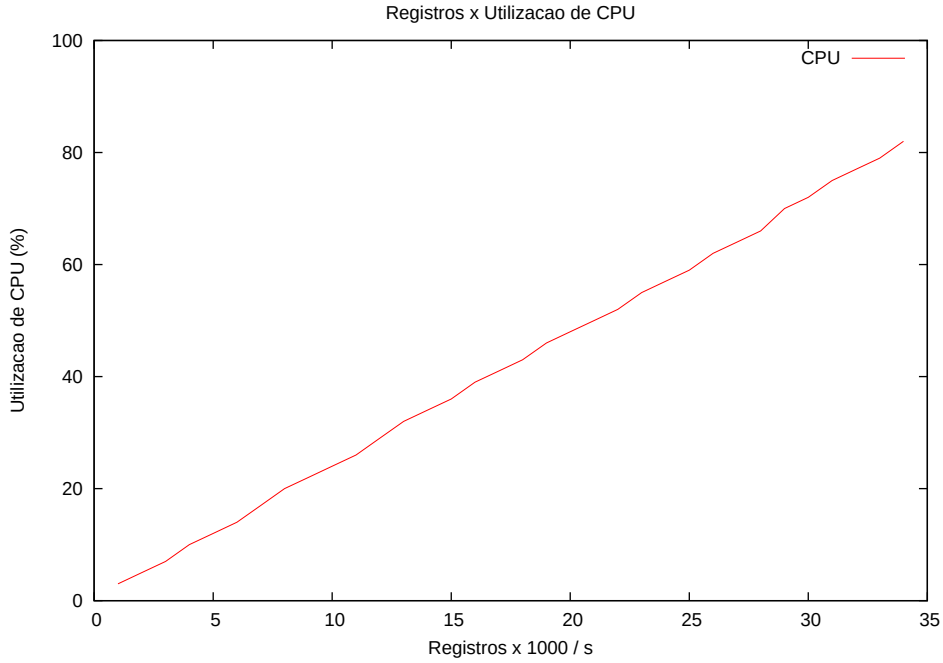

Figura 4.5: Registros/s Vs. Utilização de CPU.

A Figura 4.6 mostra a utilização da rede no nodo Borealis para cada velocidade de transmissão de registros.

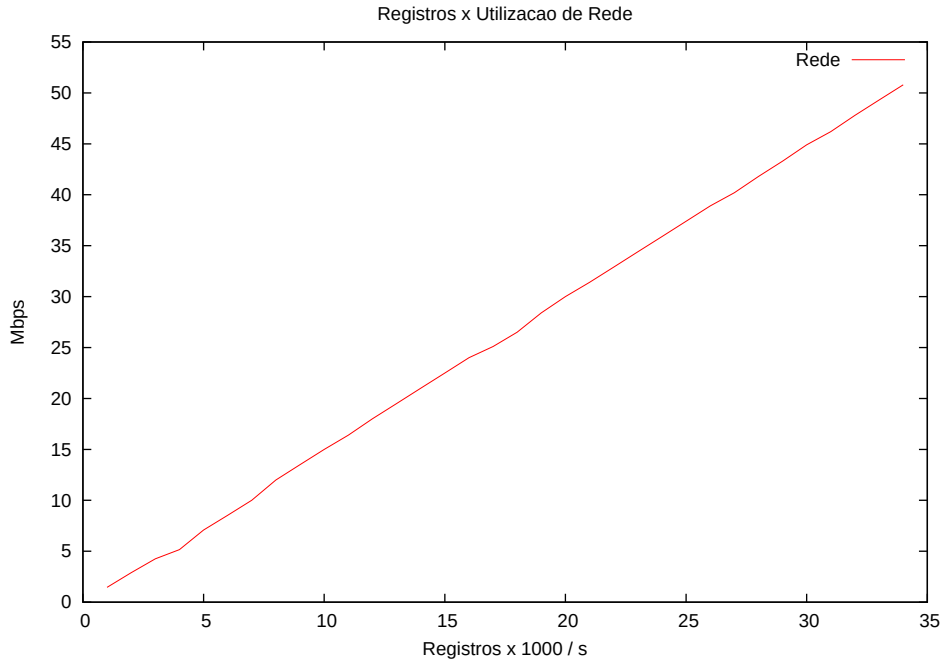

Figura 4.6: Registros/s Vs. Utilização da rede.

Neste teste a transformação do formato original da fonte de dados para o formato utilizado pelo Borealis e o envio de dados para o nodo Borealis, ou seja, as tarefas desempenhadas pelo *nnfc* e *flowsender* foram executadas em uma máquina que não era o nodo Borealis, logo o tráfego de rede observado corresponde a transmissão dos registros utilizando o protocolo de comunicação do Borealis. É possível evitar esse tráfego na rede ao executar o *nnfc* e o *flowsender* no próprio nodo Borealis. O Teste 2, descrito a seguir, utiliza essa estratégia.

4.2.2 Teste 2: Múltiplos Nodos Borealis e Carga Sintética

O primeiro teste identificou um limite na quantidade de registros processados por segundo por um único nodo Borealis. O objetivo do segundo teste era o de determinar se múltiplos nodos Borealis poderiam ser combinados para processar um número maior de registros. A topologia utilizada no segundo teste é mostrada na Figura 4.7.

O ambiente de testes foi composto por 3 nodos Borealis e um nodo cliente. As máquinas utilizadas como nodos Borealis possuiam CPUs Athlon XP 2600+, Athlon XP 2400+ e Athlon XP 2200+, e 1GB de memória cada. A máquina cliente foi a mesma

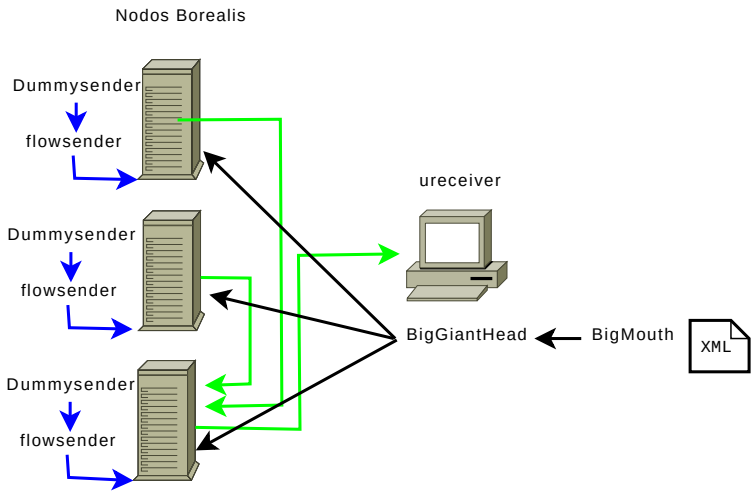

Figura 4.7: Teste 2: Múltiplos nodos Borealis.

empregada no teste anterior, isto é, uma CPU Celeron 900 com 2GB de memória.

Neste teste cada nodo Borealis executou também o *dummysender* e o *flowsender* para que os registros de fluxo fossem gerados localmente. Na máquina cliente foi executada a aplicação *BigGiantHead*, além do *ureceiver*.

A consulta de contabilização de pacotes utilizada no primeiro teste foi modificada para que cada nodo Borealis contabilizasse os seus pacotes. Os resultados parciais produzidos por cada nodo foram enviados para o primeiro nodo para que o resultado total fosse contabilizado por este. O diagrama de caixas da consulta é mostrado na Figura 4.8. O operador *Aggregate* corresponde a funcionalidade descrita na Seção 3.2.2.2.

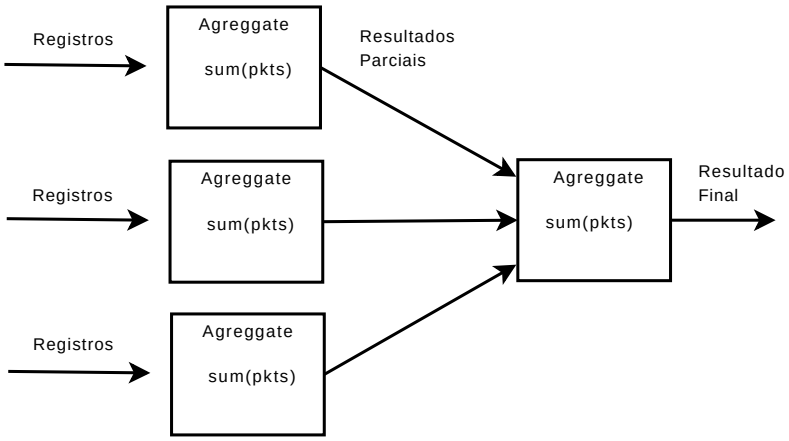

Figura 4.8: Teste 2: Diagrama de consulta.

Em cada nodo Borealis os registros de fluxo foram gerados a uma velocidade de 30 mil registros por segundo, e o resultado recebido pelo ureceiver apresentava a resposta correta esperada para o total de 90 mil pacotes por segundo em todas as 10 repetições do teste.

4.2.3 Teste 3: Carga Real

O terceiro teste teve como objetivo avaliar a utilização da ferramenta com uma carga real. O teste foi executado no Ponto de Presença no Paraná (PoP-PR) da Rede Nacional de Ensino e Pesquisa (RNP). No PoP-PR a contabilização de tráfego utilizando *Netflow* é feita com o auxílio de uma máquina que observa de maneira passiva o tráfego em uma porta "espelhada"de um switch. Uma aplicação chamada *nprobe* [10] gera os registros no formato *netflow* que são enviados para outras máquinas que coletam, armazenam e processam esses registros. A máquina que gera os registros *Netflow* foi configurada de forma a replicar os fluxos recebidos, enviando uma cópia para um nodo Borealis. A Figura 4.9 mostra uma visão geral deste cenário.

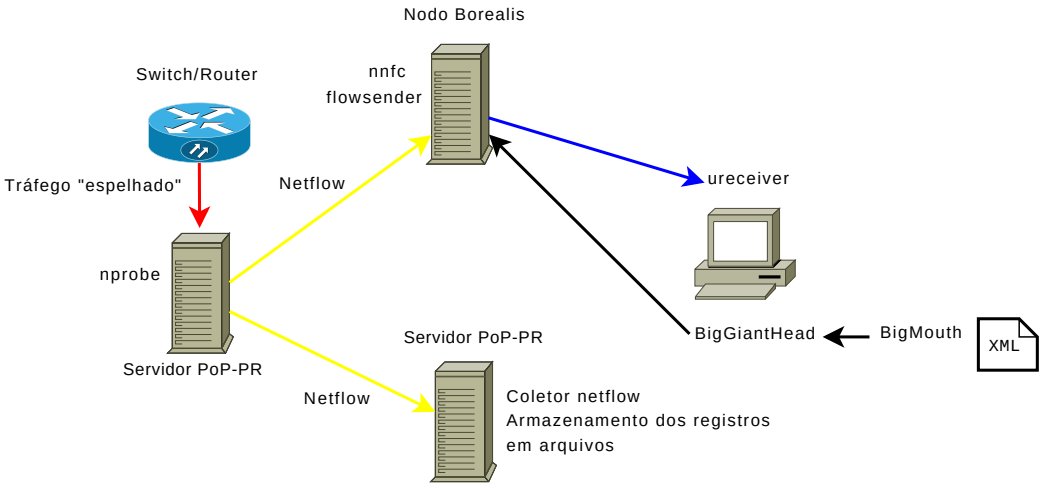

Figura 4.9: Teste 3: Visão geral.

Como a máquina do PoP-PR que armazena os registros de *Netflow* o faz gravando os fluxos recebidos a cada intervalos de 5 minutos em arquivos, optou-se também por usar nas consultas feitas no nodo Borealis janelas de 5 minutos (300 segundos), pois isso permitiria que os resultados obtidos com o Borealis fossem confrontados com os resultados

obtidos do processamento dos arquivos.

A carga real, ou seja o fluxo de registros *Netflow*, provida pela máquina que captura o tráfego, tem uma característica bem distinta da carga sintética usada até então. Enquanto que a carga sintética foi gerada de maneira contínua e a uma velocidade determinada, a carga real se apresenta na forma de rajadas. A cada intervalo de aproximadamente 30 segundos um conjunto de registros é enviado. A Figura 4.10 mostra a taxa de chegada de registros por segundo em um intervalo de tempo de 5 minutos.

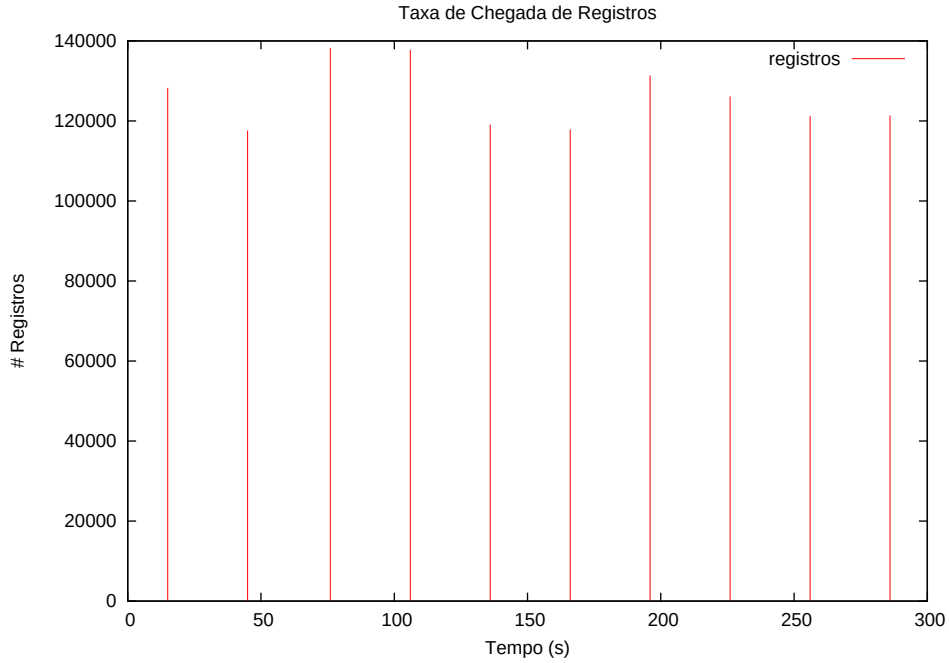

Figura 4.10: Teste 3: Chegada de novos registros por segundo.

O tráfego observado pela máquina que gera os registros de *Netflow* é da ordem de 1Gbps e 100 mil pacote/s. Quando esse tráfego é sumarizado em registros *Neflow* ele gera uma quantidade de registros da ordem de 1 milhão de registros a cada intervalo de 5 minutos.

A consulta utilizada nos dois primeiros testes foi repetida nesse novo cenário utilizando as mesmas máquinas do Teste 1. Os testes preliminares mostraram que a máquina utilizada como nodo Borealis não conseguia decodificar os registros *Netflow* com velocidade o suficiente e acabava descartando uma certa quantidade de pacotes UDP com informações do *Netflow* a cada rajada. A perda de pacotes se refletia em uma diferença entre os val-

ores dos resultados providos pelo Borealis e os valores obtidos processando os arquivos de fluxo. Para quantificar a quantidade de registros não processados foi criada uma nova consulta que simplesmente contava o número de registros a cada janela de tempo. A Figura 4.11 mostra o número de registros contabilizados pelo Borealis e o número de registros armazenados em arquivos para cada intervalo de 5 minutos.

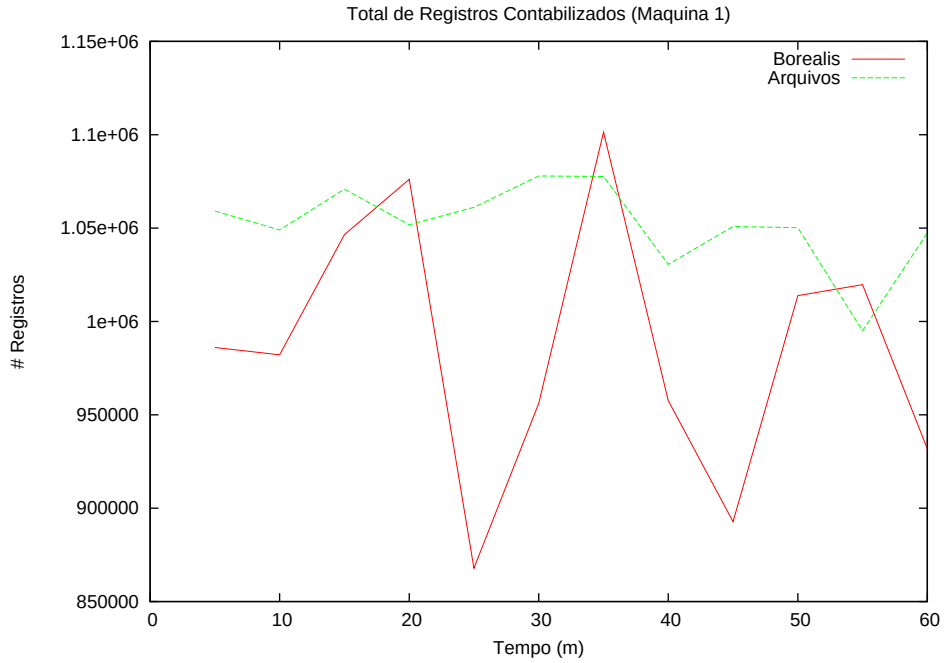

Figura 4.11: Teste 3: Número de registros contabilizados pelo Borealis (Máquina 1).

Os dados obtidos nos primeiros testes nesse cenário pareciam indicar que uma máquina mais potente seria necessária. O teste foi então repetido utilizando como nodo Borealis uma máquina com uma CPU Intel Core 2 Duo de 2Ghz e 1 GB de memória RAM. A Figura 4.12 mostra o número de registros contabilizados por esta máquina e o número de registros armazenados em arquivos para cada intervalo de 5 minutos durante uma hora de execução do teste.

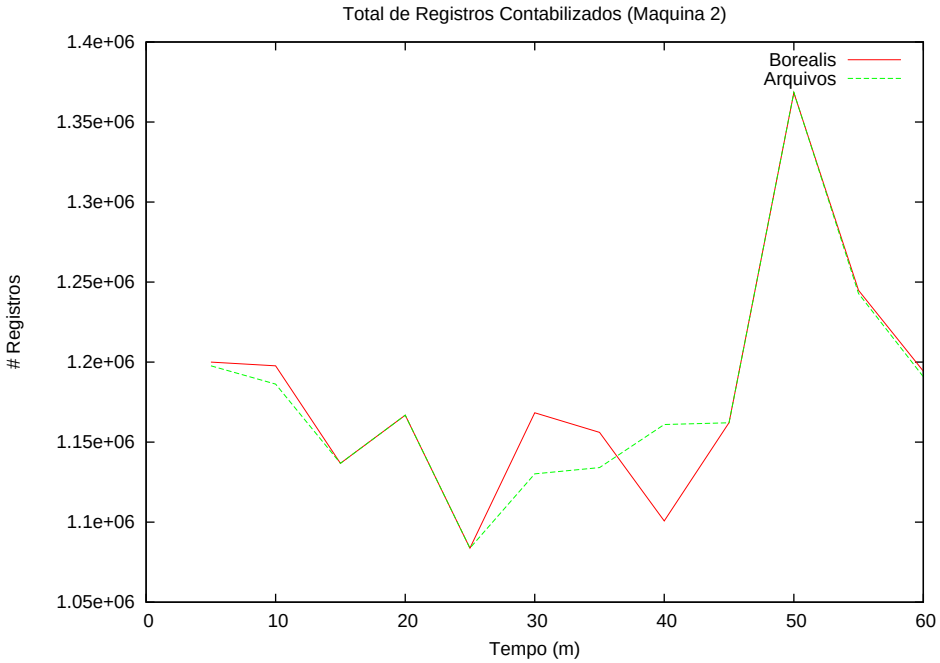

Figura 4.12: Teste 3: Número de registros contabilizados pelo Borealis (Máquina 2).

As pequenas diferenças observadas entre os valores são devido ao fato de que a ferramenta que armazena os registros em arquivos utiliza a hora local da máquina para determinar o conteúdo de cada arquivo, enquanto que o Borealis utiliza um valor de *timestamp* do próprio registro. Isto é, ao término de um intervalo de 5 minutos, contados pelo relógio da máquina local, o coletor *Netflow* armazena em um arquivos todos os registros recebidos até então. É possível que esse intervalor de 5 minutos aconteça durante uma rajada de pacotes. Já o Borealis não utiliza um relógio local, mas considera tão somente a informação de *timestamp* presente no registro. Sendo assim, é possível que o Borealis processe um pacote com um determinado *timestamp* X mesmo que a hora local já seja X+1. A Figura 4.13 apresenta os resultados da consulta que conta o total de pacotes de todos os registros de fluxos.

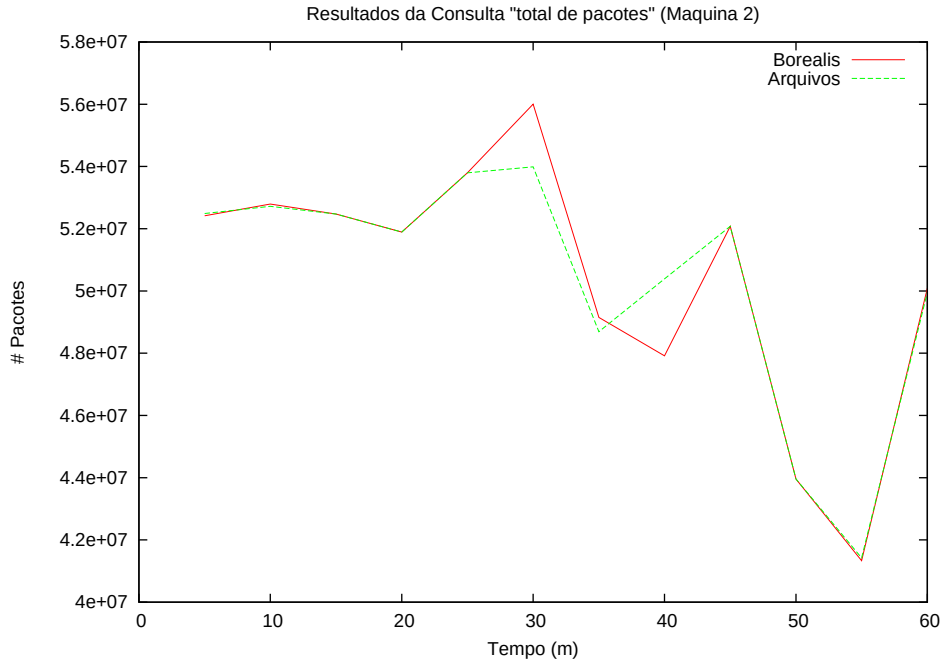

Figura 4.13: Teste 3: Resultados da consulta “total de pacotes”.

4.3 Estudos de Caso

Como forma de demonstrar a funcionalidade da ferramenta proposta três estudos de caso, ou cenários, de aplicação são descritos. Cada estudo de caso corresponde a tarefas comuns na gerência de backbones.

4.3.1 Matriz de Tráfego

Uma matriz de tráfego é o conjunto de valores de tráfego medidos dois a dois dentre os participantes de uma determinada rede. A matriz de tráfego é particulamente útil dentro de um Ponto de Troca de Tráfego, uma estrutura na qual diferentes Sistemas Autônomos (AS) trocam tráfego entre si. As informações de quanto tráfego cada participante envia para cada outro participante é importante para a área de engenharia de tráfego de cada Sistema Autônomo e permite a definição de melhores políticas de roteamento. A obtenção da matriz de tráfego é bastante simples pois basta agregar os fluxos por AS de origem e Destino contabilizando o número de octetos. Sendo assim o problema é resolvido com uma único operador do tipo *Aggregate*. O diagrama de consulta no formato XML é apresentado

na Figura 4.14:

```
<borealis>
    <output  stream="matrixresult"   schema="matrixSchema"   />
    <schema name="matrixSchema">
        <field name="time" type="int" />
        <field name="as1"       type="int" />
        <field name="as2"       type="int" />
        <field name="toctets"   type="long" />
    </schema>
    <box name="pktsum"   type="aggregate" >
         <in     stream="flowas" />
         <out    stream="matrixresult" />

         <parameter name="aggregate-function.0" value="sum(pkts)" />
         <parameter name="aggregate-function-output-name.0" value="toctets" />
         <parameter name="window-size-by"        value="VALUES" />
         <parameter name="window-size"           value="300"    />
         <parameter name="advance"               value="300"    />
         <parameter name="timeout"               value="330"    />
         <parameter name="group-by"              value="src_as,dst_as"/>
         <parameter name="order-by"              value="FIELD"  />
         <parameter name="order-on-field"        value="time"   />
         <parameter name="independent-window-alignment" value="1" />
         <parameter name="drop-empty-outputs"        value="0" />
   </query>
</borealis>
```

Figura 4.14: Definição da consulta matriz de tráfego.

Ao ser executada, a consulta gera, a cada janela de tempo, resultados contendo o *timestamp* da janela, o AS de origem, o AS de destino e a quantidade de octetos. A Figura 4.15 apresenta um exemplo de saída de resultados da consulta.

```
1217257500 65006 65001 16488787
1217257500 65000 65001 24356097
1217257500 65001 65000 572393822
1217257500 65009 65001 8272830
1217257500 65001 65009 279961720
1217257800 65000 65001 71098238
1217257800 65001 65000 1344944043
1217257800 65006 65001 34283545
1217257800 65009 65001 18086315
1217257800 65001 65009 627027711
```

Figura 4.15: Resultados da consulta matriz de tráfego.

4.3.2 Detecção de Anomalias

O segundo exemplo de utilização mostra como utilizar a ferramenta para detectar uma anomalia de tráfego conhecida como varredura de rede. Um varredura de rede acontece quando um host envia pacotes para vários outros hosts de uma rede com objetivo de descobrir quais respondem e quais não. A característica desses tipos de sonda é a grande quantidade de fluxos com apenas 1 pacote (em geral apenas um pacote UDP, ou um pacote ICMP é enviado). Para detectar esse tipo anomalia a consulta deverá primeiro filtrar os fluxos com apenas 1 pacote. Em seguida agregar os fluxos por endereço IP de origem e destino, de forma a selecionar tuplas distintas de origem/destino. Uma nova agregação é feita agrupando apenas pela origem, contando assim os destinos distintos. Opcionalmente é possível estabelecer um valor de "corte"ou seja, hosts que produziram até um determinado número de fluxos com apenas 1 pacote e que podem ser considerados "normais". O diagrama de consulta no formato XML é mostrado na Figura 4.16.

```
<borealis>
   <output  stream="anomalyresult"   schema="anomalySchema"   />
    <schema name="anomalySchema">
         <field name="time"            type="int" />
         <field name="src"             type="string" size="15" />
         <field name="flows"           type="int" />
    </schema>
    <query name="anomalydetect" >
    <box name ="filter_in"  type="filter" >
         <in   stream="flow" />
         <out  stream="intermediate1" />
         <parameter name="expression.0"  value="sport != 53 and pkts=1"    />
         <parameter name="pass-on-false-port" value="0"        />
    </box>
    <box name="anomalyag1"   type="aggregate" >
         <in     stream="intermediate1" />
         <out    stream="intermediate2" />
         <parameter name="aggregate-function.0" value="count()" />
         <parameter name="aggregate-function-output-name.0" value="tmp" />
         <parameter name="window-size-by"        value="VALUES"   />
         <parameter name="window-size"           value="300"        />
         <parameter name="advance"               value="300"        />
         <parameter name="timeout"               value="300"        />
         <parameter name="group-by"              value="src,dst"/>
         <parameter name="order-by"              value="FIELD"    />
         <parameter name="order-on-field"        value="time"     />
         <parameter name="independent-window-alignment"        value="1"     />
         <parameter name="drop-empty-outputs"        value="0"     />
   </box>
```

```
   <box name="anomalyag2"   type="aggregate" >
         <in     stream="intermediate2" />
         <out    stream="intermediate3" />
         <parameter name="aggregate-function.0" value="count()" />
         <parameter name="aggregate-function-output-name.0" value="flows" />
         <parameter name="window-size-by"        value="VALUES"   />
         <parameter name="window-size"           value="300"        />
         <parameter name="advance"               value="300"        />
         <parameter name="timeout"               value="300"        />
         <parameter name="group-by"              value="src"/>
         <parameter name="order-by"              value="FIELD"    />
         <parameter name="order-on-field"        value="time"     />
         <parameter name="independent-window-alignment"        value="1"     />
         <parameter name="drop-empty-outputs"        value="0"     />
   </box>
   <box name="limitscans"  type="filter" >
           <in   stream="intermediate3" />
           <out  stream="intermediate4"   />
           <parameter name="expression.0"        value="flows > 500" />
           <parameter name="pass-on-false-port" value="0"        />
    </box>
    <box name="showscans"  type="map" >
           <in   stream="intermediate4" />
           <out  stream="anomalyresult"   />
           <parameter name="expression.0"        value="time" />
           <parameter name="output-field-name.0" value="time" />
           <parameter name="expression.1"        value="src" />
           <parameter name="output-field-name.1" value="src" />
           <parameter name="expression.2"        value="flows" />
           <parameter name="output-field-name.2" value="flows" />
    </box>
   </query>
</borealis>
```

Figura 4.16: Definição da consulta para detectar varreduras de rede.

A consulta apresentada aqui é passível de produzir falsos positivos. Servidores DNS por exemplo produzem um grande número de fluxos com apenas um pacote. No entanto, o resultado obtido combinado com a experiência do administrador de redes é o suficiente para rapidamente identificar esses casos. Uma outra possibilidade, utilizada aqui, é filtrar as consultas a servidores DNS, bastando para tanto filtrar os fluxos com origem ou destino à porta 53.

Ao ser executada, a consulta gera, a cada janela de tempo, resultados contendo o *timestamp* da janela, o endereço IP de origem, e a quantidade endereços IP de destino distintos. A Figura 4.17 apresenta um exemplo de saída de resultados da consulta.

```
1217260500 10.36.11 1613
1217260500 10.107.209.1 577
1217260500 10.107.241.100 818
1217260500 10.107.230.63 747
1217260500 10.17.232.196 620
```

Figura 4.17: Resultados da consulta detectação de varreduras de rede.

4.3.3 Detecção de DDoS

Um tipo de ataque efetuado contra redes de computadores é aquele que tenta exaurir um recurso da rede ou de um servidor, são os chamados "ataques de negação de serviço"ou DoS (do inglês, *Deny of Service*). Um dos métodos utilizados em um DoS é o envio de uma grande quantidade de pacotes ICMP ou UDP fazendo com que um servidor ou um roteador passe a utilizar recursos da CPU processando estes pacotes e não pacotes legítimos.

Uma forma ainda mais danosa de ataque DoS é o DDoS ou *Distributed Deny of Service*. Nessa forma de DoS várias máquinas são empregadas no ataque contra um único alvo. Uma das dificuldades na contenção de um DDoS é devido ao fato de que usualmente os pacotes utilizados no ataque tem o seu endereço IP de origem falsificado, fazendo-os parecer vir de um local que não é local verdadeiro. No exemplo apresentado aqui supõe-se que o tipo de pacote utilizado e o alvo do ataque já foi identificado (o ataque pode ter sido identificado diretamente no alvo), restando a tarefa de identificar as fontes

do ataque. Como não é possivel confiar nos endereços IP de origem dos pacotes, se faz necessário identificar os roteadores e as interfaces por onde o ataque está ingressando na rede. Conhecendo os pontos de entrada do ataque na rede é possível tomar medidas para contê-lo, como por exemplo, aplicar filtros nestas interfaces e notificar os administradores responsáveis pelas redes conectadas a estas mesmas interfaces.

Uma possível consulta para detectar um determinado DDoS deve iniciar filtrando os pacotes que fazem parte do ataque, por exemplo, pacotes com destino à porta UDP/4000 do host 192.168.0.5. Em seguida os fluxos pertencentes a este ataque poderiam ser agregados contabilizando-se o número de origens distintas, agrupando-os por roteador e interface de origem. Os resultados produzidos por vários nodos Borealis poderiam ser enviados a um único nodo Borealis para que fossem agrupados e enviados à aplicação cliente. O Figura 4.18 apresenta uma consulta deste tipo, considerando uma rede com 2 nodos Borealis.

```
<borealis>
    <output  stream="ddosresult"   schema="ddosSchema"   />
    <schema name="ddosSchema">
         <field name="time"             type="int" />
         <field name="id"              type="string" size="15" />
         <field name="iif"             type="int" />
         <field name="srcs"            type="int" />
    </schema>
    <query name="ddosdetect_r1" >
    <box name ="filter_in_r1"  type="filter" >
         <in   stream="flow" />
         <out  stream="intermediate1_r1" />

         <parameter name="expression.0"  value="dst='192.168.0.5' and dport=4000"    />
         <parameter name="pass-on-false-port" value="0"        />
    </box>
```

```
    <box name="ddosag1"   type="aggregate" >
         <in     stream="intermediate1_r1" />
         <out    stream="intermediate2_r1" />

         <parameter name="aggregate-function.0" value="count()" />
         <parameter name="aggregate-function-output-name.0" value="srcs" />
         <parameter name="window-size-by"        value="VALUES"    />
         <parameter name="window-size"           value="300"       />
         <parameter name="advance"               value="300"       />
         <parameter name="timeout"               value="300"       />
         <parameter name="group-by"              value="id,iif"       />
         <parameter name="order-by"              value="FIELD"     />
         <parameter name="order-on-field"        value="time"      />
         <parameter name="independent-window-alignment" value="1"  />
         <parameter name="drop-empty-outputs"    value="1"         />
    </box>

    </query>
    <query name="ddosdetect_r2" >
    <box name ="filter_in_r2"  type="filter" >
         <in   stream="flow" />
         <out  stream="intermediate1_r2" />

         <parameter name="expression.0"  value="dst='192.168.0.5' and dport=4000"    />
         <parameter name="pass-on-false-port" value="0"        />
    </box>
    <box name="ddosag2"   type="aggregate" >
         <in     stream="intermediate1_r2" />
         <out    stream="intermediate2_r2" />

         <parameter name="aggregate-function.0" value="count()" />
         <parameter name="aggregate-function-output-name.0" value="srcs" />
         <parameter name="window-size-by"        value="VALUES"    />
         <parameter name="window-size"           value="300"       />
         <parameter name="advance"               value="300"       />
         <parameter name="timeout"               value="300"       />
         <parameter name="group-by"              value="id,iif"       />
         <parameter name="order-by"              value="FIELD"     />
         <parameter name="order-on-field"        value="time"      />
         <parameter name="independent-window-alignment" value="1"  />
         <parameter name="drop-empty-outputs"    value="1"         />
    </box>
    <query>
    <query name="ddosdetect" >
    <box name ="tudo"  type="union" >
         <in   stream="intermediate1_r1"      />
         <in   stream="intermediate1_r2"      />
         <out  stream="ddosresult"            />
   </box>
   </query>
</borealis>
```

Figura 4.18: Definição da consulta para identificação de origens de um DDoS

Ao ser executada, a consulta gera, a cada janela de tempo, resultados contendo o *timestamp* da janela, o identificador do roteador, o índice da interface de entrada e a quantidade de fluxos contabilizados. A Figura 4.19 apresenta um exemplo de saída de resultados da consulta.

```
1217268000 10.0.0.1 1 36
1217268000 10.0.0.5 3 24
1217268300 10.0.0.1 1 45
1217268300 10.0.0.5 3 23
```

Figura 4.19: Resultados da consulta para identificação de origens de um DDoS.

Capítulo 5

Conclusão

O presente trabalho demonstrou a possibilidade de utilização de Sistema Gerenciadores de Streams de Dados na análise de tráfego de rede em tempo real. Uma ferramenta que utiliza o SGSD Borealis e *Netflow* foi implementada, validada e o seu desempenho foi avaliado. Os resultados, tanto em desempenho, quanto em funcionalidade, demonstram a viabilidade da proposta. Os resultados experimentais utilizando um ambiente de produção demonstraram que é possível processar mais de um milhão de registros netflow a cada intervalo de 5 minutos, e que os resultados são tão precisos quanto aos obtidos pelo métodos tradicionais que utilizam o armazenamento dos fluxos em arquivos e processamento "off-line"destes.

Quanto a funcionalidade, foram desmonstrados 3 estudos de casos que demonstraram como é possivel com única ferramenta solucionar problemas distintos: matriz de tráfego, detecção de varreduras de redes, e identificação de origens de ataques de negação de serviço.

5.1 Considerações e Trabalhos Futuros

Durante a implementação e testes da ferramenta alguns problemas e limitações foram encontrados, e serão apresentados a seguir. Sugestões para trabalhos futuros também são sugeridos.

5.1.1 Sobre a Ferramenta

Embora a ferramenta proposta tenha cumprido o seu papel ao demonstrar a possibilidade da utilização de SGSD para a análise de tráfego de redes, ela admite melhorias em vários pontos. A seguir alguns destes pontos serão destacados e discutidos.

A *Criação de Consultas* pode ser melhorada com o auxílio de uma ferramenta visual para a criação de consultas, evitando assim que o administrador de redes tenha que escrever os arquivos de consulta utilizando diretamente o XML.

A *Visualização dos Resultados* é feita através de uma saída em formato texto. Ferramentas que recebam o resultado em formato texto e criem gráficos a partir destes, poderiam ser criadas.

Há também a possibilidade de se escrever aplicações para a utilização de outras fontes de dados, e que poderiam ampliar as possibilidades de uso da ferramenta. Dados capturados diretamente da interface de rede poderiam ser utilizados, desde que convertidos com o auxílio de uma aplicação, ao formato de entrada esperado pelo Borealis.

5.1.2 Sobre o Borealis

A versão do Borealis utilizada foi a *Winter 2007*. Algumas melhorias, baseadas em dificuldades encontradas ao utilizá-lo, são sugeridas a seguir.

Um nodo Borealis ao atingir uma certa quantidade de registros por segundo apresenta uma condição de erro a partir da qual ele para de contabilizar registros, mesmo que a quantidade de registros por segundo retorne a uma velocidade "normal". Uma possível melhoria seria a de mudar esse comportamento, fazendo emitir um alerta ao se chegar a tal condição, e fazendo-o descartar automáticamente os pacotes extras à sua capacidade. Assim, os resultados seria afetados apenas enquanto durar a condição de sobrecarga.

O sistema Borealis não admite a remoção de consultas. Assim uma consulta registrada em um nodo Borealis continuará a ser processada enquanto esse nodo estiver ativo. É necessário parar e re-iniciar a execução do processo Borealis para que a consulta seja removida. A consulta também é persistente no *BigGiantHead*, sendo necessário parar e re-iniciá-lo também.

O Borealis permite que novas "caixas", utilizadas nas consultas, sejam escritas. Isso torna possível que uma série de novas funcionalidades sejam acrescentadas ao sistema. Com as caixas padrões do sistema não é possível, por exemplo, a ordenação de resultados dentro de uma determinada janela. Uma contribuição ao sistema seria extender o operador *Agreggate* para que ele pudesse efetuar tal função. Outra melhoria seria a criação de uma caixa que selecione apenas N resultados dentro de uma determinada janela. A implementação destas duas funcionalidades permitiriam, por exemplo, a criação de consultas que identifiquem os "top ten", ou os 10 melhores ou piores resultados de uma determinada consulta.

Referências Bibliográficas

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

[2] Sociedade Brasileira para Interconexão de Sistemas Abertos BRISA. *Gerencimanento de Redes - Uma abordagem de sistemas abertos*. Makron, s.d.

[3] CAIDA. cflowd tools. `http://www.caida.org/tools/measurement/cflowd`, 1998.

[4] G. Munz; G. Carle. Distributed Network Analysis Using TOPAS and Wireshark. *Network Operations and Management Symposium Workshops, 2008.*, pp. 161–164, 2008.

[5] G. Combs. wireshark. `http://www.wireshark.org`.

[6] M. Balazinska et al. D. Abadi, Y. Ahmad. The Design of the Borealis Stream Processing Engine. *Proceedings of the 2005 CIDR Conference*, 2005.

[7] N. Koudas D. Srivastava. Data Stream Query Processing: A Tutorial. *VLDB*, 2003.

[8] A. Zhou; Y. Yan; X. Gong; J. Chang; D. Dai. SMART: A System for Online Monitoring Large Volumes of Network Traffic. *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pp. 1576–1579, 2008.

[9] Rede Nacional de Ensino e Pequisa. Mapa do Backbone RNP. `http://www.rnp.br/backbone/index.php`, setembro de 2007.

[10] Lucas Deri. nProbe. `http://www.ntop.org/nProbe.html`.

[11] A. Arasu et al. STREAM: The Stanford Data Stream Management System. *IEEE Data Engineering Bulletin*, March 2003.

[12] D. Carney et al. Monitoring streams: a new class of data management applications. *Procedings of the 27th internation conference on very large databases, Hon Kong, August 2002*, pp. 215–226, 2002.

[13] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *SIGMOD Conference*, 2004.

[14] C. Cranor; Y. Gao; T. Johnson; et all. Gigascope: High Performance Network Monitoring with a SQL Interface. *ACM SIGMOD 2002*, pp. 623–627, 2002.

[15] D. Harrington; et all. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. `ftp://ftp.rfc-editor.org/in-notes/rfc3411.txt`, Dezembro de 2002.

[16] D. Harrington; et all. STD62. `ftp://ftp.rfc-editor.org/in-notes/std/std62.txt`, Dezembro de 2002.

[17] R. Presuhn; et all. Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). `ftp://ftp.rfc-editor.org/in-notes/rfc3416.txt`, Dezembro de 2002.

[18] T. Plagemann; V. Goebel; A. Bergamini; et all. 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)*, pp. 215–226, 2004.

[19] M. Fullmer. Flow-tools. `http://www.splintered.net/sw/flow-tools/`, setembro de 2007.

[20] N. Ligocki; C. Hara. Uma Ferramenta de Monitoramento de Redes usando Sistemas Gerenciadores de Streams de Dados. *WRGS, 2006*, 2006.

[21] Cisco Systems Inc. NetFlow Services and Applications - White Paper, 2007.

[22] Andrew Klyachkin. New Netflow Collector. `http://sourceforge.net/projects/nnfc`.

[23] Karen Fang Leinwand, Allan ; Conroy. *Network Management - A Pratical perspective. 2nd edition*. Addison-Wesley, 1996.

[24] H. Balakrishnam et al. M. Cherniack, M. Balazinska. Scalable Distributed Stream Processing. *Proceedings of the 2005 CIDR COnference*, 2005.

[25] V. Jacobson; C. Leres; S. MacCanne. tcpdump. `http://ftp.ee.lbl.gov`.

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

[27] ntop team. NTop. `http://www.ntop.org`, setembro de 2007.

[28] P. Phaal; S. Panchen. Packet Sampling Basics, setembro de 2007.

[29] T. Dubendorfer; A. Wagner; B. Plattner. A Framework for Real-Time Worm Attack Detection and Backbone Monitoring. *Procedings of the 2005 First IEEE Internation Workshop on Critical Infrastructure Protection*, 2005.

[30] D. Plonka. Flowscan. `http://www.caida.org/tools/utilities/flowscan/`.

[31] M. Fullmer; S. Romig. The OSU flow-tools package and CISCO netflow logs. *Procedings of the Forteenth Systems Administration Conference (LISA-00)*, pp. 291–304, 2000.

[32] C. Cranor; T. Johnson; O. Spataschek. Gigascope: A Stream Database for Network Applications. *ACM SIGMOD 2003*, pp. 647–651, 2003.

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

[34] L. Deri; S. Suin. Effective Traffic Measurement Using ntop. *IEEE Communications Magazine*, pp. 138–143, 2000.

[35] N. Duffield; C. Lund; M. Thorup. Properties and Prediction of Flow Statistics from Sampled Packet Streams. *Procedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, Marseille, France, 2002*, pp. 159–171, 2002.

[36] L. Bin; L. Chuang; Q. Jian; H. Jianping; P. Ungsunan. A NetFlow based flow Analisys and Monitoring System in Enterprise Networks. *Computer Networks*, (52):1074–1092, 2008.

Anexo A

A-1 Código Fonte do Plugin para o NNFC

```
/*************************************************************************
 * Arquivo nnfc-0.8.3/plugins/borealis/borealis.c                        *
 *                                                                       *
 * Plugin Borealis para o NNFC                                           *
 *************************************************************************/

#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <syslog.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>

#include <libnnfc.h>

/*
 * Parameters for this plugins are:
 * localtime - store local or remote time?
 * key - IPC key
 */

typedef struct  {
        int  time;
char id[16];
char srcip[16];
        int  sport;
char dstip[16];
        int  dport;
        int  prot;
int  iif;
        int  oif;
        int  srcas;
        int  dstas;
        long pkts;
        long octets;
} message_fmt;
```

```
typedef struct  {
         long    mtype;
         message_fmt message;
         } message_buf;

int     ltime = 1;
int     msqid;
int     msgflg = IPC_CREAT | 0666;
key_t   key=0;
message_buf sbuf;
size_t buf_length;

void _connect(char *fname)
{
    struct command  *commands = 0;
    int             n = 0;

    if (!fname) return;

    if (readcfg(fname, &commands)) {
        syslog(LOG_ERR, "Parse error!\n");
        return;
    }

    CFG_GETI(ltime, "localtime", commands);
    CFG_GETI(key, "messagekey", commands);

    freecmds(commands);
}

void _close()
{
}

void begin()
{
    int status;
    buf_length = sizeof(sbuf.message) ;

    if ((msqid = msgget(key, msgflg )) < 0) {
        syslog(LOG_ERR, "Could not open message queue!\n");
        printf("Could not open message queue!\n");
        return;
    }

    return ;
}

void commit()
{
    return;
}

void inserthdr(uint32_t address, void *flowhdr)
{

    return ;
```

```
}

/* inserts flow into IPC queue */
void insertflow(uint32_t address, void *flowhdr, void *flow, int num)
{
    struct in_addr  in;
    char            str[128];
    int             version = ntohs(*((short *)flowhdr));
    struct tm       *stm;
    time_t          ttm;
    unsigned int    uptime;
    unsigned int    luptime;

    struct flowhdr5_t   *hdr5 = (struct flowhdr5_t *)flowhdr;
    struct flow1_t      *flow1 = (struct flow1_t *)flow;
    struct flow5_t      *flow5 = (struct flow5_t *)flow;
    struct flow7_t      *flow7 = (struct flow7_t *)flow;

    in.s_addr = address;
    if (!inet_ntop(AF_INET, &in, str, sizeof(str)))
        strncpy(str, "0.0.0.0", 7);

    if (ltime)
        ttm = time(NULL);
    else {
       uptime = htonl((unsigned int)hdr5->uptime);
       luptime = htonl((unsigned int)flow5->luptime);

        if (uptime > luptime) {
            ttm = htonl(hdr5->secs) - (uptime - luptime)/1000;
        } else {
            ttm = htonl(hdr5->secs) + (luptime - uptime)/1000;
        }
    }
    //stm = localtime(&ttm);

    switch (version) {
case 5: {            /* netflow v5 */
                memset(&sbuf, 0, sizeof(sbuf));
                sbuf.mtype = 5;

                sbuf.message.time = ttm;

                strncpy(sbuf.message.id, str,16);

                sprintf(sbuf.message.srcip, "%u.%u.%u.%u", flow5->srcaddr[0],
 flow5->srcaddr[1], flow5->srcaddr[2], flow5->srcaddr[3]);

                sbuf.message.sport = htons(flow5->srcport);

                sprintf(sbuf.message.dstip, "%u.%u.%u.%u", flow5->dstaddr[0],
 flow5->dstaddr[1], flow5->dstaddr[2], flow5->dstaddr[3]);

                sbuf.message.dport = htons(flow5->dstport);

                sbuf.message.prot = flow5->prot;

                sbuf.message.iif = htons(flow5->input_if);
```

```
                sbuf.message.oif =  htons(flow5->output_if);
                sbuf.message.srcas = htons(flow5->src_as);
                sbuf.message.dstas = htons(flow5->dst_as);
                sbuf.message.pkts = (unsigned long)htonl(flow5->dpkts);
                sbuf.message.octets = (unsigned long)htonl(flow5->doctets);

                if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
                    //syslog(LOG_ERR, "Could not send a message!\n");
                    printf("Could not send a message!\n");
                };
                break;
            }
      }
}

void insertpkt(uint32_t address, void *packet)
{
    struct flowhdr1_t   *hdr1 = (struct flowhdr1_t *)packet;
    short               version = ntohs(*((short *)packet));
    int                 i;
    void                *flow;

    if (!packet) return;

    inserthdr(address, packet);

    for (i = 0; i < htons(hdr1->count); i++) {
        switch (version) {
        case 1:
            flow = packet + sizeof(struct flowhdr1_t) +
                sizeof(struct flow1_t)*i;
            break;
        case 5:
            flow = packet + sizeof(struct flowhdr5_t) +
                sizeof(struct flow5_t)*i;
            break;
        case 7:
            flow = packet + sizeof(struct flowhdr7_t) +
                sizeof(struct flow7_t)*i;
            break;
        }
        insertflow(address, packet, flow, i);
    }
}
```

A-2 Código Fonte do Flowsender

```
/************************************************************************
 * Arquivo flowsender.cc                                                *
 ************************************************************************/

#include "args.h"
#include "util.h"
#include "FlowMarshal.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#include <unistd.h>
#include <string>

using namespace Borealis;

const Time  time0 = Time::now() - Time::msecs( 100 );

const uint32  SLEEP_TIME = 100;
const uint32  MAX_FLOWS = 4500;
#define USE_TIMESTAMP  1

////////////////////////////////////////////////////////////////////////////
//
// Return here after sending a packet and a delay.
//
void  FlowMarshal::sentFlow()
{
    int32        timestamp;
    Time         current_time;
    int32        cont;
    int32        num_msgs;

    //receive first message. This will block if there´s no message
    rcvMsgQ();

    Flow  first_tuple;

    timestamp = (int32)( Time::now() ).to_secs();
    if ( timestamp < 0 ) timestamp = 0;

    //fill the first tuple data with packet flow
    if (USE_TIMESTAMP == 1) {
first_tuple._data.time = _rbuf.message.time;
    } else {
        first_tuple._data.time = timestamp;
    }
    setStringField( _rbuf.message.id, first_tuple._data.id, 15 );
    setStringField( _rbuf.message.srcip, first_tuple._data.src, 15 );
    first_tuple._data.sport = _rbuf.message.sport;
    setStringField( _rbuf.message.dstip, first_tuple._data.dst, 15 );
    first_tuple._data.dport = _rbuf.message.dport;
    first_tuple._data.prot = _rbuf.message.prot;
    first_tuple._data.iif = _rbuf.message.iif;
    first_tuple._data.oif = _rbuf.message.oif;
    first_tuple._data.src_as = _rbuf.message.srcas;
    first_tuple._data.dst_as = _rbuf.message.dstas;
    first_tuple._data.pkts =  _rbuf.message.pkts;
    first_tuple._data.octs =  _rbuf.message.octets;
    batchFlow( &first_tuple );

    // ok, we already batched one tuple
    // let´s see what we still have in the queue
    cont = 1;
    num_msgs = numMsgQ();

    while ( cont <= num_msgs && cont < MAX_FLOWS ) {

        rcvMsgQ();
```

```
        timestamp = (int32)( Time::now() ).to_secs();
        if ( timestamp < 0 ) timestamp = 0;

        Flow  tuple;

        //fill tuple data with packet flow
        if (USE_TIMESTAMP == 1) {
    tuple._data.time = _rbuf.message.time;
        } else {
            tuple._data.time = timestamp;
        }
        setStringField( _rbuf.message.id, tuple._data.id, 15 );
        setStringField( _rbuf.message.srcip, tuple._data.src, 15 );
        tuple._data.sport = _rbuf.message.sport;
        setStringField( _rbuf.message.dstip, tuple._data.dst, 15 );
        tuple._data.dport = _rbuf.message.dport;
        tuple._data.prot = _rbuf.message.prot;
        tuple._data.iif = _rbuf.message.iif;
        tuple._data.oif = _rbuf.message.oif;
        tuple._data.src_as = _rbuf.message.srcas;
        tuple._data.dst_as = _rbuf.message.dstas;
        tuple._data.pkts =  _rbuf.message.pkts;
        tuple._data.octs =  _rbuf.message.octets;

        batchFlow( &tuple );
        cont++;
    }
    DEBUG << "Sending " << cont;
    sendFlow( SLEEP_TIME );
    return;
}




////////////////////////////////////////////////////////////////////////////
//
int  main( int  argc, const char  *argv[] )
{
    int32           status, status2;
    FlowMarshal     marshal;        // Client and I/O stream state.

    // default values
    string  ip   = "127.0.0.1";     // head host ip.
    int     port = 15000;
    int     key  = 1234;

//
// Maximum size of buffer with data awaiting transmission to Borealis
//.........................................................................

    int option_char;
    while ( (option_char = getopt (argc,(char**)argv,"i:p:k:h")) != EOF )
    {
        switch (option_char)
        {
        case 'i':
```

```
            ip = optarg;
            break;
        case 'p':
            port = atoi(optarg);
            break;
        case 'k':
            key = atoi(optarg);
            break;
        case 'h':
            cout << "Usage: flow [-i ip] [-p port] [-k key]" <<endl;
            return(1);
        }
    }

    // Run the front-end, open a client, subscribe to outputs and inputs.
    status = marshal.open(ip, port, key);

    if ( status )
    {
        WARN << "Could not deply the network.";
    }
    else
    {
        DEBUG  << "time0 = " << time0;

        // Send the first batch of tuples.  Queue up the next round with a delay.
        marshal.sentFlow();

        DEBUG  << "run the client event loop...";
        // Run the client event loop.  Return only on an exception.
        marshal.runClient();
    }

    return( status );
}


/*************************************************************************
 * Arquivo flowMarshal.h                                                 *
 *************************************************************************/


#ifndef  FLOWMARSHAL_H
#define  FLOWMARSHAL_H

#include  "MedusaClient.h"
#include  "TupleHeader.h"
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

////////////////////////////////////////////////////////////////////////////
//
// Generated Martialing code for the Flow program.
//........................................................................


using namespace Borealis;
```

```
class FlowMarshal
{
  public:

    FlowMarshal() {};
   ~FlowMarshal() {};

    /// Activate the front-end and subscribe to streams.
    /// Returns 0 if okay; else an error occured.
    ///
    int32  open(string ip, int port, int key);

    /// Run the client event loop.
    /// This does not return unless terminated or on an exception.
    ///
    void  runClient();


    /// Terminate the client event loop.
    ///
    void  terminateClient();


    /// Copy a string value to a fixed length array and zero fill.
    ///
    static void  setStringField(string  value,
                                char  field[],
                              uint32  length)
                         throw(AuroraException);


    /// Get the timestamp for a tuple.
    ///
    static timeval   getTimeValue(uint8  *tuple)
    {
       return(*((timeval *)(tuple - HEADER_SIZE)));
    }


  public:

    struct FlowTuple
    {
        int32    time;
        char    id[ 15 ];
        char    src[ 15 ];
        int32    sport;
        char    dst[ 15 ];
        int32    dport;
        int32    prot;
        int32    iif;
        int32    oif;
        int32    src_as;
        int32    dst_as;
        int64    pkts;
        int64    octs;
    } __attribute__((__packed__));
```

```
    struct  Flow : public TupleHeader
    {
        FlowTuple   _data;
    } __attribute__((__packed__));


    /// The sentFlow  method must be defined by the application.
    /// It is called after sendFlow is done and a pause.
    ///
    void  sentFlow();

    /// Enque a Flow for input.
    ///
    void  batchFlow(Flow  *tuple);

    /// Send enqued Flow inputs.
    ///
    void  sendFlow(uint32  sleep);

  private:

    /// Connect the Flow input stream.
    ///
    void  connectFlow(string ip, int port);

    ///  Resume here.  Extend with a user callback.
    void  delayFlow();

    /// open message queue, to receive packet flows
    int32  openMsgQ(int key);

    /// receive a packet flow (return 1 if thereis a msg, or 0 if thereis not)
    int32   rcvMsgQ();

    /// return the number of messages in queue
    int32   numMsgQ();

  private:

    /// Client connections to Borealis nodes.
    ///
    MedusaClient      *_client;


    /// Event state for input streams.
    /// These are declared with a smart pointer as fast_post_event requires it.
    ///
    ptr<StreamEvent>   _eventFlow;

    /// message queue stuff

    public:

    struct PktFlow {
        int  time;
char id[16];
char srcip[16];
```

```
        int  sport;
char dstip[16];
        int  dport;
        int  prot;
int  iif;
        int  oif;
        int  srcas;
        int  dstas;
        long pkts;
        long octets;
} ;

    struct flowbuffer {
         long     mtype;
         PktFlow  message;
         } ;

     int          _msqid;
     flowbuffer   _rbuf;


};

#endif                          // FLOWMARSHAL_H


/*************************************************************************
 * Arquivo flowMarshal.cc                                                *
 *************************************************************************/

#include  "FlowMarshal.h"
#include  "util.h"
#include <string>

#define FLOW_XML     "flow.xml"
#define MAX_BUFFER      400000000

#define FLOW  "flow"


////////////////////////////////////////////////////////////////////////////
//
// Generated marshaling code for the Flow program.
//.........................................................................


////////////////////////////////////////////////////////////////////////////
//
//  Subscribe to input and output streams.
//
int32  FlowMarshal::open(string ip, int port, int key)
{
    int32   status;

    // Open a client to send data.
    _client = new MedusaClient(InetAddress());

    connectFlow(ip, port);
```

```
    //also open msgQ

    status = openMsgQ(key);

    return(status);
}

int32  FlowMarshal::openMsgQ(int key)
{
    int32   status=0;

    if ((_msqid = msgget(key, 0666 | IPC_CREAT )) < 0) {
        ERROR << "msgget failed";
        exit(1);
    }

    return(status);
}

int32  FlowMarshal::numMsgQ()
{
    struct msqid_ds     bufinfo;

    if (msgctl(_msqid, IPC_STAT, &bufinfo) == -1) {
        ERROR << "msgctl failed";
        exit(1);
    }
    return( (int32) bufinfo.msg_qnum);

}

int32  FlowMarshal::rcvMsgQ()
{
    int32 size;

    size = sizeof(_rbuf);

    if (msgrcv(_msqid, &_rbuf, size, 5, 0 ) > 0) {
        return ( 1 );
    }
    return ( 0 );
}


void  FlowMarshal::runClient()
{
//.....................................................................


    //  This does not return unless terminated or on an exception.
    _client->run();

    return;
}
//////////////////////////////////////////////////////////////////////////////
//
//  Terminate the client I/O event loop.
```

```
void  FlowMarshal::terminateClient()
{
//.....................................................................


    _client->terminate();

    return;
}



////////////////////////////////////////////////////////////////////////////
//
//  Copy a string value to a fixed length array and zero fill.
//
void  FlowMarshal::setStringField( string  value,
                                     char  field[],
                                   uint32  length )
                              throw( AuroraException )
{

    if (value.length() > length)
    {   Throw(AuroraException,
              "Protocol string over " + to_string(length) + ".");
    }

    strncpy(field, value.c_str(), length);

    return;
}



////////////////////////////////////////////////////////////////////////////
//
void  FlowMarshal::connectFlow(string ip, int port)
{
//.....................................................................


    // Starting to produce events on input stream.
    //
    if (!_client->set_data_path(MAX_BUFFER, Util::get_host_address(ip.c_str()), port))
    {   ERROR << "Failed setting data path";
    }
    else
    {   DEBUG << "Set data path";

        _eventFlow = ptr<StreamEvent>(new StreamEvent(FLOW));
        _eventFlow->_inject = true;
    }

    return;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
void  FlowMarshal::batchFlow( Flow  *tuple )
{
//.....................................................................


    // Tuples are buffered in a string.
    //
    _eventFlow->insert_bin(string((const char *)tuple,
                                  sizeof(Flow)));

    return;
}



////////////////////////////////////////////////////////////////////////////
//
void  FlowMarshal::sendFlow( uint32  sleep )
{
//.....................................................................


    // Transmit data to the node.
    Status  status = _client->fast_post_event(_eventFlow);

    while (!status)
    {   if (status.as_string() == DataHandler::NO_SPACE)
        {   // Wait if no more space in buffer.
            // At this point the data was never put in the buffer.
            // It needs to be requeued unless we want to drop it.
            WARN << "We dropped a tuple.";
            Thread::sleep(Time::msecs(sleep));

            // retry (make this conditional).
            status = _client->fast_post_event(_eventFlow);
        }
        else
        {   ERROR << "Connection closed... stopping event stream";
            return;
        }
    }

    if (sleep)
    {    // The event loop is activated so that the queue can be processed.
         // The callback is enqueued with a timer.
         // We only callback with a timer because this is looping.
         // We also need a delayed callback so the queue can be processed.
         // If we just go to sleep the event loop will not be run.
         //
         (new CallbackTimer(_client->get_loop(),
                            wrap(this, &FlowMarshal::delayFlow)))
                 ->arm(Time::now() + Time::msecs(sleep));

    }

    return;
```

```
}


////////////////////////////////////////////////////////////////////////////
//
//  Resume here after sending a tuple.
//
void  FlowMarshal::delayFlow()
{
//.........................................................................


    // Release the previous event.
    //
    _eventFlow.reset();

    // Construct a new Flow input event.
    //
    _eventFlow = ptr<StreamEvent>(new StreamEvent(FLOW));
    _eventFlow->_inject = true;

    // Return to the application code.
    //
    sentFlow();

    return;
}

////////////////////////////  end  FlowMarshal.cc  ////////////////////////
```

A-3 Código Fonte do BigMouth

```
/**************************************************************************
 * Arquivo BigMouth.cc                                                    *
 **************************************************************************/

#include <unistd.h>
#include <iostream>
#include <string>
#include <fstream>
#include  "util.h"
#include "HeadClient.h"


using namespace Borealis;

int main(int argc, const char *argv[]) {

    string  ip   = "127.0.0.1";     // head host ip.
    int     port = 35000;         // head port.
    string  file_name = "";
    string  xmlthing="";
    string  buf;
```

```
    std::ifstream datafile;

    HeadClient  *client;
    RPC<void>  rpc;

    int option_char;
    while ( (option_char = getopt (argc,(char**)argv,"i:p:f:h")) != EOF )
    {
        switch (option_char)
        {
        case 'i':
            ip = optarg;
            break;
        case 'p':
            port = atoi(optarg);
            break;
        case 'f':
            file_name = optarg;
            break;
        case 'h':
            cout << "Usage: BigMouth [-i ip] [-p port] -f <filaname>" <<endl;
            exit;
            break;
        }
    }

    if (file_name == "") {
        cout << "No file to open" <<endl;
        exit ;
    }

    datafile.open(file_name.c_str());
    if ( ! datafile) {
        cout << "Can´t open the file" << endl;
        exit(1);
    }
    getline(datafile, buf);
    while (datafile) {
        xmlthing += buf;
        getline(datafile, buf);
    }

    client = (HeadClient *)new HeadClient( InetAddress( ip, port ));

    rpc = client->deploy_xml_string( xmlthing );

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

}
```

A-4 Código Fonte do ureceiver

```
/*****************************************************************************
 * Arquivo ureceiver.c c                                                     *
 *****************************************************************************/
#include <unistd.h>
#include <string>
#include "args.h"
#include "receiverMarshal.h"

using namespace Borealis;

const Time  time0 = Time::now() - Time::msecs( 100 );

map<string, CatalogSchema> MyDiagram::_output_schema;

int  main( int  argc, const char  *argv[] )
{
    int32           status;
    receiverMarshal marshal;        // Client and I/O stream state.
    string          xml_file = "";
    int     port = 2222;

    int option_char;
    while ( (option_char = getopt (argc,(char**)argv,"p:f:h")) != EOF )
    {
        switch (option_char)
        {
        case 'p':
            port = atoi(optarg);
            break;
        case 'f':
            xml_file = optarg;
            break;
        case 'h':
            cout << "Usage: ureceiver [-p port] [-f xml_file]" <<endl;
            return(1);
        }
    }
    // Run the front-end, open a client, subscribe to outputs and inputs.
    status = marshal.open(port);

   if (!marshal.parse_file(xml_file) || !marshal.infer_schema())
    {
                cout << "Could not parse the xml_file" <<endl;
        return -2;
    }

    marshal.runClient();

    marshal.terminateClient();

    return( status );
}


/*****************************************************************************
 * Arquivo MyDiagram.h                                                       *
```

```
 ***************************************************************************/
#ifndef  PAQUETDIAGRAM_H
#define  PAQUETDIAGRAM_H

#include "Diagram.h"

BOREALIS_NAMESPACE_BEGIN

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

public:
    static map<string, CatalogSchema> _output_schema;

};

BOREALIS_NAMESPACE_END
#endif


/***************************************************************************
 * Arquivo MyDiagram.cc                                                    *
 ***************************************************************************/

#include  "MyDiagram.h"
#include  "parseutil.h"

BOREALIS_NAMESPACE_BEGIN

//  Determine the schemas for pending intermediate streams.
//  Check for cyclic deployment.
Status  MyDiagram::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();
                   }
               }
               // Remove the box from the to-do list;  Rescan boxes.
               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


/*****************************************************************************
 * Arquivo receiverMarshal.h                                                 *
 *****************************************************************************/
#ifndef  RECEIVERMARSHAL_H
#define  RECEIVERMARSHAL_H

#include  "MyDiagram.h"
#include  "MedusaClient.h"
#include  "TupleHeader.h"


using namespace Borealis;

class receiverMarshal : public MyDiagram
{
  public:

    receiverMarshal() {};
   ~receiverMarshal() {};

    int32  open(int port);
    void  runClient();
    void  terminateClient();

    /// Copy a string value to a fixed length array and zero fill.
    static void  setStringField(string  value,
                                char  field[],
                              uint32  length)
                         throw(AuroraException);

    /// Get the timestamp for a tuple.
    static timeval   getTimeValue(uint8  *tuple)
    {
       return(*((timeval *)(tuple - HEADER_SIZE)));
    }


  public:
```

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

    struct  OutputTuple
    {
     string str;
    };

  private:

    /// Handler to dispatch tuples received.
    static Status outputHandler(ptr<StreamEvent> event);

  public:

    static void  receivedOutput(OutputTuple *tuple, string output);

  private:

    /// Subscribe to the output stream.
    void  subscribeOutput(int port);

  private:

    /// Client connections to Borealis nodes.
    MedusaClient      *_client;
    ptr<StreamEvent>   _eventPacket;
};

#endif                          // MYTESTMARSHAL_H


/**************************************************************************
 * Arquivo receiverMarshal.cc                                             *
 **************************************************************************/
#include  "receiverMarshal.h"
#include  "util.h"

#define MY_ENDPOINT     "0.0.0.0"

//  Subscribe to output streams.
int32  receiverMarshal::open(int port)
{
    int32   status;

    // Open a client to send and receive data.
    _client = new MedusaClient(InetAddress());

    // Subscribe to outputs.
    subscribeOutput(port);

    return(0);
}

//  Activate the client I/O event loop.
```

```
void  receiverMarshal::runClient()
{
 //  This does not return unless terminated or on an exception.
    _client->run();

    return;
}

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

    return;
}

//  Copy a string value to a fixed length array and zero fill.
void  receiverMarshal::setStringField( string  value,
                                       char  field[],
                                     uint32  length )
                                throw( AuroraException )
{
    if (value.length() > length)
    {   Throw(AuroraException,
              "Protocol string over " + to_string(length) + ".");
    }

    strncpy(field, value.c_str(), length);

    return;
}

// Dispatch output on our fast datapath to a handler.
Status  receiverMarshal::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 (index = 0; index < event->_inserted_count; index++)
        {
            offset += HEADER_SIZE;
    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();
        }
   }

    return(true);
}

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  receiverMarshal::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());
}
}

// Subscribing to receive output on a fast datapath.
void  receiverMarshal::subscribeOutput(int port)
{
//.....................................................................

    DEBUG << "Subscribing to receive output.";

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

    if (status)
    {   DEBUG << "Done subscribing to output.";
    }
    else
    {   ERROR << "Could not subscribe to output.";
    }

    return;
}
```

A-5 Código Fonte do dummysender

```
/*********************************************************************
 * Arquivo dummysender.c                                             *
 *********************************************************************/
```

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <syslog.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct  {
        int  time;
char id[16];
char srcip[16];
        int  sport;
char dstip[16];
        int  dport;
        int  prot;
int  iif;
        int  oif;
        int  srcas;
        int  dstas;
        long pkts;
        long octets;
} message_fmt;

typedef struct  {
         long    mtype;
         message_fmt message;
         } message_buf;

int send_default(key_t key, int i)
{

    int     msqid;
    int     msgflg = IPC_CREAT | 0666;
    int     count=1;
    message_buf sbuf;
    size_t buf_length;
    buf_length = sizeof(sbuf.message);

    if ((msqid = msgget(key, msgflg )) < 0) {
        printf("Could not open message queue!\n");
        exit;
    }

    sbuf.mtype = 5;
    sbuf.message.time = (int) time(NULL);
    sprintf(sbuf.message.id, "1.2.3.4");
    sprintf(sbuf.message.srcip, "10.0.0.1");
    sbuf.message.sport = 1000;
```

```
    sprintf(sbuf.message.dstip, "20.0.0.1");
    sbuf.message.dport = 20;
    sbuf.message.prot = 6;
    sbuf.message.iif = 10;
    sbuf.message.oif =  20;
    sbuf.message.srcas = 1000;
    sbuf.message.dstas = 2000;
    sbuf.message.pkts = 1;
    sbuf.message.octets = 100;

    for(count=1; count<=i; count++){

       if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
           printf("Could not send a message!\n");
       }
    }
    return count;
}

int send_random(key_t key, int i)
{
    int     msqid;
    int     msgflg = IPC_CREAT | 0666;
    int     count=1;
    message_buf sbuf;
    size_t buf_length;
    buf_length = sizeof(sbuf.message);

    if ((msqid = msgget(key, msgflg )) < 0) {
        printf("Could not open message queue!\n");
        exit;
    }

    sbuf.mtype = 5;
    sbuf.message.time = (int) time(NULL);

    srand (time (0));

    for(count=1; count<=i; count++)
    {
        sprintf(sbuf.message.id, "10.0.0.%d", rand() % 5);
        sprintf(sbuf.message.srcip, "%d.%d.%d.%d", rand() % 255,
rand() % 255, rand() % 255, rand() %255);
        sbuf.message.sport = rand() % 65535;
        sprintf(sbuf.message.dstip, "%d.%d.%d.%d", rand() % 255,
rand() % 255, rand() % 255, rand() %255);
        sbuf.message.dport = rand() % 65535;
        sbuf.message.prot = 6;
        sbuf.message.iif = rand() % 3;
        sbuf.message.oif = rand() % 3;
        sbuf.message.srcas = rand() % 100;
        sbuf.message.dstas = rand() % 100;
        sbuf.message.pkts = rand() % 1000;
        sbuf.message.octets = (sbuf.message.pkts * (rand() % 1400)) + 64;

        if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
           printf("Could not send a message!\n");
           exit;
```

```
       }
    }
    return count;
}

int send_file(key_t key, char* dfile, int i)
{
    int     msqid;
    int     msgflg = IPC_CREAT | 0666;
    int     count=1;
    int     sent=0;
    message_buf sbuf;
    size_t buf_length;
    buf_length = sizeof(sbuf.message);
    FILE    *fp;

    if ((msqid = msgget(key, msgflg )) < 0) {
        printf("Could not open message queue!\n");
        exit;
    }
    sbuf.mtype = 5;
    //sbuf.message.time = (int) time(NULL);
    for(count=1; count<=i; count++)
    {
        if((fp=fopen(dfile, "r"))==NULL)
        {
            printf("Cannot open file.\n");
            exit(1);
        }
        while ( fscanf(fp, "%li %s %s %d %s %d %d %d %d %d %d %li %li",
 &sbuf.message.time, &sbuf.message.id, &sbuf.message.srcip, &sbuf.message.sport,
 &sbuf.message.dstip, &sbuf.message.dport, &sbuf.message.prot, &sbuf.message.iif,
 &sbuf.message.oif, &sbuf.message.srcas, &sbuf.message.dstas, &sbuf.message.pkts,
 &sbuf.message.octets) != EOF)
        {

            if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
                printf("read from file: Could not send a message!\n");
                exit;
            }
            sent += 1 ;
            if ( sent % 4000 == 0 )
            {
                printf("sleeping 2s\n");
                sleep(2);
            }
        }
        fclose(fp);
    }
}
void main( int  argc, const char  *argv[] )
{
    int     repeat=1;
    char*    mode="d";
    key_t   key=1234;
    char*    file=NULL;
    int option_char;
    while ( (option_char = getopt (argc,(char**)argv,"m:f:r:k:h")) != EOF )
```

```
    {
        switch (option_char)
        {
        case 'r':
            repeat = atoi(optarg);
            break;
        case 'k':
            key = atoi(optarg);
            break;
        case 'f':
            file = optarg;
            break;
        case 'm':
            mode = optarg;
            break;
        case 'h':
             printf("Usage: dummysender [-m mode [-f file] ] [-r repeat] [-k key] [-h]\n");
             printf("\tmode can be: [d]efault, [r]andom or [f]ile\n");
            exit;
        }
    }

   //printf("mode = %s, file = %s, repeat = %d, key = %d\n", mode, file, repeat, key);

   switch ( *mode )
   {
        case 'd':  send_default(key, repeat);
                   break;
        case 'r':  send_random(key, repeat);
                   break;
        case 'f':  send_file(key, file, repeat);
                   break;
        default:   printf("unknow mode\n");
                   break;
   }
}
```