UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL PIMENTEL AFFONSO DE OLIVEIRA

Monitoramento e análise visual de tráfego de dados de aplicações baseadas em protocolos epidêmicos

Trabalho de Conclusão de Curso

Prof. Dr. Taisy Silva Weber
Orientadora

Porto Alegre, dezembro de 2012.

Agradecimentos

Primeiramente à professora Taisy Weber, pelo interesse e apoio que demostrou em ser minha orientadora, desde o momento de definição de escopo e tema principal até as revisões finais do texto escrito.

Aos meus pais, avó e irmã pela paciência e apoio diários, tanto no decorrer deste trabalho quanto durante o curso inteiro.

Aos meus amigos, de aula e de empresa, pelo interesse com que me ouviram divagar e pelas suas críticas e sugestões.

À Fernanda, por entender todo meu palavreado técnico e pelas divagações compartilhadas sobre a vida e tudo o mais.

SUMÁRIO

LISTA DE FIGURAS 5

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

1.1 Objetivo 12

1.2 Estrutura do trabalho 12

2 ALGORITMOS EPIDÊMICOS 13

2.1 Motivação 13

2.2 Características Fundamentais 14
2.2.1 Empurrar ou puxar rumores? 15

2.3 Protocolos para disseminação confiável 16
2.3.1 Pbcast: um protocolo de multicast bimodal 17
2.3.2 Lpbcast: um protocolo de multicast probabilístico econômico 19
2.3.3 NEeM: Network Friendly Epidemic Multicast 20
2.3.4 Detalhes de Implementação do NEeM 0.7 21
2.3.4.1 Classe Transport 22
2.3.4.2 Classe Overlay 22
2.3.4.3 Classe Gossip 23

3 MONITORAMENTO DE TRÁFEGO DE REDE 26

3.1 Motivação 26

3.2 Monitoramento ativo 27

3.3 Monitoramento passivo 27

3.4 JMX: Instrumentação da Máquina Virtual Java 28
3.4.1 Visão geral 28
3.4.2 Camada de Instrumentação 29
3.4.3 Camada de Agente 30

3.4.4 Camada de Serviços Distribuídos 30

3.5 Uso do JMX no NEeM: 31

4 PROJETO LUSIR (*LIVE USER INTERACTIVE REMOTE MONITOR*). 33

4.1 Módulo de Monitoramento 34

4.2 Módulo de Análise 35

5 PROTÓTIPO LUSIR 36

5.1 Modo de uso do protótipo 36
5.1.1 Aba de monitoramento 37
5.1.2 Aba de análise 39

5.2 Descrição das classes do protótipo 46
5.2.1 Classes de Layout 47
5.2.2 Classe de Eventos do Usuário 47
5.2.3 Classes de Interface JMX 48
5.2.4 Classes Utilitárias 50

6 SIMULAÇÕES E RESULTADOS PRÁTICOS 52

6.1 Testes durante o Desenvolvimento 52
6.1.1 Visão geral 52
6.1.2 Análise dos dados mais primitivos 55
6.1.3 Análise semanticamente mais rica 61
6.1.4 Ponderações finais sobre a análise da simulação com o programa Crowd 67

6.2 Testes com programa demonstrativo de Chat do NEeM 67
6.2.1 Visão geral 68
6.2.2 Análise dos dados mais primitivos 71
6.2.3 Análise semanticamente mais rica 74
6.2.4 Ponderações finais sobre a análise da simulação com o programa Chat 77

7 CONCLUSÃO 78

LISTA DE FIGURAS

Figura 3.1: Relação entre componentes das camadas da arquitetura JMX 29

Figura 4.1: Fluxo de dados monitorados pelo protótipo LUsIR 34

Figura 5.1: Interface básica da aba “Live Monitoring” no protótipo LUsIR 37

Figura 5.2: Início de uma sessão de monitoramento no protótipo LUsIR 38

Figura 5.3: Final de uma sessão de monitoramento no protótipo LUsIR 39

Figura 5.4: Interface sem dados importados na aba Console do protótipo LUsIR 40

Figura 5.5: Execução da consulta de seleção padrão na aba Console do protótipo LUsIR 41

Figura 5.6: Execução de uma consulta de amostra na aba Console do protótipo LUsIR 42

Figura 5.7: Plotando dados de uma consulta de amostra na aba Console do protótipo LUsIR 43

Figura 5.8: Execução de duas consultas de amostra na aba Console do protótipo LUsIR 44

Figura 5.9: Plotando dados de duas consultas de amostra na aba Console do protótipo LUsIR 44

Figura 5.10: Execução de parte do conteúdo do editor de consultas na aba Console do protótipo LUsIR 45

Figura 5.11: Salvando o conteúdo do editor de consultas na aba Console do protótipo LUsIR 46

Figura 5.12: Relação entre classes do protótipo LUsIR 46

Figura 5.13: Ciclo de funcionamento durante uma sessão de monitoramento 48

Figura 5.14: Ciclo de funcionamento de uma instância da classe Sensor 49

Figura 6.1: Gráficos por atributo monitorado durante experimento de Crowd 54

Figura 6.2: Exemplo de consulta utilizada para gerar um dos gráficos analisados após experimento de Crowd 55

Figura 6.3: Gráfico demonstrando o atributo “Bytes Recebidos por nodo” ao final de experimento de Crowd 56

Figura 6.4: Gráfico demonstrando o primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” ao final de experimento de Crowd 57

Figura 6.5: Consulta para filtrar os dados do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” 58

Figura 6.6: Gráfico demonstrando o primeiro minuto de amostras do atributo “Bytes Enviados por nodo” ao final de experimento de Crowd 59

Figura 6.7: Consulta para filtrar os dados do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” 59

Figura 6.8: Gráfico demonstrando uma associação das amostras do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” com “Bytes Enviados por nodo” 60

Figura 6.9: Gráfico demonstrando uma associação das amostras dos primeiros cinco minutos de amostras do atributo “Bytes Recebidos por nodo” com “Bytes Enviados por nodo” 61

Figura 6.10: Gráfico demonstrando uma associação das amostras dos primeiros cinco minutos de amostras do atributo “Shuffles Enviados por nodo” com “Bytes Enviados por nodo” 62

Figura 6.11: Consulta para evidenciar os picos e vales da série de amostras do atributo “Shuffles Enviados por nodo” 63

Figura 6.12: Consulta para evidenciar os picos e vales da série de amostras do atributo e aumentar seu offset “Shuffles Enviados por nodo” 64

Figura 6.13: Gráfico comparativo dos Bytes e Shuffles enviados num período de 3 minutos de experimento 65

Figura 6.14: Dados contendo os Shuffles enviados pelo nodo 4444 num período de 3 minutos de experimento 66

Figura 6.15: Gráfico comparativo dos Shuffles enviados e recebidos pelo nodo 4444 num período de 3 minutos de experimento 67

Figura 6.16: Exemplo de identificador de um nodo, conforme ele é fornecido ao usuário 68

Figura 6.17: Exemplo de mensagens, conforme elas são fornecidas ao usuário 68

Figura 6.18: Conversação monitorada, conforme vista pelo nodo 53182 (topo, a esquerda), 53183 (topo, a direita) e 53197 (centro) 69

Figura 6.19: Gráficos por atributo monitorado durante experimento de Chat 70

Figura 6.20: Gráfico demostrando as mensagens enviadas pelo nodo 53182 e recebidas pelos outros nodos durante experimento de Chat 71

Figura 6.21: Gráfico demostrando as mensagens recebidas pelo nodo 53187 durante experimento de Chat 72

Figura 6.22: Gráfico demostrando as mensagens recebidas pelo nodo 53187 e a relação delas com eventos de Deliver e Multicast, durante experimento de Chat 73

Figura 6.23: Gráfico demostrando as mensagens recebidas pelo nodo 53187, já excluindo aquelas que foram delivered ao usuário ou multicasted, durante experimento de Chat 74

Figura 6.24: Consulta demostrando os momentos nos quais as mensagens foram entregues ao usuário do nodo 53182 durante experimento de Chat 75

Figura 6.25: Exemplo de como a ordem dos eventos e a ordem de obtenção de amostras pode variar 76

Figura 6.26: Situação em que o momento de tempo no qual as amostras foram obtidas reflete a ordem dos eventos na aplicação 77

Figura 6.27: Situação em que o momento de tempo no qual as amostras foram obtidas não reflete a ordem dos eventos na aplicação 78

LISTA DE TABELAS

Tabela 5.1: Lista de atributos monitorados pelo protótipo LUsIR 37

Tabela 6.1: Comportamento de cada atributo monitorado durante experimento de Crowd 53

Tabela 6.2: Comportamento de cada atributo monitorado durante experimento de Crowd 70

RESUMO

Entender o funcionamento de um conjunto de nodos num sistema distribuído não é uma tarefa trivial. Dentre estes sistemas, aqueles que utilizam protocolos epidêmicos são particularmente desafiadores por causa da dinamicidade das conexões entre estes nodos. A maioria das técnicas que auxiliam nessa tarefa se apoia em uma enorme quantidade de dados acumulados que descrevem certas ações tomadas por esses nodos e tentam extrair padrões e explicá-los. Vários algoritmos sofisticados foram criados para evidenciar detalhes desses dados que um ser humano não conseguiria detectar.

Porém, existem padrões que estes algoritmos não conseguem encontrar, em especial comportamentos inesperados do sistema. Pouco se tem explorado sobre como a percepção humana pode ajudar neste processo. O foco deste trabalho, portanto, é criar um sistema que permita coletar e exibir dados obtidos de uma aplicação distribuída, sendo executada em diversos nodos de uma rede, e possibilitar a um operador humano explorar esses dados a fim de reconhecer comportamentos esperados ou inesperados.

Palavras-Chave: Protocolos Epidêmicos, Monitoramento de Tráfego em Redes, Visualização de Dados, Protocolos Epidêmicos.

Analysis and monitoring of traffic from epidemic based applications

ABSTRACT

Understand the behavior of a set of nodes in a distributed system is not a trivial task. Among these systems, those based on epidemic protocols are particularly challenging because of the quantity of dynamic connections between nodes. Most known techniques try to extract patterns (and explain them) from a huge quantity of data, that describe some important tasks made by these nodes. Many sophisticated algorithms were created to detect details on this huge amount of data, while a human being is not able to do that.

However, there are some pattern that those algorithms can`t find, especially when the system shows some unexpected behavior. Therefore, this work focus on creating a system that can allow the extraction and analysis of data obtained from a distributed application, executed in many nodes in the same network, in order to help a human operator to analyze and explore this data and recognize expected and unexpected behaviors.

Keywords: Epidemic Protocols, Network Traffic Monitoring, Data Visualization, Gossip Protocols.

1 INTRODUÇÃO

Quando programas de computador começaram a ser criados, a única forma de uma pessoa, sem acesso ao código fonte, entender seu comportamento era através da leitura de arquivos de logs. Através destes arquivos, ela podia analisar o que a aplicação tinha feito em momentos passados. Obviamente que essa funcionalidade dependia do programa organizar uma política na qual a aplicação periodicamente salva informações sobre seu funcionamento em alguma forma de meio persistente, geralmente um arquivo de texto em disco.

Conforme os programas realizavam operações mais complexas a quantidade de informação que devia existir nesses arquivos de log, para que um operador compreendesse o que de fato estava acontecendo com a aplicação, aumentava gradativamente. Pior era o caso de programas que precisavam se comunicar com outros programas que lhe prestam serviços, por exemplo, de persistência de dados. Não raramente, no arquivo de log se misturam tanto as informações sobre o funcionamento da aplicação quanto com aquelas para a comunicação com outros programas. Na medida em que o programador detectava que certa informação que não constava nos arquivos de texto era necessária, ele era requisitado a inseri-la, sem a preocupação de obedecer a nenhuma forma de padronização de formato ou conteúdo.

Com o aumento da quantidade de dados que precisavam ser policiados ao mesmo tempo, procuraram-se novas formas de representação que pudessem transmitir a uma pessoa uma quantidade maior de informação de maneira mais eficiente. Gráficos e séries pareceram uma forma adequada de agrupar e visualizar estes dados. Porém, uma grande quantidade de informação textual ainda vai gerar uma massa grande de dados em formato gráfico e esta precisa ser explorada, filtrada e analisada com o intuito de revelar alguma informação útil.

Hoje, a ausência de padrões em ferramentas de obtenção de dados torna difícil a transformação para formas gráficas de representação. Os próprios programas de representação gráfica costumam divergir quanto ao formato que deve ser usado pelos dados antes deles serem convertidos para forma gráfica e quanto às funcionalidades que disponibilizam ao usuário para refinar esses dados e facilitar sua representação.

Cada nova peça de software que é criada representa um novo desafio para um analista, que se vê diante do problema de gerenciar dados em formatos estranhos a fim de representá-los de uma forma que se consiga extrair o comportamento atual do sistema.

1.1 Objetivo

Neste contexto, este trabalho usa um conjunto de tecnologias que, combinadas, tem por objetivo desempenhar as funções de:

- Aquisição e formatação de dados num formato comum, que possa ser usado por outras ferramentas de criação de representações gráficas dos dados;
- Exploração e refinamento destes dados usando uma linguagem estruturada já usada e conhecida na área de banco de dados;

Para evidenciar a necessidade de representações gráficas de dados, em especial quando se precisa entender o funcionamento de programas que se comunicam com outros programas, foi usado como exemplo uma aplicação de sistemas distribuídos que faz uso de protocolos epidêmicos. Neste tipo de protocolo, cada programa se comunica com uma parcela de todos os outros programas (EUGSTER, 2004) e, com isso, precisa disseminar uma informação de maneira confiável a todos eles.

1.2 Estrutura do trabalho

O capítulo 2 apresentará o ferramental teórico para entender o funcionamento de protocolos epidêmicos e a utilidade destes em sistemas distribuídos. Ainda, neste capítulo será apresentada a biblioteca de software livre NEeM (PEREIRA, 2003) que provê ao programador uma API para criação de aplicações baseadas em protocolos epidêmicos.

No capítulo 3 serão descritas e discutidas estratégias para se monitorar programas complexos, em especial sistemas distribuídos. Também será descrita a forma como a máquina virtual Java providencia uma API, para o programador explicitar quais variáveis do seu programa devem ser monitoradas e como um usuário externo requisita essas informações.

O capítulo 4 descreverá a especificação e definirá as necessidades de um software que desenvolva, de maneira integrada, as tarefas de análise e monitoramento do protocolo NEeM.

O capítulo 5 descreverá o protótipo desenvolvido para exibir e refinar os dados obtidos, além de um caso de uso completo do sistema e uma descrição exata das classes envolvidas na sua criação, bem como as relações entre elas.

Por fim, o capítulo 6 avaliará o sistema criado baseado em dois programas demonstrativos criados e distribuídos pelos mesmos criadores do NEeM. Além disso, neste capítulo será feita a análise dos dados monitorados, usando o protótipo desenvolvido, a fim de entender se o que foi monitorado explica o funcionamento do NEeM descrito no capítulo 4.

O capítulo 7 ponderará sobre o que foi realizado e definirá possíveis trabalhos futuros que poderiam complementar o protótipo.

Detalhes sobre a instalação do sistema e obtenção dos códigos-fonte poderão ser vistos no Apêndice.

2 ALGORITMOS EPIDÊMICOS

Neste capítulo serão abordados os conceitos nos quais os protocolos epidêmicos estudados neste trabalho se baseiam. Será apresentada uma visão do progresso histórico que o estudo nessa área teve, até o surgimento do NEeM, protocolo utilizado nas experiências práticas neste trabalho.

2.1 Motivação

Conforme as redes de computadores aumentam de tamanho, as empresas precisam cada vez mais tornar e manter suas informações disponíveis a um maior número de clientes e por um maior intervalo de tempo. Aplicações com um servidor de conteúdo único acabam limitadas pela capacidade de prover dados desta única máquina. Ainda, se essa máquina parar de funcionar, toda a aplicação é prejudicada.

Sistemas provedores de serviços a partir de uma *cloud* surgiram para que não mais um servidor fosse responsável pelo funcionamento de aplicações, mas sim um conjunto deles. Ao cliente, porém, tem de ser proporcionada a certeza de que, independente de qual servidor da "nuvem" atenda às suas requisições, a aplicação é a mesma e age sobre o mesmo conjunto de dados. Existe, portanto, a necessidade de replicar e manter sincronizadas as informações da aplicação em todos estes servidores.

O desafio de manter a consistência de dados entre vários servidores foi estudado por Demers (1987) que, na época da Xerox, precisava melhorar essa funcionalidade do sistema Clearinghouse, um serviço da Xerox de *Domain Name Server* (DNS) distribuído que mapeava nomes para endereços na Internet.

Uma implementação disponível na época se baseava em mensagens diretas: cada atualização numa cópia do banco de dados forçava aquele servidor a mandar uma mensagem para todos os outros. Essa alternativa tinha algumas características interessantes, como:

1. Eficiência, já que gerava apenas o número necessário de mensagens;
2. Falta de confiabilidade, pois as mensagens poderiam ser perdidas;
3. Cada servidor precisava ter total conhecimento da rede, para poder enviar uma mensagem a todos outros servidores.

Uma das formas de resolver os problemas 2. e 3. listados acima é com o uso de protocolos epidêmicos, cujo funcionamento pode ser explicado pela analogia em Demers (1987):

> Temos vários indivíduos, inicialmente inativos (suscetíveis). Um rumor é plantado quando uma pessoa se torna ativa (infectada), telefona para outra pessoa randômica e compartilha esse rumor. Cada pessoa ouvindo esse rumor

> também se torna ativa e, da mesma forma, compartilha esse rumor com outras pessoas.

Por essa analogia, cada servidor não precisaria conhecer todos os elementos da rede para disseminar corretamente um rumor (o que resolve o problema 3.). Ainda, servidores em estados de falha (inativos) não atrapalham a comunicação na rede e a disseminação da informação para outros servidores ativos (o que resolve o problema 2.).

Porém, segundo Karp (2000), enquanto um algoritmo por mensagens diretas manda n-1 mensagens em 1 rodada, um algoritmo epidêmico mandaria $\Omega(n \ln \ln n)$ mensagens em $O(\ln n)$ rodadas. Portanto, o preço que se paga pela confiabilidade, robustez e simplicidade de algoritmos epidêmicos acaba sendo a eficiência em relação a quantidade de mensagens enviadas.

2.2 Características Fundamentais

Da analogia descrita anteriormente obtemos dois estados (dos três possíveis) de "indivíduos" (servidores ou, simplesmente, nodos numa rede), relativos ao recebimento de uma mensagem (ou "rumor", para acompanhar a analogia da seção anterior) qualquer, descritos em Demers (1987), como segue:

- **Suscetível**, se ainda não recebeu a mensagem;
- **Infectado**, se recebeu a mensagem e está disseminando-a para outros nodos;
- **Removido**, se recebeu a mensagem, mas não está mais disseminando essa informação;

Várias alternativas são discutidas para decidir quando um nodo deve passar do estado Infectado para Removido. Em Eugster (2004) essa escolha depende de um parâmetro *t*, que regula a quantidade de mensagens que um nodo dissemina antes de se tornar inativo. A quantidade de nodos que são escolhidos randomicamente para serem infectados também é regida por um parâmetro, chamado de *fanout*. Um terceiro parâmetro básico, ainda segundo Eugster, seria a quantidade de rumores que o nodo "conhece", ou seja, quantas mensagens ele guardaria em buffer para disseminá-las pela rede.

Esses parâmetros foram apresentados primeiramente no desenvolvimento do protocolo *pbcast* de Birman (1999). A solução descrita por ele faz com que as mensagens mais antigas sejam descartadas e não mais disseminadas na rede. Dessa forma, nodos com falhas transientes, quando voltarem a operar normalmente, não serão inundados com mensagens velhas e desinteressantes.

Outra discussão tenta definir quem deve ter o papel de disseminar a informação. A analogia sobre o funcionamento de protocolos epidêmicos que foi usada anteriormente diz que esse anúncio (e posterior disseminação) deve ser feito pelo nodo que detém a nova informação. Portanto, cabe ao nodo inicial *empurrar* pela rede a nova informação a fim de infectar vizinhos suscetíveis. Porém, existe a possibilidade de os nodos que querem saber sobre novidades ativamente irem procurar por ela e *puxarem* essa informação dos nodos que as possuem. Essas duas variações foram descritas por Demers (1987) e avaliadas matematicamente por Karp (2000).

2.2.1 Empurrar ou puxar rumores

Em Demers (1987) é descrito o algoritmo de anti-entropia que tem como objetivo sincronizar os bancos de dados entre vários nodos do sistema Clearinghouse. Ele era executado periodicamente nos servidores da aplicação, a fim de compensar as perdas de mensagens no algoritmo de mensagens diretas. A cada execução, cada servidor escolhia outro, de maneira randômica, a fim de resolver as diferenças que havia entre seus bancos de dados. Três modos de funcionamento existiam referentes à forma como a informação atualizada era transmitida:

- Aquele nodo com a nova informação empurrava-a para o outro nodo;
- O nodo sem a nova informação puxava-a a partir do outro nodo;
- Uma forma mista de funcionamento.

Ainda, Demers descreve que a variação que puxava (*pull*) a informação convergia mais rapidamente, ou seja, fazia com que a informação nova se tornasse conhecida por todos os nodos, do que a variação que empurrava essa informação (*push*). Karp (2000) descreve que a primeira variação precisava, considerando uma quantidade n de nodos, de apenas $\Theta(n \ln \ln n)$ mensagens para que todos os nodos conhecessem um novo rumor, em contraste com as $\Theta(n \ln n)$ necessárias para que a segunda variação alcançasse o mesmo resultado. Contudo, essa afirmação apenas pode ser feita se, na variação que puxa a informação, existir um meio de parar a disseminação na rede.

Karp (2000) ainda descreve as diferenças na propagação de um rumor de cada uma destas duas variações, analisando cada rodada de envio de mensagens. Em uma rodada, cada nodo escolhe outro, randômica e uniformemente, para se conectar.

Segundo ele, a variação de empurrar funciona de uma maneira mais previsível, visto que “o conjunto de nodos informados cresce exponencialmente até que metade dos nodos foi informada [do novo rumor]" e continua "uma vez que metade dos nodos esteja desinformada da informação, esse conjunto diminui num fator constante a cada rodada" (KARP, 2000).

Já a variação de puxar precisa de, no mínimo, $O(\ln n)$ rodadas para infectar essa mesma metade de nodos. Karp ainda fala que “deste ponto em diante, o algoritmo de puxar tem uma vantagem em relação ao algoritmo de empurrar já que a fração de nodos desinformados bruscamente dobra de uma rodada para a próxima” (KARP, 2000).

A fim de combinar essa previsibilidade do algoritmo de empurrar com o crescimento rápido do algoritmo de puxar, Karp desenvolve um algoritmo híbrido puxa-empurra que se baseia num mecanismo para definir a condição de parada da disseminação da informação. Este mecanismo usa dois contadores: um por mensagem (transmitido juntamente com o rumor a ser disseminado) e outro de estado (local a cada servidor). Cada nodo pode assumir quatro estados, como segue:

- **Estado A:** similar ao estado suscetível de Demers. Neste estado, o contador global possui valor zero (0);
- **Estado B-m:** sendo **m** o valor do contador global daquele nodo. Neste estado o servidor difunde mensagens com o algoritmo de empurrar. Junto à mensagem deve ir o valor atual do contador global. O nodo neste estado transita para o estado C em um dos casos abaixo:
    - Quando recebe uma mensagem de um nodo no estado C;
    - No momento em que o contador global **m** atinge um valor de $O(\ln \ln n)$;

- **Estado C:** Estado em que o servidor difunde mensagens com o algoritmo de puxar. Cada nodo permanece neste estado por O(ln ln *n*) rodadas e depois passa para o estado D.
- **Estado D:** similar ao estado removido de Demers.

O algoritmo de Karp ainda conta com a "regra do meio". Entretanto, para que ela seja passível de ser aplicada deve-se extrair de cada mensagem o valor **m'** que ela carrega. Ainda, contabiliza-se, para um dado nodo com contador global de valor **m**, a quantidade de vezes que $m' \geq m$ e a quantidade de vezes que $m' < m$. Se o primeiro valor superar o segundo, este servidor passa para o estado B-**m+1** (e, portanto, incrementa o valor do seu contador global em uma unidade).

Com esta regra do meio, tem-se a certeza de que nodos em estado B-m passarão ao estado C quando ou m chegar ao valor constante de O(ln ln *n*) ou receber uma mensagem de um nodo no estado C, que previamente já recebeu mensagens suficientes para transitar a este estado. Com este controle, tem-se que as mensagens serão empurradas aos outros nodos até quando este tipo de comunicação for favorável e, quando essa condição não existir mais, as mensagens serão puxadas pelos nodos ainda desinformados.

2.3 Protocolos para disseminação confiável

O protocolo que será usado como objeto de estudo neste trabalho, o NEeM (PEREIRA 2000), se baseia nos conceitos básicos descritos acima e em estratégias testadas em protocolos anteriormente desenvolvidos por outros grupos de pesquisa.

Alguns outros protocolos foram considerados para estudo, como:

- HyParView: desenvolvido por Leitao (2007), o Hybrid Partial View é um protocolo que, assim como o NEeM, usa conexões TCP/IP com seus vizinhos para aumentar a sua confiabilidade e tratamento de falhas. Como característica marcante, cada nodo desse algoritmo usa dois conjuntos de vizinhos, os ativos (em quantidade igual a *fanout*+1) e os passivos. Se um dos vizinhos ativos apresenta uma falha (detectada rapidamente graças ao uso de TCP/IP), um vizinho passivo é "promovido", o que faz com que sempre, cada nodo, se conecte com *fanout*+1 vizinhos.

- HEAP: desenvolvido por Frey (2009), o HEterogeneity-Aware Gossip Protocol se preocupa em aumentar a eficiência da disseminação de rumores manipulando dinamicamente o *fanout* de nodos. Nodos mais eficientes (com maior quantidade de banda, para tráfego de dados, disponível) tem seu *fanout* aumentado, enquanto os menos eficientes tem seu *fanout* diminuído na mesma proporção. Dessa forma, a heterogeneidade (a diferença entre capacidade de banda disponível em cada nodo) da rede é respeitada.

A escolha do NEeM se deve ao fato dele já ter sido utilizado em trabalhos anteriores do grupo de pesquisa na qual esse trabalho está incluído, como em Wilges (2012). Ainda, o NEeM já possui uma interface de monitoramento pronta, o que facilitou a criação do protótipo de monitoramento e análise que será explicado nos capítulos 4 e 5.

Portanto, como esse trabalho se debruçará sobre o NEeM, é natural descrevermos os protocolos que mais influenciaram sua criação. O primeiro deles, o Pbcast de Birman (1999), é uma evolução do algoritmo original descrito em Demers (1987). O objetivo dele é manter estável a transferência de dados, de uma ponta a outra da comunicação. Ainda, naquele trabalho são sugeridas diversas otimizações que serviram de base para trabalhos futuros, além de ser definida uma política de gerência de rumores conhecidos.

Um destes trabalhos "futuros", o Lpbcast de Eugster (2004), se preocupa em melhorar a forma como os nodos conhecem seus vizinhos mais distantes e em como os rumores são descartados.

Esta estratégia de descarte de rumores e a garantia de uma entrega fim-a-fim é discutida e refinada no NEeM.

2.3.1 Pbcast: um protocolo de multicast bimodal

Na área de comunicação de grupo e difusão de mensagens, os trabalhos anteriores ao Pbcast (acrônimo para *probabilistic broadcast*) de Birman (1999), também chamado de "multicast bimodal", eram focados em uma de duas abordagens para prover uma forma de multicast confiável:

- Garantia de propriedades fortes de confiabilidade, como atomicidade (se um dado nodo receber uma mensagem multicast, todos deverão recebê-la) ou ordenação consistente (manter a ordem de entrega de mensagens);
- Garantir de que cada nodo localmente irá se recuperar de perda de mensagens ou outras falhas e prover escalabilidade.

Um problema com a primeira abordagem era que "para obter essas fortes propriedades de confiabilidade é preciso empregar custosos protocolos, aceitar a possibilidade de um comportamento instável ou imprevisível diante de uma carga de mensagens acentuada e tolerar limites na escalabilidade" (BIRMAN 1999, p. 2).

Já a segunda abordagem pecava por não prover confiabilidade fim-a-fim, já que esses protocolos se baseavam em técnicas de "melhor esforço" (algum mecanismo razoável que possa ser aplicado a um nodo que tenha apenas conhecimento local) para evitar ou se recuperar de falhas.

O objetivo do Pbcast era definir um algoritmo epidêmico que mantivesse a estabilidade na taxa de transferência de dados (algo que nenhuma das abordagens anteriores visava) e garantisse a manutenção da confiabilidade da comunicação, mesmo diante de falhas. Birman afirma, contudo, que o objetivo não era unir as duas abordagens vigentes de multicast confiável, dado que o Pbcast não foi criado com o objetivo de prover propriedades rigorosas de sistemas confiáveis.

Tradicionalmente, os tipos de falhas consideradas em protocolos epidêmicos eram apenas do tipo permanentes (*hard)*, as quais englobam falhas em processos ou particionamentos na rede. O Pbcast também tratava falhas evasivas (*soft)*, do tipo em que mensagens enviadas não eram corretamente recebidas (provavelmente devido a problemas de estouro de buffers ou filas), visto que essas falhas causavam um distúrbio na taxa de transferência de dados que o protocolo se propunha a manter. Falhas arbitrárias ou maliciosas (também conhecidas como *bizantinas*) não eram cobertas.

O Pbcast operava em dois modos, executados concorrentemente:

- **Disseminação otimista:** disseminação de mensagens por um protocolo multicast, sem garantia de confiabilidade;
- **Anti-entropia de duas fases:** baseando-se no conceito da anti-entropia de Demers, o Pbcast usa esse modo para resolver as diferenças de histórico de mensagens entre dois nodos. Randomicamente, membros escolhiam parceiros para quem enviar um resumo do histórico de mensagens (*digest*) que ele possuía. O parceiro escolhido solicitava ao membro o reenvio das mensagens que ele não possuía, puxando rumores desconhecidos.

Para que o modo de anti-entropia de duas fases funcione seria necessário que cada processo mantivesse um conjunto de rumores conhecidos. Para manter fixa a quantidade de rumores conhecidos, o Pbcast ordenava-os pela rodada em que cada um foi recebido e descartava os mais antigos. O controle de "quão mais antigo" se dá através do parâmetro que controla a quantidade de vezes que uma mensagem deve ser disseminada. Esta característica de descarte auxilia nodos que sofrem perdas de mensagens transientes, já que eles são direcionados a darem atenção às mensagens mais recentes na rede e podem desconsiderar as mais antigas (que, provavelmente, ele já tenha perdido).

Várias otimizações ao modo de anti-entropia de duas fases foram sugeridas para diminuir os custos do protocolo diante de cenários com falhas.

A primeira delas sugere que mensagens apenas são reenviadas a nodos que as solicitarem na mesma rodada na qual a mensagem original foi enviada. Se a resposta a uma solicitação de retransmissão levar um tempo maior que uma rodada, então ou a rede ou o nodo a qual foi pedida essa mensagem apresenta problemas e, portanto, mandar essa mensagem na rede tem uma probabilidade baixa de ser efetiva.

A segunda otimização limita o número de bytes por rodada que um nodo pode enviar quando for solicitado a retransmitir mensagens. Essa estratégia auxilia a divisão de carga pela rede e impede que um nodo fique sobrecarregado com retransmissões.

A terceira otimização sugere que seja feito um controle acerca das mensagens não-entregues, ou seja, que sofreram falhas de transmissão e daquelas que foram mais requisitadas para serem reenviadas em rodadas anteriores. Sugere-se, portanto, manter um ciclo de reenvio destas mensagens. Caso essa não fosse a abordagem usada, talvez fossem reenviadas mensagens que ainda estão em trânsito, causando redundância de mensagens na rede.

A quarta otimização formaliza a ordem de reenvio de mensagens, sugerindo que se dê preferência a retransmitir as mensagens mais novas. De outra forma, ou seja, dando-se preferência para retransmitir mensagens mais velhas, os nodos que se recuperem de falha transientes vão sempre querer recuperar as suas mensagens perdidas e ficarão sempre defasados em relação às mensagens mais recentes.

A quinta otimização se refere ao limite de rodadas do protocolo. Seria de se pensar que esse número deveria ser o mesmo em todos os nodos e ser mantido sincronizado, mas, como as decisões sobre descarte e envio de mensagens só levam em conta informações locais de cada nodo, não é necessário esse controle.

A sexta otimização ajuda a tornar o protocolo mais escalável, afirmando que um nodo não precisa ter conhecimento de todos os outros nodos. Essa afirmação parte do princípio de que já existe uma estrutura de interconexões existentes (talvez a própria Internet) que faz com que, através de um sub-conjunto de vizinhos, um nodo possa alcançar e mandar mensagens a todos os membros da comunicação multicast.

A sétima otimização faz com que, caso um nodo receba dois (ou mais) pedidos para retransmitir uma mesma mensagem, ele a retransmita imediatamente por multicast. A probabilidade de que isso ocorra é baixa, exceto em casos em que um grande número de nodos não possua mais a mensagem (e ainda a quer). Por exemplo, pode ser que um sub-conjunto de vizinhos (criado por causa da otimização anterior) específico não tenha mais aquela mensagem.

2.3.2 Lpbcast: um protocolo de multicast probabilístico econômico

Eugster (2001) desenvolveu o Lpbcast (*lightweight probabilistic multicast*), baseado no Pbcast de Birman (1999), para explorar mecanismos que possam:

- Fazer com que nodos conheçam outros membros interessados em rumores, não apenas seus vizinhos;
- Melhorar a política de descarte de mensagens.

O primeiro problema foi parcialmente abordado no Pbcast. Nele, sugeriu-se deixar a gerência de membros para outro protocolo, o Astrolabe, desenvolvido somente para essa finalidade. O Lpbcast, por outro lado, sugeriu que esse mecanismo fosse incorporado às mensagens de disseminação de rumores. Assim, cada mensagem disseminada na rede contém um grupo de membros inscritos (*subscribers*) e não inscritos (*unsubscribers*) para receber rumores, além de informações sobre quais rumores são conhecidos pelo nodo emissor (similar ao *digest* do Pbcast). Com essas informações o nodo atualiza a sua visão da rede e, a partir dela, escolhe aqueles com quem vai trocar informações em cada rodada.

Para manter essas “visões” dos nodos da rede atualizadas, precisa-se periodicamente disseminar mensagens, mesmo que nenhum rumor novo tenha sido conhecido. A quantidade de membros em cada “visão” também é limitada.

Em relação à política de descarte, o Pbcast é regido por um parâmetro que define quantas vezes o rumor será retransmitido antes de ser “esquecido” por um nodo. Já no Lpbcast definiu-se um fator de “envelhecimento” da mensagem, ou seja, uma associação de cada rumor a um número inteiro que represente quantas vezes aquela mensagem foi disseminada (em rodadas). Também é sugerida a estratégia de que a idade de uma mensagem já conhecida (em buffer) seja incrementada quando uma cópia dela é recebida pelo nodo, fazendo com que a idade também possa ser um reflexo de redundância de mensagens na rede.

Periodicamente são descartadas as mensagens mais “antigas” até que o limite do buffer seja alcançado.

A estratégia de contagem de cópias de mensagens conhecidas é sugerida também para auxiliar na manutenção da visão da rede de cada nodo. Para isso, a cada mensagem de cada membro da “visão” é associado um contador de frequência de mensagens.

Quando se precisa escolher um nodo para ser removido da "visão", calcula-se a média de frequência de todas as mensagens e escolhe-se um nodo randomicamente: ele é removido se sua frequência for maior que a média; caso contrário incrementa-se sua frequência em uma (1) unidade. Pode-se ponderar a média por uma constante *k* entre zero (0) e um (1).

2.3.3 NEeM: Network Friendly Epidemic Multicast

Tipicamente, protocolos multicast são baseados em UDP/IP (KUROSE, 2005) já que não existe o conceito de conexão ponto a ponto: um nodo se comunica com um grupo. Protocolos epidêmicos oferecem uma maior confiabilidade à comunicação, pois, teoricamente, a carga e consequentes erros de entrega de mensagens ficarão distribuídos na rede. Se essa teoria se provar falsa e houver congestionamento num determinado conjunto de conexões, mensagens em sequência deixarão de ser entregues.

Pereira (2003) propõe, portanto, o uso de protocolos epidêmicos baseados em TCP/IP (KUROSE, 2005), que provê controle de congestionamento nativamente em comunicações epidêmicas. Nestes protocolos, tipicamente, cada nodo conhece alguns outros nodos e troca informações com eles, e não com o "grupo" inteiro. Desta necessidade foi criado o NEeM, um protocolo amigável-a-rede no sentido de não prejudicar a rede em períodos de congestionamento.

Outros pontos que justificam o uso de protocolos baseados em TCP/IP são:

- Internet services providers (ISPs) priorizam tráfego TCP/IP para garantir melhor tempo de resposta e escalabilidade. Por isso, pacotes pertencentes a outros protocolos tendem a ser descartados caso necessário;
- Caso as conexões físicas entre nodos sejam compartilhadas com outras aplicações há um aumento na probabilidade de congestionamentos ocorrerem. Com TCP/IP controlando seu funcionamento para evitar congestionamentos, as mensagens ficam acumuladas nos buffers dos nodos de origem, ao invés de serem perdidas. Abre-se, portanto, a possibilidade de se definir um critério de escolha para quais mensagens devem ser descartadas em caso de estouro do limite deste buffer;

Três estratégias são apresentadas por Pereira (2003) para decidir, num eventual problema de congestionamento, quais mensagens devem ser descartadas caso haja estouro no espaço do buffer, como segue:

- **Descarte randômico de uma mensagem:** interessante somente como estratégia secundária;
- **Descarte baseado em idade:** similar à estratégia do Lpbcast de Eugster (2001), descarta-se a mensagem que tenha sido disseminada mais vezes, já que provavelmente outros nodos, não congestionados, também já a possuem e possam disseminá-la. Diferente de Eugster, porém, a informação de idade é carregada pela mensagem, não sendo controlada localmente. Ao receber uma mensagem com idade *x* o nodo ira retransmiti-la com idade *x+1*;
- **Descarte baseado em semântica:** descarte de mensagens consideradas obsoletas pela aplicação que faz uso do protocolo epidêmico. Sugere-se que cada mensagem carregue um mapa de bits que, combinado com um

> identificador único entre mensagens, teremos que, "se o *i*-ésimo bit estiver ligado no bitmap pertencente a mensagem *j*, o protocolo é informado que a mensagem com número de sequência *j – i* é considerada obsoleta" (EUGSTER 2003, p. 7).

Uma quarta estratégia seria considerar um buffer first-in/first-out (FIFO). Essa estratégia é usada no NEeM 0.7[1], usado nas simulações deste trabalho.

Por fim, Pereira (2006) discute a melhor estratégia de transmissão a ser usada, dentre:

- Cada mensagem enviada carrega o "rumor" (*payload*);
- Manda-se uma mensagem de anúncio do rumor. Aqueles nodos que se interessarem devem responder. A eles, somente, será enviado o rumor inteiro. Essa estratégia se assemelha a fase de anti-entropia de duas fases do Pbcast de Birman (1999)**.** Porém, enquanto o Pbcast faz o nodo interessado "puxar" o rumor, no NEeM o nodo interessado apenas confirma o anúncio e pede a transmissão;

Essa escolha influencia como os recursos da rede são utilizados pelo protocolo. Pereira diz que essa escolha não precisa ser única e faz do NEeM um protocolo híbrido, ou seja, que opera com as duas estratégias.

2.3.4 Detalhes de Implementação do NEeM 0.7

Na seção anterior, foram discutidos os motivos pelo qual o NEeM[1] foi criado e quais estratégias usadas por ele. Neste capítulo será descrito como esse algoritmo foi implementado procurando-se traçar um paralelo com o que foi descrito em Pereira (2003, 2006) e em Carvalho (2007). É importante entender a quais camadas cabem determinadas responsabilidades e como é feita a relação entre elas afim de melhor entender como monitorá-las, o que será explorado no próximo capítulo.

O usuário da biblioteca NEeM 0.7 deve realizar os seguintes passos para utilizá-la:

1. Instanciar a classe MulticastChannel, passando como parâmetros o endereço e porta local a serem usados para abrir conexões;
2. Abrir conexões (TCP/IP) com os vizinhos imediatos;
3. Periodicamente, ler dados do canal (com uma chamada para o método MulticastChannel.read) ou escrever dados no canal (com uma chamada para o método MulticastChannel.write)

Assim que o objeto MulticastChannel é instanciado, ele cria uma instância das classes Transport, Overlay e Gossip.

Essa interface do usuário com o objeto MulticastChannel é similar a descrita em Pereira (2006) e em Carvalho (2007), na qual o usuário chama o procedimento "Multicast(d) para disseminar uma mensagem com a informação *d*".

[1] http://neem.sourceforge.net/

2.3.4.1 Classe Transport

O objeto desta classe simula a camada de rede, e serve de camada de interface para as outras mandarem e receberem pacotes da rede.

Esta camada também atende as requisições de entrada ou saída (E/S) de qualquer conexão. No intuito de ter um desempenho aceitável, todas as conexões são controladas por um único laço de execução. Quando uma conexão possuir algum evento (seja um dado a ser escrito ou lido na conexão, ou quando uma conexão é aceita ou destruída), ela notifica o objeto Transport. Essa notificação de múltiplas conexões é proporcionada pela biblioteca "java.nio", um pacote adicionado ao Java 1.4 para permitir a gerência de conexões não-bloqueantes. Em Pereira (2006) e em Carvalho (2007) é feita a referência ao uso dessa biblioteca pelo NEeM.

Para permitir que outros objetos "escutem" dados que trafeguem em determinadas portas, geralmente para controle interno do protocolo, o objeto Transport possui uma lista de DataListeners. A cada DataListener é associado a uma porta. Seu método *receive* é chamado quando uma mensagem for totalmente recebida naquela porta.

Quando um objeto deseja, na verdade, monitorar as conexões criadas ou destruídas, o objeto Transport permite a existência de um ConnectionListener que é chamado quando um destes eventos acontece. Na arquitetura do NEeM, o objeto Overlay se cadastra como ConnectionListener no objeto Transport, afim de ser informado sobre novos vizinhos (ou a desistência de antigos).

Ainda, essa camada possui uma lista de tarefas, que são executadas num determinado tempo. Cada nova tarefa é agendada pelo objeto Transport para ser executada posteriormente (geralmente no próximo nano-segundo sem tarefas a executar). Essa arquitetura de tarefas permite que outras camadas deleguem ao laço de execução do objeto Transport algumas ações, permitindo a apenas uma linha de execução controlar o que precisa ser feito e atender a requisições.

Quando o usuário, em posse de um objeto MulticastChannel, deseja escrever um dado no canal ou se conectar com um vizinho imediato (passo 2 descrito na seção anterior), ele deve criar uma tarefa no objeto Transport. Essas tarefas podem ainda ser periódicas, como o método de *shuffle* do objeto Overlay ou o método de *retransmit* do objeto Gossip.

Cada pacote recebido ou enviado por esse objeto apresenta um cabeçalho de 6 bytes antes do seu real conteúdo (de responsabilidade dos objetos Overlay ou Gossip), dos quais os 4 bytes mais significativos codificam o tamanho total do pacote e os 2 bytes restantes a porta a ser usada para enviar ou receber esse dado. Quando um pacote é recebido por inteiro, ou quando conexões são abertas ou fechadas, o próprio objeto Transport, ao ser avisado, cria uma tarefa para si mesmo, ativando o DataListener ou ConnectionListener adequado.

2.3.4.2 Classe Overlay

Já o objeto da classe Overlay se encarrega de gerenciar o anúncio e criação da vizinhança de um nodo. Ela se comunica com o objeto Transport, quem realmente cuida da criação e destruição de conexões, e provê ao objeto Gossip acesso à vizinhança do nodo. A quantidade de vizinhos é limitada pelo parâmetro *Overlay.fanout*, que por padrão tem o valor 15.

Essa classe faz o papel do Peer Sampling Service descrito em Pereira (2006) e Carvalho (2007).

Diferentemente do Lpbcast de Eugster (2001) e do algoritmo original em Pereira (2003), o compartilhamento da vizinhança não é feito no corpo das mensagens de dados. Um objeto Overlay usa três canais TCP/IP somente para esse controle, criados como DataListeners no objeto Transport descrito anteriormente: os canais de ID, de Shuffle e de Join. Cada mensagem desse objeto carrega um identificador único de 16 bytes e um endereço de rede (cujo endereço IP ocupa 4 bytes mais significativos e a porta utilizada ocupa os 2 bytes menos significativos).

Ao receber uma mensagem pelo canal de Join, um nodo manda a todos os seus vizinhos uma mensagem de anúncio da presença deste remetente pelo canal de Shuffle, o que ajuda a torná-lo rapidamente conhecido.

Ao receber uma mensagem pelo canal de ID, o nodo insere o remetente da mensagem como um novo vizinho e manda uma mensagem pelo canal de Join para pedir ao vizinho que divulgue a presença dele na sua própria vizinhança. Sabendo que mensagens de Join causam um tráfego de dados excessivo na rede, uma mensagem recebida pelo canal de ID só gera o envio de dados pelo canal de Join se a vizinhança do nodo era vazia antes de receber essa mensagem.

Ao receber uma mensagem pelo canal de Shuffle, o nodo adiciona o remetente a sua vizinhança, se ele ainda não estiver nela e se a quantidade de vizinhos for menor que o *Overlay.fanout*. Se não for esse o caso, ele ainda pode adicionar um novo vizinho, com 50% de probabilidade. Se mesmo assim o remetente não for adicionado a sua vizinhança, o nodo anuncia a presença deste para algum vizinho (escolhido randomicamente) que, talvez, possa aceitar o remetente.

Sempre que uma nova conexão com um vizinho é criada, é enviada uma mensagem de ID, para se anunciar ao novo vizinho e pedir para ele adicionar o nodo a sua própria vizinhança. Também na criação da conexão é feito o corte de conexões, escolhidas randomicamente, para que elas se mantenham limitadas pelo valor do parâmetro *Overlay.fanout.*

Por fim, o objeto Overlay cria uma tarefa no objeto Transport para, periodicamente (por padrão, 10 segundos), escolher dois vizinhos imediatos e anunciar um deles ao outro (através do canal de Shuffle). Esse período pode ser alterado (ou lido) em tempo de execução através de diretivas JMX (descritas no próximo capítulo) que alterem (ou leiam) o dado Shuffle Period.

2.3.4.3 Classe Gossip

O objeto Gossip controla a disseminação dos dados, providos pelo usuário através do objeto MulticastChannel, para uma certa quantidade de vizinhos, providos pela classe Overlay.

Vários parâmetros são usados nesse objeto, como segue. Todos serão usados com o prefixo "*Gossip.*" daqui para frente:

- **Fanout**: controla a quantidade de mensagens (por rodada) a serem enviadas. Também pode ser visto como a quantidade de vizinhos com os quais o nodo conta um rumor;

- **TTL**: o número máximo de vezes que uma mensagem pode ser disseminada. É incrementado de 1 (uma) unidade a cada re-envio de uma mesma mensagem;
- **PushTTL**: o valor mínimo que uma mensagem deve ter para ela ter seu envio postergado. Nestes casos, apenas um *ack* é enviado (e não a carga de dados inteira);
- **MinPullSize**: o tamanho mínimo que uma mensagem deve ter para ela ser considerada passível de ter seu envio postergado. Nestes casos, apenas um anúncio *ack* é enviado;
- **PullPeriod**: o período da tarefa periódica de transmissão de mensagens de *nack*.

Para possibilitar a disseminação de rumores, dois DataListeners são criados no objeto Transport: o de dados e o de controle. No canal de dados apenas trafegam mensagens, conforme o usuário as envia (através do objeto MulticastChannel). No canal de controle trafegam anúncios de *ack* (ou "eu tenho", que anunciam que o nodo possui determinada mensagem) ou de *nack* (ou "eu quero", que anunciam que um nodo tem interesse em determinada mensagem).

Cada mensagem ou anúncio possui três campos: um identificador único de 16 bytes (que ajuda a identificar se uma mensagem é repetida ou não), um contador de por quantos *hops*, ou nodos, aquele rumor passou (o que também pode ser encarado como a "idade" da mensagem, como descrito em PEREIRA (2003)), que ocupa 1 (um) byte somente, e o conteúdo da mensagem, de tamanho variável (no caso de anúncios, é vazio).

Duas filas FIFO (First-in, First-out) são usadas: a de *cached*, controlando as mensagens conhecidas e a de *queued*, controlando os anúncios conhecidos.

Quando um anúncio de *ack* é recebido e o nodo não possui a mensagem na fila *cached*, ele adiciona esse anúncio, juntamente com o remetente, à fila de *queued*. Com isso, o nodo controla quais dos seus vizinhos possuem a mensagem completa, ou seja, a quem ele precisa pedir essas mensagens. Esse comportamento é idêntico ao descrito pelo procedimento Receive(IHAVE) em Pereira (2006) e Carvalho (2007).

Quando um anúncio de *nack* é recebido e o nodo possui a mensagem na fila *cached*, ele manda a mensagem completa. De forma similar ao tratamento de *ack*s, esse comportamento é idêntico ao procedimento Receive(IWANT) de Pereira (2006) e Carvalho (2007).

Quando uma mensagem completa é recebida, os seguintes passos são executados:

1. Se o nodo já possui a mensagem na fila de *cached*, ele apenas descarta esta duplicata e encerra o tratamento dela;
    a. Em caso negativo, é inserida na fila de *cached*;
2. Se o nodo já possui o anúncio para essa mensagem na fila de *queued*, ele remove o anúncio;
3. Se for uma mensagem recebida por algum vizinho, encaminha-se essa mensagem para a aplicação resgatá-la através do objeto MulticastChannel;
4. Se a quantidade de vezes que aquela mensagem foi re-transmitida (o número de *hops* dela) é maior que ou igual ao parâmetro *Gossip.TTL*, encerra o tratamento dela sem disseminá-la aos vizinhos;
5. Decide qual estratégia usar: disseminar um *ack* na rede ou enviar a mensagem inteira

Esse comportamento é similar ao descrito no procedimento Forward de Pereira (2006) e Carvalho (2007). Contudo, a aplicação da política de enfileiramento da mensagem, se já existir um anúncio a ela, realizada no passo 1), cabe ao procedimento Receive(MSG) do mesmo artigo. Já a decisão sobre qual estratégia de envio usar, realizada no passo 5), é feita pelo método L-Send.

Essa decisão depende de dois fatores: a) a "idade" da mensagem (ou número de hops) e b) o tamanho da mensagem completa (em bytes). Quando ambos dos fatores superarem os parâmetros *Gossip.PushTTL* e *Gossip.MinPullSize*, respectivamente, apenas um anúncio será disseminado na rede; em caso negativo, a mensagem irá ser disseminada por completo. Em ambos os casos, se consulta a vizinhança a partir do objeto Overlay e se seleciona, randomicamente, *Gossip.Fanout* nodos que receberam essa mensagem ou anúncio.

Em Pereira (2006), se divide o conjunto de nodos selecionados para receberem dados em dois grupos: um deles vai receber a mensagens completas e o outro apenas anúncios delas. Já em Carvalho (2007), essa decisão é feita por um algoritmo que se baseia nas condições da rede. Por questões de simplicidade, este trabalho irá usar o comportamento implementado na versão 0.7, mais similar ao algoritmo apresentado por Pereira (2006).

Finalmente, uma tarefa é criada no objeto Transport para solicitação de retransmissão periódica (a cada *Gossip.pullPeriod* * 2 milisegundos). A cada execução dessa tarefa, o nodo varre sua fila de *queued* para descobrir quais os anúncios (*ack*) conhecidos e quais remetentes enviaram cada anúncio. Em Pereira (2006) e em Carvalho (2007) essa mesma tarefa é exercida pela Task2 da camada de Agendamento de Envio.

Anúncios muito antigos (enviados pelo último remetente a mais de *Gossip.pullPeriod* milissegundos) são ignorados, similar a ideia do Pbcast de Birman (1999) de orientar nodos a darem atenção apenas a rumores novos. Este controle é uma otimização ao algoritmo mostrado em Pereira (2006) e Carvalho (2007).

Para cada um dos rumores "mais novos", se destaca o último remetente dele (aquele que enviou o rumor mais recentemente) e se envia um anúncio de *nack* a este vizinho apenas. Esse remetente é então excluído da lista de controle daquele anúncio para se garantir que ele não seja mais usado em futuras requisições (idêntico ao descrito em Pereira (2006)).

2.4 Conclusão

O estudo de protocolos epidêmicos apresentado neste capítulo, culminando numa descrição detalhada do funcionamento do algoritmo e da implementação mais recente do NEeM, é a base para a análise e monitoramento dos dados gerados pelas aplicações de teste. Um monitoramento não pode ser feito sem se conhecer a fundo quais dados e eventos devem ser monitorados, assim como a análise dos dados gerados deve se basear no mesmo conhecimento para verificar se o comportamento observado é o mesmo que o esperado.

3 MONITORAMENTO DE TRÁFEGO DE REDE

Para que a análise dos dados seja possível, em capítulos posteriores, precisamos obtê-los de alguma forma. O NEeM nos permite fazer isso através da interface JMX. Essa forma de monitoramento é totalmente baseada na plataforma Java, o que faz com que a operação do algoritmo epidêmico não se preocupe com os dados sendo monitorados. Seus detalhes de funcionamento, assim como os detalhes da interface que o NEeM expõe a um monitor externo, serão vistas nesse capítulo.

3.1 Motivação

Conforme programas são executados e se comunicam em rede com outros programas ou serviços, cria-se a necessidade de uma estratégia de monitoração para entender e gerenciar o comportamento destes programas.

Também existe a necessidade de avaliar se tais programas mantêm níveis de qualidade de serviços adequados (do Inglês, *service level agremeents* ou SLA). Outra necessidade é detectar se esses programas foram alvo de invasões. Almgren (2001) faz um estudo das características de sistemas detectores de intrusões (do Inglês, *Intrusion Detection Systems* ou IDS). De acordo com o modo que estes sistemas coletam os dados, ele os chama de *network based data collection,* se eles monitoram os pacotes na rede, ou *host based data collection,* se monitoram os dados recebidos por uma máquina específica. Mesmo que este trabalho não foque no monitoramento dos atributos de segurança de um sistema, essa divisão será usada, já que pareceu natural pensar nesse modo de classificar protocolos dessa área.

Quando se deseja monitorar a atividade da rede, Finamore (2010) divide as estratégias existentes em passivas, se somente observam o tráfego gerado na rede, ou ativas, se injetam ou modificam o tráfego existente para induzir um efeito a ser medido.

Ele descreve um sistema de monitoramento passivo, criado primeiramente por Mellia (2005), o Tstat (*TCP Statistics and Analysis Tool*). Além de obter dados, esse sistema é capaz de gerar gráficos e dados estatísticos.

Quando se deseja monitorar os dados de uma máquina específica, geralmente se monitora o conjunto de servidores de uma aplicação. Para programas servidores desenvolvidos em Java existe a facilidade de contar com as extensões para gerenciamento (do Inglês, *Java Management Extensions* ou JMX). O NEeM, descrito na seção anterior, também expõe dados a agentes de monitoramento externos usando essa tecnologia.

3.2 Monitoramento ativo

Nesta categoria, segundo Finamore (2010) recaem tarefas “como gerenciamento da rede (ICMP), tomografia da rede (traceroute), medição de atrasos e capacidade (ping, capprobe, pathchar) e estudos empíricos (netem, dummynet)”.

O protocolo ICMP (*Internet Control Protocol*) é usado por roteadores, segundo a RFC 792, para informar a um IP de origem sobre diversas situações de erro. Uma situação comum que causa mensagens ICMP serem enviadas é quando o campo TTL (*Time to Live*) de um pacote chega a 0 (zero). Baseado neste funcionamento, aplicativos como o traceroute mandam rajadas de pacotes, com o campo TTL sendo incrementado linearmente, e aguardam o recebimento (ou não) destes pacotes de erro. Com a diferença de tempo entre envio e resposta, os aplicativos medem o tempo de resposta da rede ou descobrem informações sobre a topologia corrente.

Aplicações baseadas em ICMP injetam tráfego, mesmo que “artificial”, na rede a ser medida. Outras ferramentas, nas quais o dummynet Rizzo (1997) é o principal representante, interceptam a comunicação entre o protocolo a ser testado e a aplicação do usuário com o intuito de monitorar o comportamento da aplicação (modificando o comportamento do protocolo), do protocolo (modificando parâmetros na rede) ou auxiliar no trabalho de entender como essa comunicação ocorre.

Como as soluções descritas (e similares) modificam as condições da rede ou os dados que trafegam nela, elas são muitas vezes descartadas. Porém, é fácil ver como o dummynet pode se comportar de uma maneira menos intrusiva, apenas copiando dados referentes à comunicação entre aplicação e protocolo para posterior análise.

3.3 Monitoramento passivo

De acordo com Finamore (2010), nesta abordagem “a troca de dados na rede é meramente observada, com o intuito de inferir algumas propriedades principais, mais tomando cuidado para não interferir com os dados observados”.

Segundo Kurose (2005), uma ferramenta básica para observar o tráfego de vários protocolos é chamada de um *packet sniffer* (ou somente *sniffer*), uma ferramenta que, passivamente, copia mensagens sendo enviadas ou recebidas por um computador. Um *sniffer* bem conhecido é o Wireshark[1], que ainda permite ao usuário analisar e interagir com os dados obtidos.

Já o TCPTrace[2] se preocupa em criar estatísticas a partir dos dados de um único fluxo TCP/IP. Estes dados são obtidos por um outro programa como, por exemplo, o tcpdump, um *sniffer* que já vem instalado com sistemas operacionais GNU/Linux.

Por outro lado, o Tstat de Mellia (2005) agrupa as funções de coleta e criação de estatísticas. Ainda, ele não se limita ao protocolo TCP/IP, e também processa dados de outros protocolos da camada de transporte, como o UDP/IP, ou de aplicação, como o BitTorrent ou o Skype.

---

[1] www.wireshark.org/

[2] www.tcptrace.org/

O projeto desenvolvido nesse TCC irá usar uma abordagem de monitoramento que seja integrada ao módulo estatístico, da mesma forma que o Tstat. Similar a ele, também estamos interessados nos dados da camada de aplicação. Ainda, similar a forma como o TCPTrace usa dados obtidos pelo tcpdump, o formato de dados usado neste trabalho é legível por outras aplicações.

3.4 JMX: Instrumentação da Máquina Virtual Java

Java Management Extensions (JMX) é, de acordo com a especificação de sua versão 1.4 (SUN 2004), uma definição de "arquitetura, design patterns, APIs e serviços para aplicações e gerenciamento de rede e monitoramento na linguagem de programação Java". Estas extensões foram usadas inicialmente apenas pela edição *enterprise* da plataforma Java (plataforma J2EE, ou *Java 2 Enterprise Edition*) e somente foram incorporadas na edição padrão (plataforma J2SE, ou *Java 2 Standard Edition*) na sua versão 5.0. A partir desta versão, qualquer solução desenvolvida na plataforma Java pode fazer uso dessa tecnologia para permitir um fácil gerenciamento do sistema, de forma escalável e segura.

Sistemas servidores de serviços, como o JBoss, usam a tecnologia JMX para permitir que um operador monitore e configure, remotamente, uma aplicação. Sistemas distribuídos recentemente começaram a perceber a importância de prover facilidades de monitoramento.

Protocolos de comunicação epidêmica, por outro lado, não tinham possibilitado formas padrão de monitoramento até a implementação do NEeM 0.7, que já conta com uma interface JMX pronta para ser usada. Faz parte deste trabalho, portanto, construir uma ferramenta de monitoramento remoto baseada em JMX para estudar este protocolo.

Aplicações que fazem uso de monitoramento baseado em JMX delegam ao desenvolvedor do sistema a responsabilidade de definir o que deve ser monitorado. Soluções anteriores atribuíam essa responsabilidade a administradores de sistemas. Sem acesso ao código, eles precisavam coletar a maior quantidade de dados (úteis ou não) possível para depois analisar e tentar compreender o comportamento de uma aplicação baseado nestes dados. Com JMX, o desenvolvedor se torna responsável por definir quais dados devem ser vigiados (como quantidade de conexões persistentes, ou threads inativas e ativas). Assim, podem ser feitas medidas semanticamente mais ricas. No caso do NEeM, pode-se medir quantos anúncios um dado nodo mandou na rede ou quantos rumores o usuário disseminou.

3.4.1 Visão geral

A arquitetura JMX é dividida em três camadas:

- **Instrumentação:** a camada que especifica como são implementados os recursos a serem gerenciados;
- **Agente:** a camada que especifica como agentes de gerenciamento de recursos serão implementados. Ela serve de interface do mundo externo aos recursos instrumentados pela camada anterior;
- **Serviços distribuídos**: a camada que gerencia os recursos disponíveis, através de aplicações gerenciadoras (*management applications*).

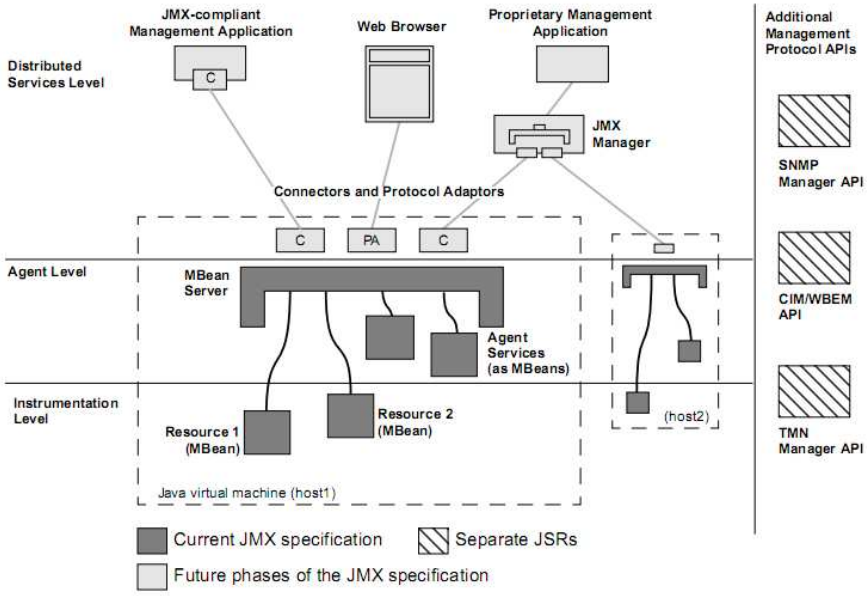


Figura 3.1: Relação entre componentes das camadas da arquitetura JMX

Fonte: SUN 2004

3.4.2 Camada de Instrumentação

Classes que obedecem a convenções de codificação e que se comportam como partes reusáveis de software são chamadas de JavaBeans. No JMX, existem dois tipos de Managed Beans (MBeans), pedaços de software responsáveis por tornar recursos gerenciáveis:

- **MBeans Padrão (Estáticos)**: classes simples que expõem explicitamente sua interface gerenciável;
- **MBeans Dinâmicos**: classes que não possuem uma interface explícita em tempo de compilação, mas em tempo de execução (*runtime*).

Ambos os tipos de MBeans foram criados de forma que sejam flexíveis, simples e fáceis de implementar.

No caso de MBeans Estáticos, devem ser respeitadas algumas convenções, como:

1. Criar uma interface com sufixo *MBean* adicionado ao nome da classe que irá implementá-la. Apenas as operações nessa interface podem ser acessadas por agentes gerenciadores;
2. Implementar todas as operações definidos na interface com sufixo *MBean*;

Como apenas as operações da interface são visíveis, elas precisam contemplar operações de acesso a qualquer atributo da classe a ser monitorada.

Já MBeans Dinâmicos não precisam prover uma interface em tempo de compilação. Porém, estas classes precisam implementar a interface DynamicMBean, que especifica 6 operações:

- **getMBeanInfo:** um método que deve retornar um objeto que encapsula a definição da interface (operações e atributos) a ser considerada para este MBeans;
- **getAttribute e getAttibutes:** métodos que recebem o nome de um atributo (ou uma lista de nomes) e retorna o(s) valor(es) deste(s) atributo(s);
- **setAttribute e setAttibutes:** de maneira similar ao método acima, estes escrevem um (ou mais) valores no(s) atributo(s) especificado;
- **invoke:** um método que "invoca" uma operação do MBean.

Para que MBeans possam ser utilizados da mesma forma por aplicações gerenciadoras, independente de qual tipo pertencem, MBeans Estáticos também são acessados através dos métodos descritos acima. Como não é da responsabilidade deles prover essas operações descritas, cabe a um Servidor MBean ser acessado pela aplicação (através destes métodos) e descobrir a interface pré-definida pelo MBean Estático. MBeans Dinâmicos, por outro lado, assumem a responsabilidade em implementar estes métodos.

Ainda, MBeans podem emitir (ou escutar) notificações enviadas por outros MBeans. Estas notificações podem ser criadas quando um atributo do MBean é modificado. MBeans que desejam emitir ou escutar notificações devem implementar interfaces específicas.

Servidores MBeans descobrem as funcionalidades de MBeans Estáticos graças a criação, em tempo de compilação, de metadados referentes a ele, um conjunto de classes especiais que descreve suas operações, atributos e notificações.

3.4.3 Camada de Agente

Nesta camada se concentram componentes que permitem que aplicações gerenciadoras interajam com MBeans de maneira segura e escalável.

O principal agente é o Servidor MBean, que detêm as funcionalidades de registro de MBeans e de controle de acesso a eles. Como ponto único de acesso aos dados monitorados, existem formas de modificar o acesso às operações que ele provê às aplicações gerenciadoras. A aplicação a ser monitorada interage com o Servidor MBean para registrar quais seus recursos estarão disponíveis e sobre quais condições de segurança. Ainda, através do Servidor MBean aplicações gerenciadoras podem descobrir quais MBeans estão disponíveis.

Outros serviços dessa camada são: os conectores-servidores, que ativamente se comunicam com as aplicações gerenciadoras, através de conectores-clientes; o serviço m-let, que cuida da criação dinâmica de MBeans; o serviço de relações, que agrupam diversos MBeans num mesmo papel (*role*); e serviços de gerência de notificações de tempo.

3.4.4 Camada de Serviços Distribuídos

As aplicações gerenciadoras se comunicam com a camada de agentes através de conectores-clientes, que gerenciam a comunicação com o Servidor MBean, lidam com concorrência e enfileiram as notificações que este cliente deseja escutar.

O único conector-cliente que a especificação obriga existir é o conector RMI (Remote Method Invocation). RMI já fazia parte da plataforma J2SE antes do JMX existir. Seu objetivo era oferecer a possibilidade de comunicação entre objetos distribuídos. Com RMI se permite descobrir objetos remotos disponíveis, previamente cadastrados num servidor de nomes, e obter referências (ou cópias) deles.

Da mesma forma, uma aplicação de gerenciamento precisa se conectar a um Servidor MBean já conhecido e usar a conexão com ele para enviar comandos, ler ou escrever dados em MBeans previamente cadastrados.

3.5 Uso do JMX no NEeM:

O NEeM 0.7 usa MBeans Estáticos para expor seus dados a serem monitorados. A classe **Protocol** implementa a interface **ProtocolMBean** e provê acesso a dados dos três objetos principais da implementação do NEeM descritos no capítulo 2: Transport, Overlay e Gossip. Os atributos expostos pela classe Protocol são:

- **AcceptedSocks / ConnectedSocks**: quantidade acumulada sobre quantas conexões com vizinhos foram criadas (via chamadas *accept* ou *connect*, repectivamente) pelo objeto Transport;

- **BufferSize / QueueSize:** parâmetros no objeto Transport não usados pelo protocolo NEeM;

- **BytesReceived / BytesSent**: quantidade acumulada sobre quantos bytes foram recebidos ou enviados (respectivamente) por todas as conexões do objeto Transport;

- **DataReceived / DataSent**: quantidade acumulada de mensagens (não anúncios) recebidas ou enviadas (respectivamente) pelo objeto Gossip;

- **HintsReceived / HintsSent**: quantidade acumulada de anúncios *ack* recebidas ou enviadas (respectivamente) pelo objeto Gossip;

- **PacketsReceived / PacketsSent:** quantidade acumulada de pacotes enviados ou recebidos (respectivamente) pelo objeto Transport;

- **PullReceived / PullSent**: quantidade acumulada de anúncios *nack* recebidas ou enviadas (respectivamente) pelo objeto Gossip;

- **ShuffleReceived / ShuffleSent**: quantidade acumulada de mensagens recebidas ou enviadas (respectivamente) pelo canal de Shuffle do objeto Overlay;

- **Delivered**: quantidade acumulada de mensagens de dados devolvidas pelo objeto Gossip ao usuário;

- **GossipFanout / OverlayFanout:** parâmetros de Fanout dos respectivos objetos;

- **JoinRequests:** quantidade acumulada de mensagens recebidas pelo canal de Join do objeto Overlay. Útil para saber quantas rajadas de mensagens foram criadas para ajudar a inserir nodos novos na rede;

- **MinPullSize / PullPeriod / PushTimeToLive / TimeToLive:** valores dos respectivos parâmetros do objeto Gossip;

- **Multicast**: quantidade acumulada de vezes que o usuário chamou a diretiva multicast.

- **PurgedConnections:** quantidade acumulada de conexões destruídas pelo objeto Overlay quando se detectou que o número total de conexões ativas era maior que o parâmetro *Overlay.fanout*;

- **ShufflePeriod**: periodicidade da tarefa na qual um nodo escolhe dois vizinhos imediatos e anunciar um deles ao outro;

Estes atributos são considerados a interface do NEeM com o programa de análise e monitoramento que será desenvolvido nesse trabalho, e, portanto, serão extensivamente utilizados nos próximos capítulos.

3.6 Conclusão

Com a interface JMX do NEeM conhecida, sabemos quais das classes e eventos apresentadas no capítulo 2, quando se descreveu o funcionamento de protocolos epidêmicos, são passíveis de serem monitoradas. Os próximos capítulos se encarregam de construir uma aplicação que nos permita fazer esse monitoramento, de forma conjunta com a análise posterior dos dados.

4 PROJETO LUSIR (*LIVE USER INTERACTIVE REMOTE MONITOR)*

Os capítulos anteriores introduziram protocolos epidêmicos, a necessidade de monitorá-los para entender seu comportamento e como o NEeM provê uma interface que exibe os atributos a serem monitorados de maneira padronizada. Este capítulo, portanto, descreve como foi usada essa interface.

Um programa de monitoramento, com a interface JMX, pode obter, remotamente, esses dados. A cada rodada de amostragem, todos os atributos de um nodo remoto são obtidos. A própria máquina virtual Java obtêm dados sobre o uso do protocolo NEeM no nodo, remoto ao monitor, e controla os objetos MBeans criados.

Diferente das ferramentas de análise citadas no capítulo 3, que usam dados estáticos, já consolidados por programas de monitoramento (como o tcpdump), este trabalho provê a visualização dos dados, em tempo de execução, enquanto estão sendo obtidos (*live*). Ao usuário é dado o controle de selecionar qual dos atributos monitorados deve ser exibido.

Similar ao Tstat, um mesmo módulo é usado tanto para tarefa de monitoramento quanto para consolidar os dados obtidos em gráficos e efetivamente fazer a análise. Porém, diferente da maioria das ferramentas de visualização, a criação de gráficos fica sob responsabilidade do usuário: a ele é providenciada uma tabela com os dados obtidos e, através de um comando SQL, ele deve selecionar quais dados serão convertidos numa forma gráfica. Este tipo de controle dá ao usuário liberdade para experimentar com as suas amostras, mas exige dele um resultado pré-formatado de uma maneira específica, de modo que um gráfico seja corretamente gerado. O usuário pode salvar as suas consultas para utilizá-las em outra ocasião (talvez com outros dados).

Portanto, o programa proposto obtêm dados sobre o funcionamento de nodos remotos através da interface JMX e os exibe enquanto estão sendo obtidos, além de prover uma interface para o usuário interagir com os dados coletados. Faz sentido, portanto, chamá-lo de Monitor LUsIR (*Live User Interactive Remote Monitor)*. O fluxo dos dados monitorados segue na figura 4.1.

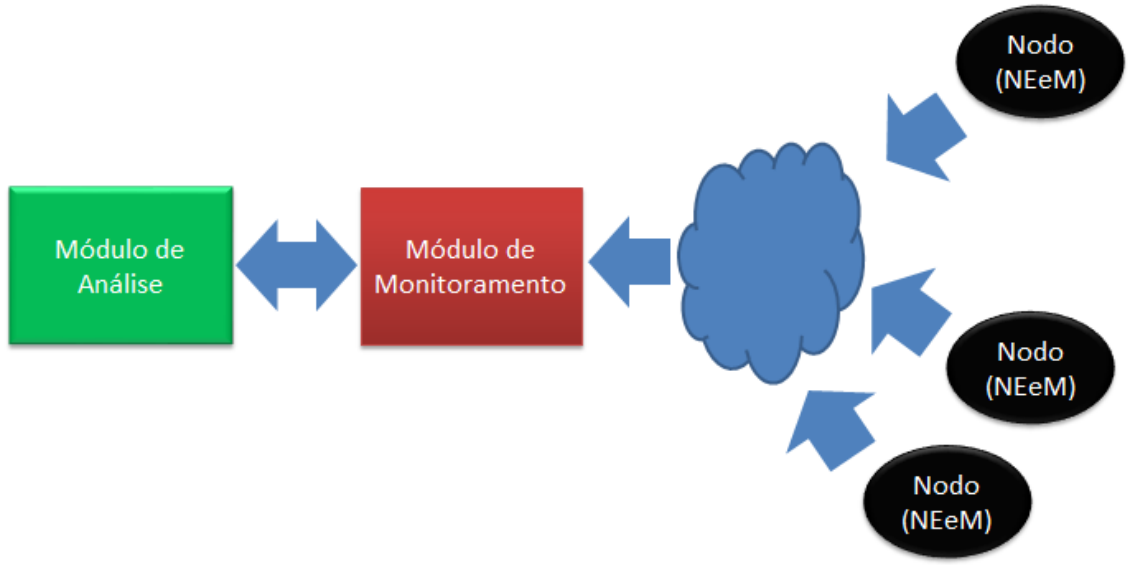


Figura 4.1: Fluxo de dados monitorados pelo protótipo LUsIR

4.1 Módulo de Monitoramento

Cabe a esse módulo se comunicar com os objetos distribuídos através da interface JMX de cada um, obter periodicamente as informações monitoradas e torna-las acessíveis ao módulo de análise.

O primeiro passo é obter acesso ao Servidor MBeans, um servidor potencialmente remoto onde todos os objetos distribuídos se registraram (como visto no capítulo 3). Ao criar uma conexão com esse servidor, o módulo de monitoramento tem como descobrir quais os objetos Protocol (a classe que implementa a interface MBean no NEeM 0.7) existem e, portanto, ter acesso a seus atributos.

Num dado instante de monitoramento, o módulo de monitoramento deve, para cada nodo, ler todos os atributos a serem monitorados. Também deve manter um registro dos dados monitorados no passado, a qual nodo esse dado pertence e em qual momento de tempo (ou instante de monitoramento) ele foi obtido.

Desse registro, alguns deles podem ser disponibilizados ao módulo de análise para que sejam mostrados dinamicamente, durante a obtenção deles. O restante dos dados deve ser armazenado em um arquivo para sofrer uma análise pós-experimento.

Como este módulo conhece os atributos do NEeM, ele consegue discernir quais deles apresentam dados acumulados. Destes, é interessante que ele consiga extrair qual foi o acúmulo ocorrido em cada momento de tempo, e não somente os valores totais. Para o módulo de análise, deve ficar claro que todos os atributos acumulados possuem o prefixo “Total”. Por exemplo, o atributo “TotalBytesReceived” no momento de tempo *t* apresenta a quantidade acumulada de bytes recebidos até aquele momento, enquanto o atributo “BytesReceived” apresenta a quantidade de bytes recebidos exatamente no momento *t*.

4.2 Módulo de Análise

A esse módulo é atribuída a responsabilidade de apresentar ao usuário os dados coletados e possibilitar a ele explorá-los.

Ainda, deve possibilitar ao usuário visualizar esses dados enquanto eles estão sendo coletados. O usuário deve controlar quais atributos (1 ou mais) ele deseja visualizar nesse momento de coleta de dados.

Ao apresentar os dados, é necessário deixar ao usuário selecionar qual formato seria melhor apresentá-los: de forma gráfica ou de forma tabular. O primeiro formato ajuda a relacionar padrões visuais que um algoritmo teria maior dificuldade. O segundo formato ajuda a entender como os dados estão distribuídos, quais devem ser apresentados numa forma gráfica e quais devem ser filtrados, ou quais devem sofrer alguma alteração (já que se torna mais útil exibir e analisar valores derivados desses dados brutos).

Essa atividade de exploração, que deve ser feita no formato tabular, deve ser feita usando comandos (ou *queries*) SQL (*Structered Query Language*). Quando se desejar tornar esses dados visíveis em formato gráfico, deve-se criar *queries* que modifiquem os dados para deixá-los em um formato específico (com determinado número de colunas e/ou linhas) para possibilitar a transformação em gráficos.

Para facilitar as explorações iniciais, podem ser disponibilizadas *queries* de amostra (*samples*), que fazem com que algumas métricas comuns (como a quantidade acumulada de Bytes Recebidos por segundo, por exemplo) possam ser visualizadas em forma gráfica.

4.3 Conclusão

O projeto LUsIR apresentado neste capítulo combina o monitoramento do capítulo 3 com o protocolo epidêmico descrito no capítulo 2. O próximo capítulo (o 5º) remete ao desenvolvimento de um protótipo que implemente esse projeto, enquanto o capítulo 6 irá descrever experimentos realizados com o mesmo.

5 PROTÓTIPO LUSIR

Com o projeto do LUsIR apresentado anteriormente, foi feito um protótipo na linguagem de programação JAVA que possibilite pôr em prática a ideia de se ter uma ferramenta que permita o monitoramento e análise de dados de maneira integrada. Nas próximas sessões ficará claro como o usuário deve usar o protótipo e os detalhes de seu funcionamento.

O protótipo possui três conjuntos de classes principais: um se responsabiliza apenas por objetos de tela e seus posicionamentos; outro, por coordenar os eventos resultantes das interações com o usuário; e um último que gerencia a obtenção dos dados via interface JMX. A organização de classes e responsabilidades específicas de cada uma será explicada nas próximas seções.

Uma rodada de amostragem de 1 segundo foi usada, momento no qual todos os atributos de um determinado servidor JMX remoto (que pode conter vários nodos executando o NEeM) são lidos. Esse ciclo de obtenção de dados será apresentado em detalhes nos seções posteriores.

Esses dados que são lidos também são gravados num arquivo texto com valores separados por vírgula (*Comma Separated Value File*, ou arquivo CSV) para posterior análise. Apenas os últimos 60 segundos de dados, de todos os atributos, são mantidos por cada nodo, que contém uma lista por atributo para realizar este controle.

No momento de análise, esses dados são novamente importados na ferramenta e passados para um banco de dados SQlite[1], escolhido por simplicidade e economia de espaço de armazenamento. A exibição desses dados analisados (ou enquanto são obtidos) é feita com a biblioteca JFreeChart[2], um software livre que permite exibir dados em forma gráfica.

5.1 Modo de uso do protótipo

A figura 5.1 mostra a interface do protótipo assim que ele é iniciado. Duas abas principais existem: a de monitoramento (Live Monitoring) e a de análise (Console).

[1] http://www.sqlite.org/

[2] http://www.jfree.org/jfreechart/

5.1.1 Aba de monitoramento

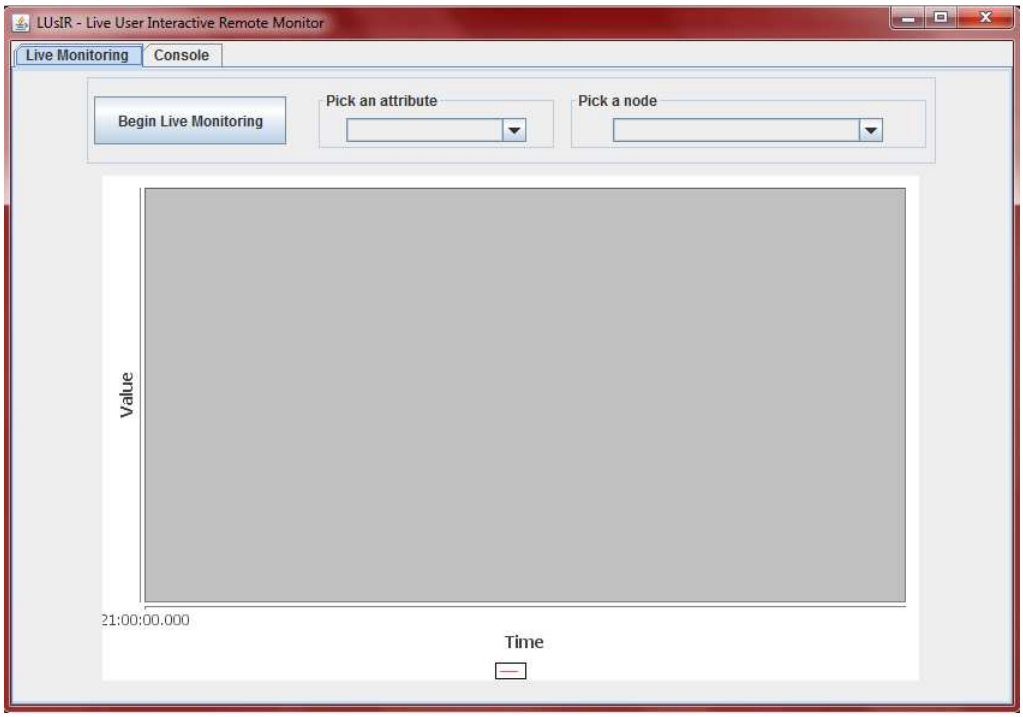


Figura 5.1: Interface básica da aba “Live Monitoring” no protótipo LUsIR

Essa primeira tela permite ao usuário começar uma nova sessão de monitoramento (ao clicar no botão “Begin Live Monitoring”). Cada sessão de monitoramento gera um novo arquivo CSV contendo todos os dados amostrados naquele período, de todos os nodos. Além disso, o usuário pode acompanhar os dados dinamicamente, conforme eles são obtidos, como pode ser visto na figura 5.2.

Assim que a sessão de monitoramento é iniciada, os combo-boxes “Pick an attribute” e “Pick a node” são populados com os atributos a serem monitorados e os nodos acessíveis, respectivamente. A lista de atributos passíveis de serem monitorados, já explicados no capítulo 3.5, é composta por:

MaxIds	BytesSent	PullReceived	BufferSize
Delivered	GossipFanout	ShufflePeriod	AcceptedSocks
Multicast	PushTimeToLive	OverlayFanout	ConnectedSocks
DataSent	MinPullSize	JoinRequests	PacketsReceived
HintsSent	PullPeriod	PurgedConnections	PacketsSent
PullSent	DataReceived	ShufflesReceived	BytesReceived
QueueSize	HintsReceived	ShufflesSent	TimeToLive

Tabela 5.1: Lista de atributos monitorados pelo protótipo LUsIR

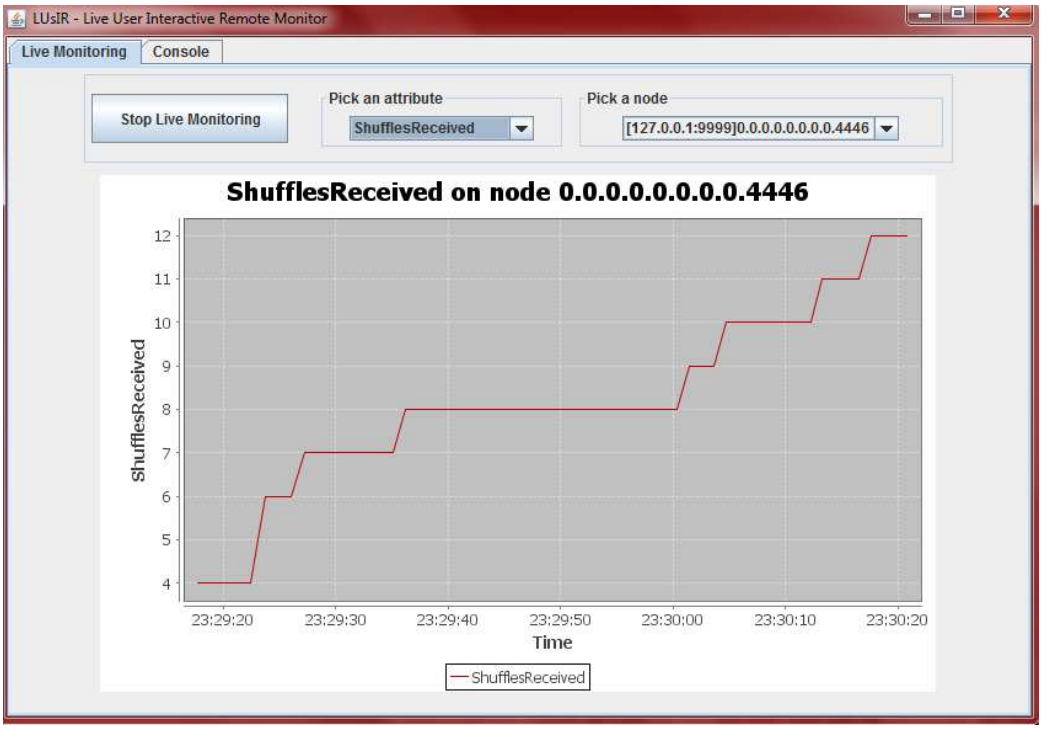


Figura 5.2: Início de uma sessão de monitoramento no protótipo LUsIR

Na figura acima, está sendo mostrado o atributo ShufflesReceived (ou seja, a quantidade acumulada de mensagens recebidas pelo canal de Shuffle do objeto Overlay) do nodo ativo na porta 4446. Usando os combo-boxes citados pode-se modificar esses parâmetros. Como limitação do protótipo, apenas um atributo, pertencente a um determinado nodo, pode ser visualizado por vez.

Ao clicar em “Stop Live Monitoring”, se encerra a sessão de monitoramento e os dados amostrados são salvos num arquivo CSV e numa instância de um banco de dados SQLite, para já serem explorados.

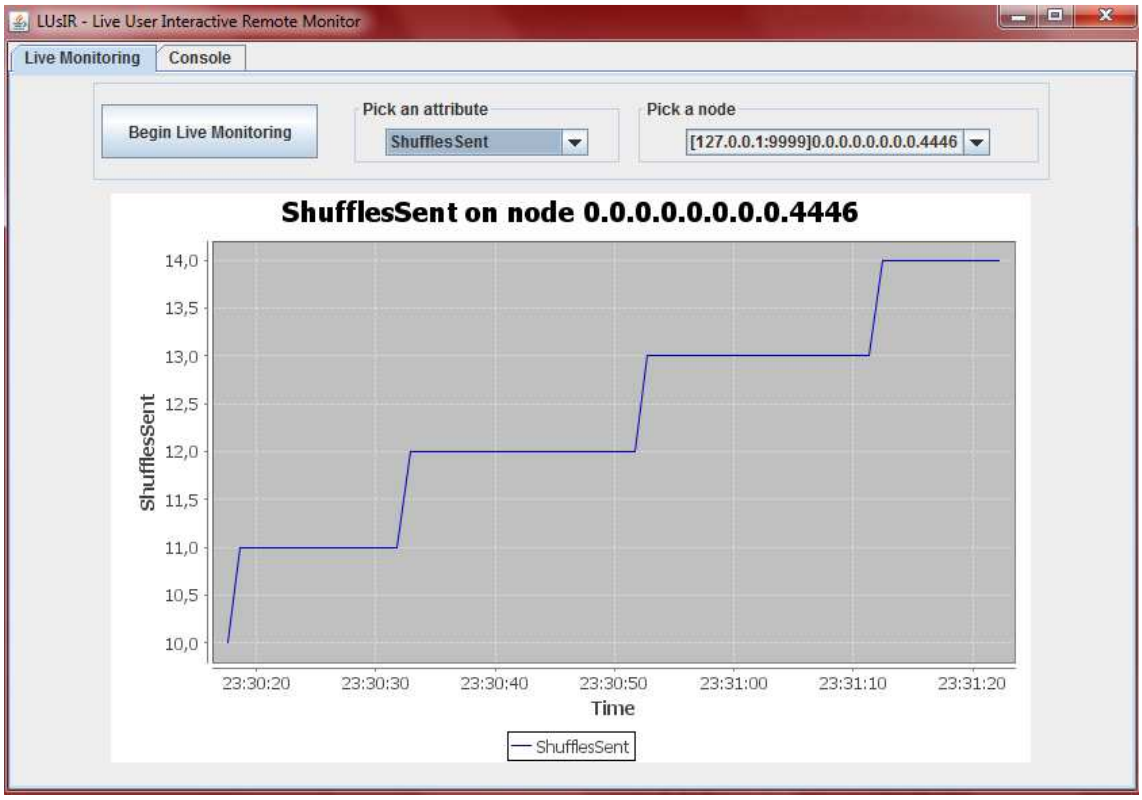


Figura 5.3: Final de uma sessão de monitoramento no protótipo LUsIR

Na tela com uma sessão de monitoramento encerrada, pode-se modificar os combo-boxes citados e visualizar os dados obtidos. Por exemplo, na figura 5.3, foi observado o gráfico gerado pelas amostras do atributo ShufflesSent.

Não por acaso, a cor da série mostrada na figura 5.3 é diferente daquela mostrada na figura 5.2. Conforme os combo-box tem seu valor modificado pelo usuário, e novos dados são mostrados em tela, o programa se encarrega de usar uma cor diferente da anterior para representar esses novos dados. Isso ajuda ao usuário perceber que houve uma modificação no conjunto de dados serem mostrados, mesmo que o valor ou a curva desses dados se mantenha igual.

5.1.2 Aba de análise

Caso nenhuma sessão de monitoramento tenha sido criada, a tela da figura 5.4 será mostrada e caberá ao usuário importar os dados CSV de sessões passadas através do botão “Import Data”.

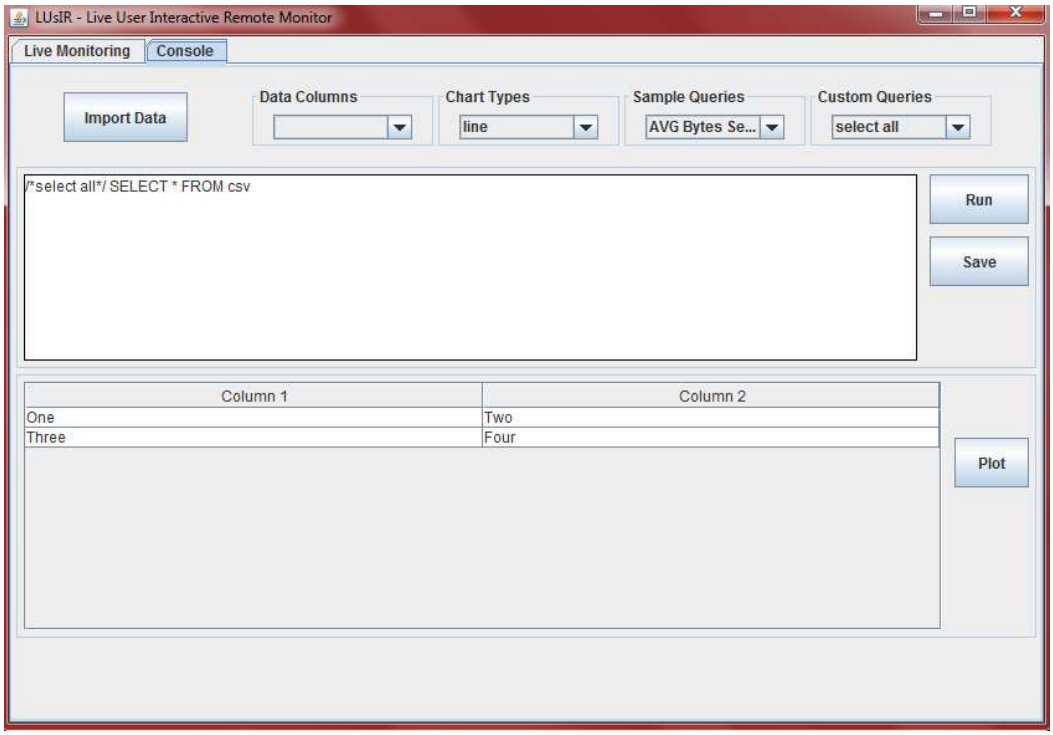


Figura 5.4: Interface sem dados importados na aba Console do protótipo LUsIR

Caso o usuário já tenha passado por uma sessão de monitoramento (ou importou corretamente os dados), a combo-box “Data Columns” será populado com as colunas representando os atributos monitorados durante a última sessão (ou importadas do CSV).

Apenas uma tabela é usada, contendo todos os dados do usuário. A query padrão exibe todos esses dados ao se clicar no botão de “Run”, como mostrado na figura 5.5.

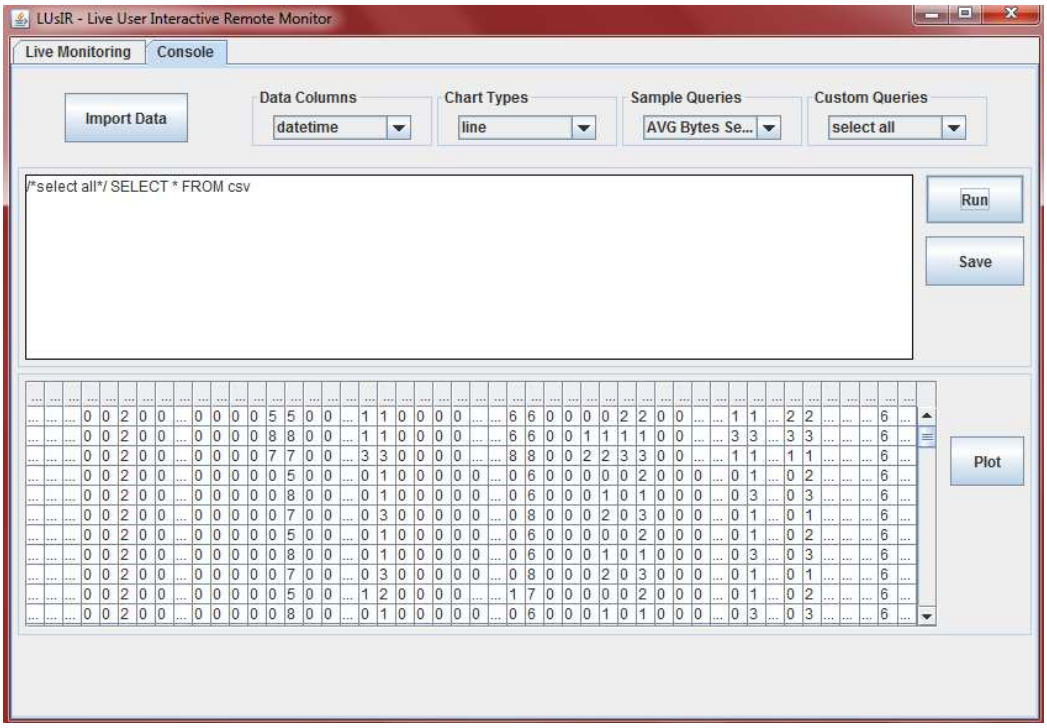


Figura 5.5: Execução da consulta de seleção padrão na aba Console do protótipo LUsIR

Este combo-box pode ser usado para explorar todas as colunas possíveis de serem usadas na criação de consultas SQLs.

Para que os dados sejam exibidos no formato gráfico, devem ser realizadas consultas que retornem dados num determinado formato. O usuário pode explorar as consultas de amostra listadas no combo-box “Sample Queries”. Ao selecionar um dos itens daquele combo-box, a consulta de amostra será mostrada no editor de consultas, como mostrado na figura 5.6.

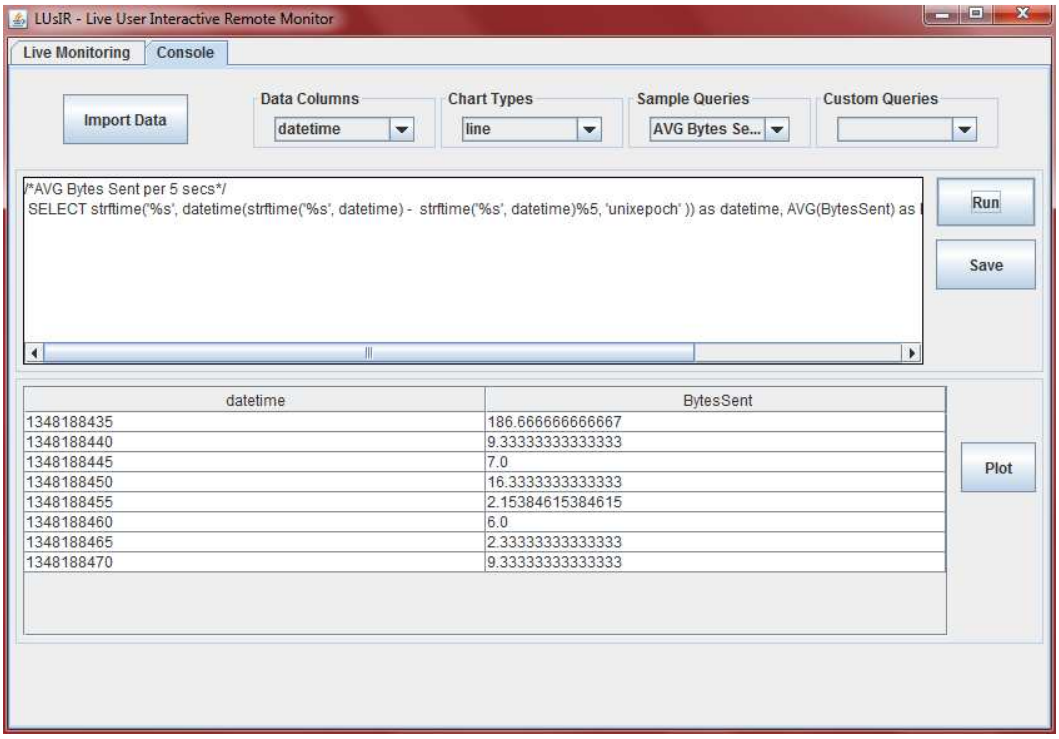


Figura 5.6: Execução de uma consulta de amostra na aba Console do protótipo LUsIR

Nota-se como a consulta da figura 5.6 fez com que duas colunas fossem retornadas como resultado: a primeira delas apresenta a data e hora que aquele dado foi obtido, e a segunda o valor desse dado. Ainda, entre comentários (/* e */) está uma string de texto que representa o nome dessa consulta, que será usado como título para o gráfico a ser gerado.

O formato de data e hora que deve ser utilizado é *unix epoch*, ou seja, a quantidade de segundos desde 1970.

A consulta fez com que os dados fossem retornados com 5s de diferença entre eles, e que cada valor representa a média de Bytes enviados nesses 5s.

Ao clicar no botão “Plot”, esses dados tabulares são convertidos para forma gráfica, usando o título da consulta previamente realizada, como pode ser visto na figura 5.7.

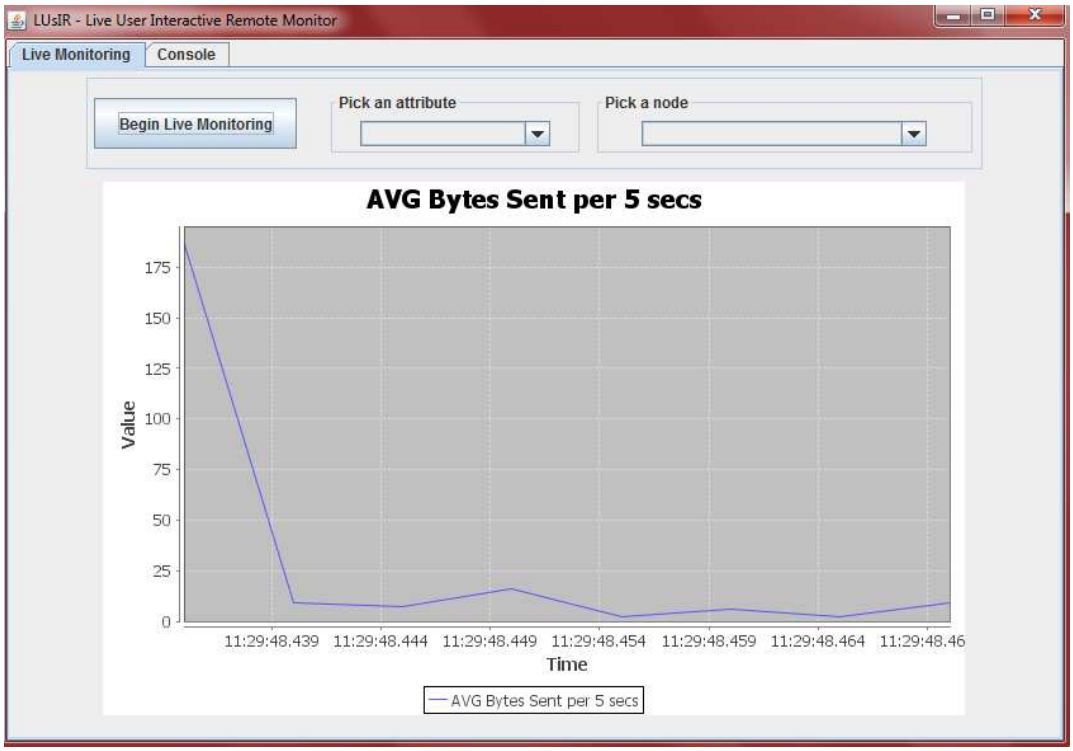


Figura 5.7: Plotando dados de uma consulta de amostra na aba Console do protótipo LUsIR

Mais de uma série temporal pode ser gerada. Para isso, o usuário deve usar duas ou mais consultas, como na figura 5.8, abaixo. Note que as consultas precisam, obrigatoriamente, serem finalizadas com o caracter “;”.

Um detalhe que se tornará visível nos próximos capítulos, conforme utilizamos o LUsIR, é que as séries plotadas mantém um certo padrão de cores. A primeira query da sequência sempre gera uma série com a cor vermelha; a segunda, com a cor azul; a terceira com a cor verde. Ou seja, a ordem em que as queries executadas aparecem afeta a cor que seu “resultado” terá. Isso nos ajuda, como pode ser notado no capítulo 6.1, a gerar gráficos medindo o mesmo atributo em diversos nodos. Como a primeira query retorna dados do primeiro nodo, e assim sucessivamente, conseguimos manter uma padronização interessante na coloração dos resultados.

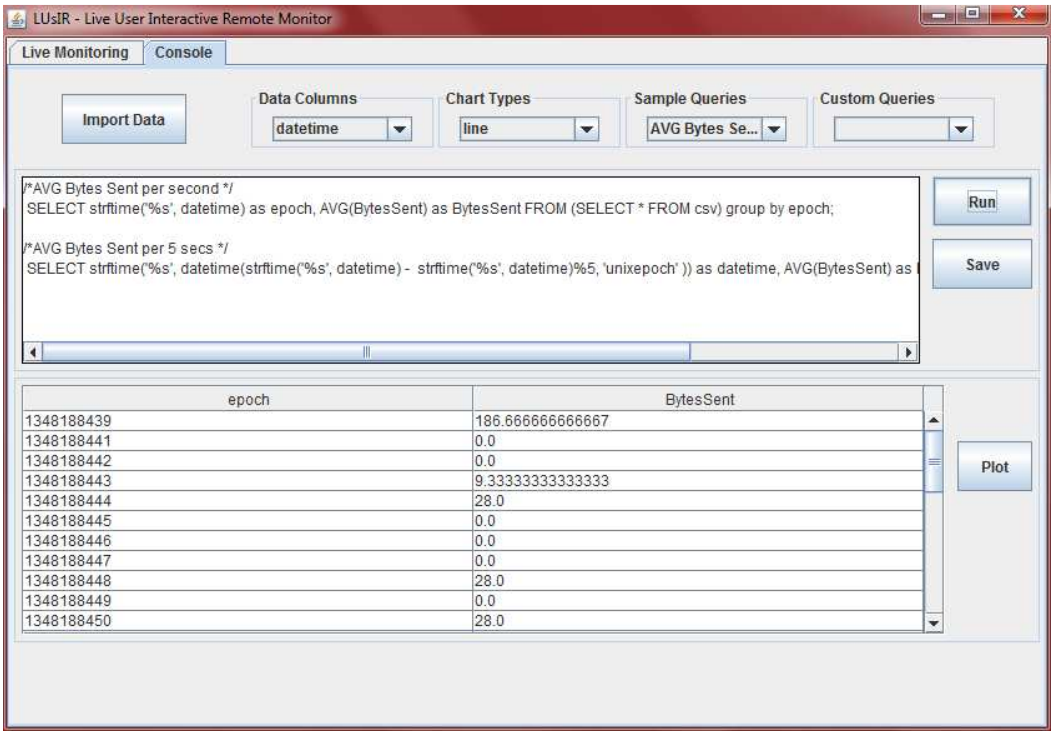


Figura 5.8: Execução de duas consultas de amostra na aba Console do protótipo LUsIR

Figura 5.9: Plotando dados de duas consultas de amostra na aba Console do protótipo LUsIR

Ainda, o usuário pode escolher executar somente uma parte de uma consulta no seu editor. Para isso, ele deve selecionar a parte que deseja executar e apertar o botão de “Run”.

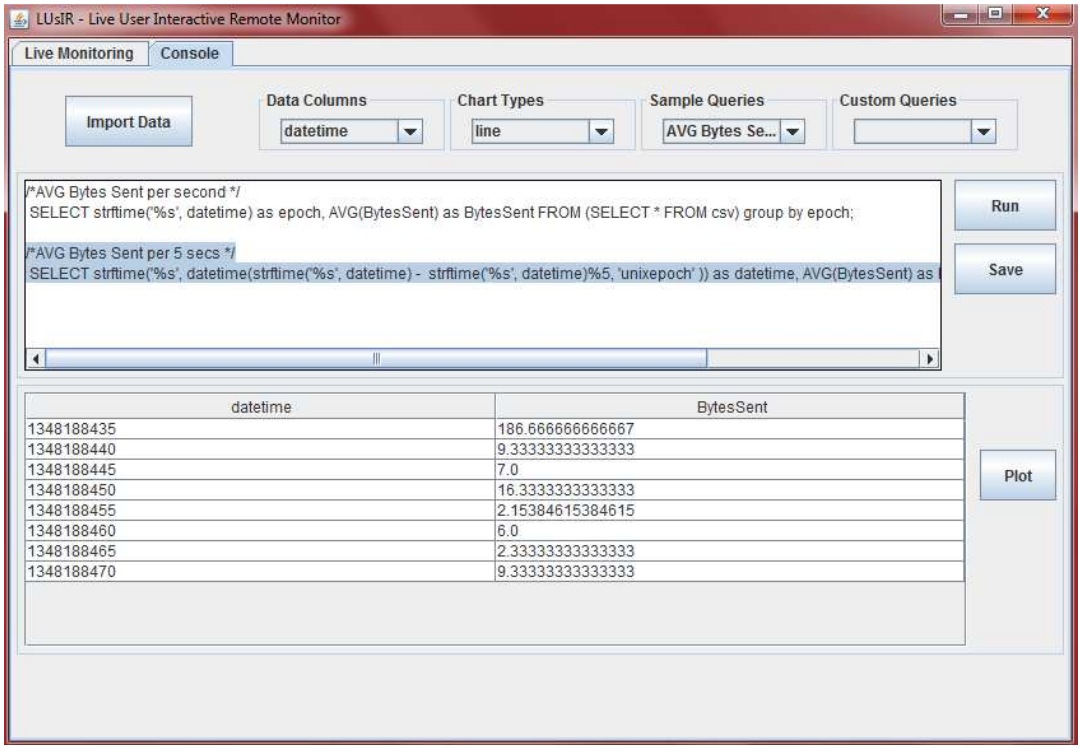


Figura 5.10: Execução de parte do conteúdo do editor de consultas na aba Console do protótipo LUsIR

Por fim, o usuário pode escolher salvar o conteúdo do editor de consultas. O título da primeira consulta do editor será usado para identificar todo o seu conteúdo, mesmo que existam várias outras consultas no editor. O conteúdo salvo será acessível através do combo-box de “Custom Queries” e poderá ser usado mesmo que o protótipo seja fechado e re-aberto.

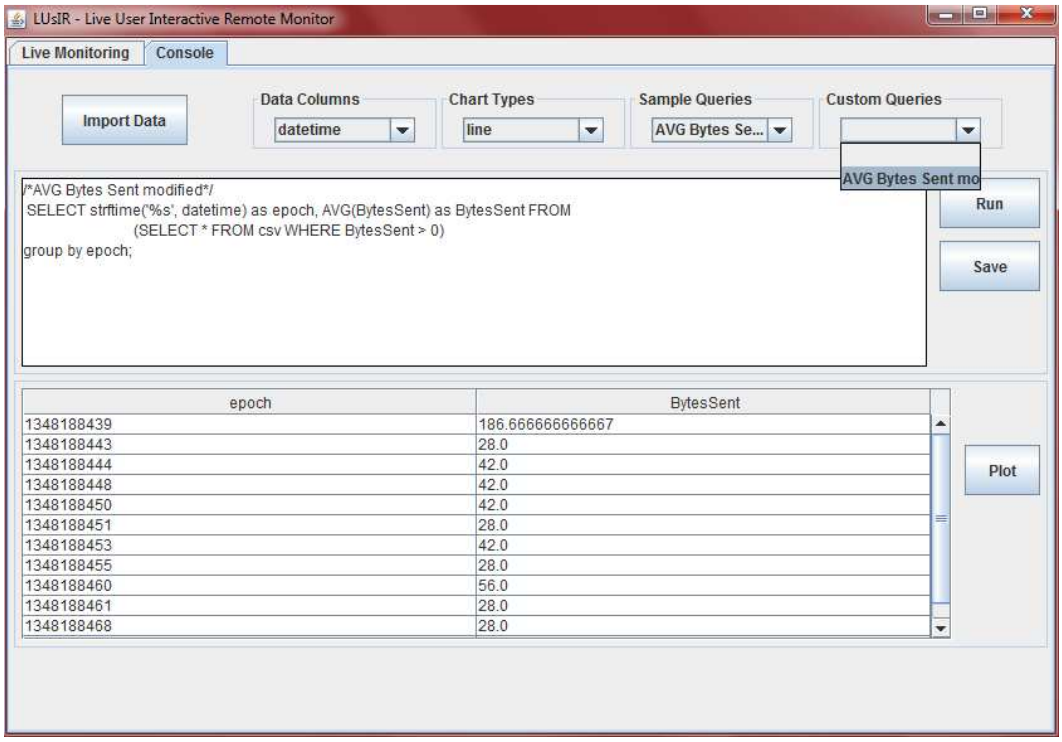


Figura 5.11: Salvando o conteúdo do editor de consultas na aba Console do protótipo LUsIR

5.2 Descrição das classes do protótipo

A figura 5.12, abaixo, mostra a organização de classes do protótipo e como elas se relacionam e alguns atributos importantes, destacados.

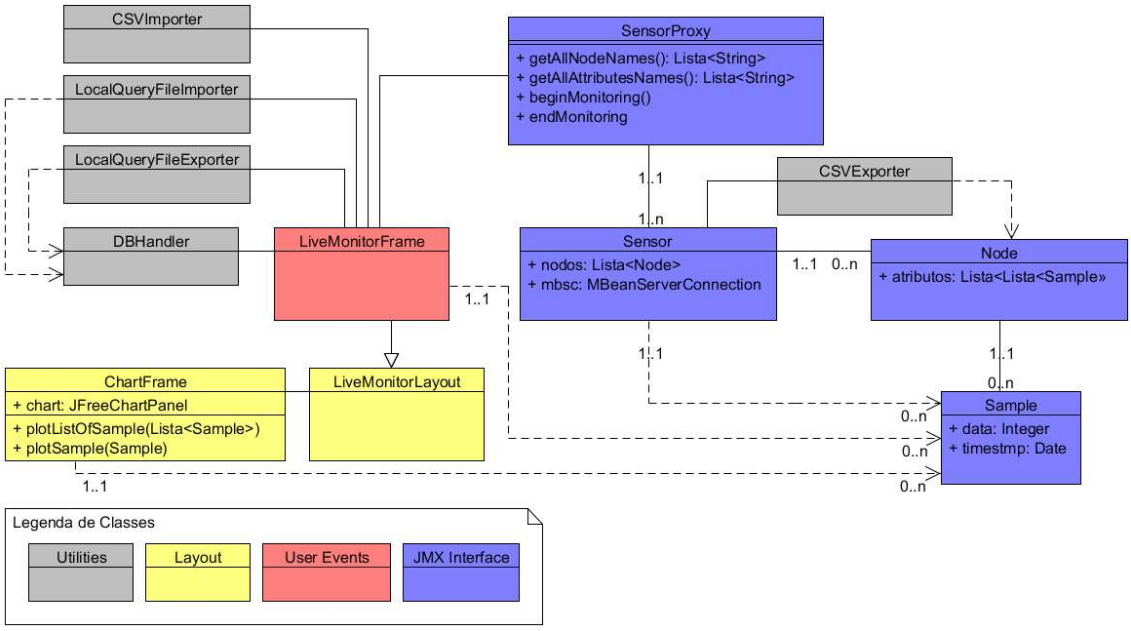


Figura 5.12: Relação entre classes do protótipo LUsIR

5.2.1 Classes de Layout

A essas classes cabe controlar como os elementos de tela são exibidos, em termo de posição de layout.

A classe ChartFrame controla o acesso ao elemento que cria gráficos em linha da biblioteca JFreeChart. A ela, também, cabe receber um objeto da classe Sample e inserir seus dados numa série temporal já criada ou receber uma lista de objetos Sample e montar uma série temporal com eles. Outros tipos de gráficos poderiam ser usados, e caberia a essa classe usar o correspondente correto na biblioteca citada.

A classe LiveMonitorLayout cuida do posicionamento de todos os elementos de tela. Como foram usados componentes Swing e o gerenciador de layout GridBagLayout, cabe a essa classe cuidar dos detalhes que precisam ser considerados ao desenvolver interfaces gráficas na linguagem Java. Caso essas tecnologias de layout decidam ser modificadas, apenas esta classe será afetada.

5.2.2 Classe de Eventos do Usuário

Todos os eventos necessários aos componentes da classe LiveMonitorLayout são implementados pela classe-filha LiveMonitorFrame. Essa é a classe principal do protótipo LUsIR.

Cabe a essa classe controlar como esses eventos afetam os dados e os componentes de tela. Quando uma sessão de monitoramento é iniciada pelo usuário, essa classe se responsabiliza por, a cada 1 segundo, consultar o objeto SensorProxy em busca de novos dados, a serem mostrados dinamicamente, a fim de deixar o usuário sabendo o que está sendo efetivamente monitorado. Os detalhes dessa interação podem ser vistos na figura 5.13. Essa consulta, porém, não pode ser demorada, já que todos os outros eventos de tela tem de ser atendidos também pela classe LiveMonitorFrame.

Através dos métodos beginMonitoring e endMonitoring do SensorProxy é feito o controle, por parte da classe LiveMonitorFrame, do início e fim da sessão de monitoramento. Assim que uma sessão inicia, os combo-boxes “Pick an attribute” e “Pick a node” são populados dinamicamente pela classe LiveMonitorFrame com dados obtidos a partir dos métodos e getAllAttributesNames e getAllNodeNames, respectivamente, ambos do objeto SensorProxy.

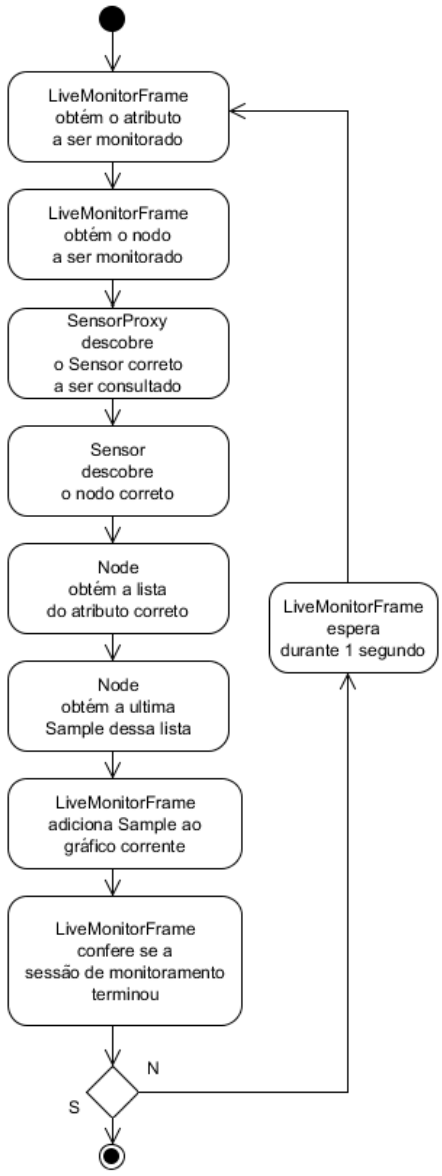


Figura 5.13: Ciclo de funcionamento durante uma sessão de monitoramento

Por fim, essa classe faz uso de classes utilitárias para:

- Carregar um arquivo CSV, gerado por uma sessão de monitoramento já terminada;
- Executar as consultas SQL do usuário a esses dados;
- Carregar as “Sample Queries” e “Custom Queries” do usuário a partir dos arquivos samples.sql e custom.sql, respectivamente;
- Salvar novas “Custom Queries”;

5.2.3 Classes de Interface JMX

A essas classes cabe se comunicar com o servidor de nomes remoto, para obter as referências aos nodos executando o protocolo NEeM, monitorar os dados disponíveis,

salvar em um arquivo de log cada amostra obtida e disponibilizar esses dados ao usuário, que deseja visualizá-los dinamicamente.

O objeto SensorProxy gerencia todos os objetos Sensor e se encarrega de criar uma interface de acesso unificada a eles, afim de facilitar o uso de suas informações pela classe LiveMonitorFrame. A cada início de sessão de monitoramento, esse objeto tenta se conectar com 5 (cinco) diferentes endereços de servidores MBeans: a partir de 127.0.0.1:9995 até 127.0.0.1:9999.

Para cada conexão bem sucedida é criado um objeto Sensor, que se conecta com um conector RMI para ter acesso a interface JMX disponibilizada por um Servidor MBeans do Neem 0.7

Um objeto da classe Sensor possui uma linha de execução (*thread*) própria, separada daquela responsável por controlar a interface com o usuário. A essa *thread* cabe amostrar os dados de todos os nodos disponíveis, seguindo o fluxograma da figura 5.14, abaixo. Para que isso seja possível, o objeto da classe Sensor possui uma lista de objetos do tipo Node, cada um representando um determinado nodo executando o protocolo NEeM. Cada dado amostrado carrega um marcador de tempo (*timestamp*) e o dado numérico lido.

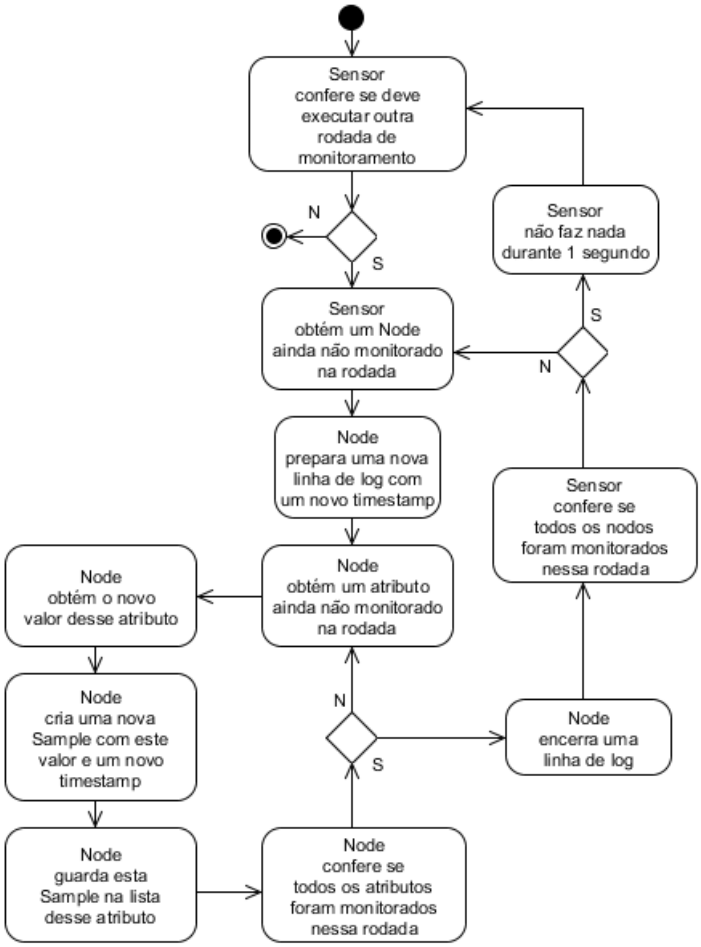


Figura 5.14: Ciclo de funcionamento de uma instância da classe Sensor

Por simplicidade, na figura 5.14 foi mostrado apenas o fluxo de ações, dentro da sessão de monitoramento, para uma instância da classe sensor. Note como cada objeto Node é responsável por escrever sua lista de objetos Sample em um arquivo de log.

A classe utilitária CSVExporter é responsável por auxiliar nessa tarefa. Como apenas uma instância desta classe é utilizada, por todos os objetos Node residentes num objeto Sensor, todos escrevam no mesmo arquivo de texto no formato CSV, nomeado de "<sensor-host-name>.log.csv". Por exemplo, um objeto Node pertencente a uma instância da classe Sensor, operando na porta 9998 da máquina local, escreveria suas informações sobre amostras no arquivo "127.0.0.1.9998.log.csv".

A classe LiveMonitorFrame, que controla as interações com o usuário, é responsável por controlar o início e término das rodadas de monitoramento realizadas por essa *thread* do objeto da classe Sensor. Ainda, ela solicita ao objeto da classe Sensor a lista de objetos (do tipo Sample) de um determinado objeto Node a ser plotada, por solicitação do usuário (como mostrado na figura 5.3). É mantido em memória apenas as 60 (sessenta) mais recentes amostras de cada atributo, em cada nodo, o suficiente para que o usuário visualize o último minuto inteiro de amostragem (já que cada rodada de amostragem é realizada em 1 segundo).

5.2.4 Classes Utilitárias

Por fim, às classes utilitárias cabe fazer a interface necessária para ler e escrever as queries do usuário e de amostra para arquivos de texto, ler e escrever em arquivos de log no formato CSV e de facilitar a interação, do usuário e dos objetos utilitários, com o banco de dados SQLite interno.

Os objetos Node, vistos anteriormente, usam a classe CSVExporter para salvar os dados monitorados em um arquivo de log. Cabe a essa definir o formato de cada linha de log, o cabeçalho desse arquivo e o marcador de tempo (*timestamp*) a ser utilizado para representar o conjunto de amostras obtidas por aquele nodo num determinado instante de tempo.

Similarmente, cabe a classe CSVImporter a responsabilidade de ler um arquivo em formato CSV e importa-lo no banco de dados interno da aplicação. Com isso, os dados de uma sessão de monitoramento previamente realizada podem ser livremente explorados pelo usuário. Mais de um log pode ser importado, no mesmo banco de dados. Seus resultados são adicionados a mesma tabela interna na aplicação, como se os logs tivessem sido consolidados em um único arquivo.

A exploração dos dados é feita através de consultas ao banco de dados. Quando o programa é iniciado, as consultas de amostra, são lidas a partir de um arquivo chamado *samples.sql*. As que o usuário previamente salvou são lidas a partir de um arquivo chamado *customs.sql*. A importação do conteúdo desses arquivos no protótipo cabe a classe LocalQueryFileImporter. Ao salvar uma nova consulta customizada, o protótipo faz uso da classe LocalQueryFileExporter.

A classe DBHandler gerencia o banco de dados, que contém toda a informação que deve ser persistida em memória afim de ser acessível via comandos SQL. Duas tabelas são utilizadas:

- *csv*: contendo os dados gerados na sessão de monitoramento importada pelo usuário. Esse arquivo possui tantas colunas quanto entradas no cabeçalho do CSV importado previamente.
- *user_queries*: contendo as consultas, tanto as de amostra quanto as customizadas pelo usuário. Possui três colunas
    - *name*: coluna do tipo TEXT contendo o título dessa consulta;
    - *type*: coluna do tipo TEXT que carrega o tipo dessa consulta. Os valores possíveis são "*custom*" ou "*samples*";
    - *query*: coluna do tipo TEXT que possui a consulta SQL propriamente dita

O usuário tem total controle o banco de dados durante uma sessão de análise. Ele pode criar e destruir tabelas, por exemplo. Porém, como nenhum meio de persistência dessas configurações foi pensado, assim que o programa for fechado essas modificações serão perdidas. Um novo banco de dados, marcado com um marcador de tempo, será gerado quando o protótipo for novamente reaberto.

5.3 Considerações sobre a implementação do protótipo LUsIR

O protótipo LUsIR apresentado anteriormente agrega as funções de monitoramento e de análise, como apresentadas no projeto apresentado no capitulo 4. O próximo capítulo introduz alguns experimentos realizados com esse protótipo, seja durante seu desenvolvimento (para validar se as modificações realizadas não impactaram nenhuma funcionalidade principal) ou após o término dele.

Diferentemente do sugerido pelo projeto, o protótipo apenas deixa o usuário visualizar um item por vez. Ainda, ele somente possibilita a criação de gráficos em linha.

Porém, certas facilidades não previstas pelo projeto do LUsIR tiveram de ser implementadas, seja para facilitar o uso da aplicação ou para possibilitar que algum experimento pudesse ser realizado da melhor forma. Dentre elas, nota-se:

- O padrão de cores é mantido o mesmo e controlado pela ordem das consultas executadas pelo usuário;
- Mais de um arquivo de log é gerado (um para cada servidor MBean encontrado);
- Mais de um arquivo de log pode ser importado, na mesma sessão de análise;
- O usuário pode salvar suas queries para posterior uso.

6 EXPERIMENTOS E RESULTADOS PRÁTICOS

Para demonstrar o funcionamento e a utilidade do protótipo desenvolvido nesse trabalho, foram feitos testes com programas de exemplo disponibilizados juntamente do código do NEeM 0.7.

Durante o desenvolvimento do protótipo LUsIR, uma versão ligeiramente modificada da classe Crowd foi utilizada sempre que se desejava estimular o módulo de análise. A facilidade desta classe é que ela cria diversos nodos rodando o NEeM, mas conectados a um mesmo servidor MBeans – e, portanto, possibilitando ao programa só se preocupar em criar uma conexão RMI.

Após o desenvolvimento ter sido concluído, a classe Chat foi utilizada para testar efetivamente a solução desenvolvida. Essa classe forçou o protótipo a se conectar a vários servidores MBeans diferentes. Cada "cliente" de chat (um nodo do NEeM) forçava uma nova JVM a ser inicializada. Portanto, múltiplas conexões RMI precisavam ser levadas em conta.

6.1 Testes durante o Desenvolvimento

Durante o desenvolvimento do protótipo LUsIR, foi utilizado um pacote Java contendo o NEeM 0.7 e a classe Crowd. De acordo com o comentário no cabeçalho dessa classe de amostra ela "cria uma multidão de nodos NEeM. Todas as mensagens recebidas são descartadas. Nenhuma mensagem é enviada.".

A "multidão" criada por esse teste é de apenas 3 (três) nodos do NEeM, que serão identificados daqui em diante pela porta de saída pela qual eles realizam sua comunicação: 4444, 4445 e 4446. Todos os nodos se conectam ao mesmo servidor MBeans, o que nos dá a facilidade de apenas precisar se conectar com um endereço IP de somente um conector RMI.

6.1.1 Visão geral

A aplicação, como descrito, não realiza multicast de mensagem alguma. O único tráfego observável se deve ao criado pela classe Overlay (ver capítulo 2.3.4.2) para modificar a vizinhança de um nodo periodicamente. Esse comportamento é explicitado pela tabela 6.1, que descreve como cada atributo se comportou no decorrer do tempo.

Atributos	Comportamento
MaxIds	**Constante (valor 100)**
Delivered	**Constante (valor 0)**
Multicast	**Constante (valor 0)**
DataSent	**Constante (valor 0)**
HintsSent	**Constante (valor 0)**
PullSent	**Constante (valor 0)**
QueueSize	**Constante (valor 10)**
BytesSent	**Crescente**
GossipFanout	**Constante (valor 11)**
PushTimeToLive	**Constante (valor 2)**
MinPullSize	**Constante (valor 64)**
PullPeriod	**Constante (valor 120)**
DataReceived	**Constante (valor 0)**
HintsReceived	**Constante (valor 0)**
PullReceived	**Constante (valor 0)**
ShufflePeriod	**Constante (valor 10)**
OverlayFanout	**Constante (valor 15)**
JoinRequests	**Constante (dependente do nodo)**
PurgedConnections	**Constante (valor 0)**
ShufflesReceived	**Crescente**
ShufflesSent	**Crescente**
BufferSize	**Constante (valor 1024)**
AcceptedSocks	**Crescente**
ConnectedSocks	**Crescente**
PacketsReceived	**Crescente**
PacketsSent	**Crescente**
BytesReceived	**Crescente**
TimeToLive	**Constante (valor 6)**

Tabela 6.1: Comportamento de cada atributo monitorado durante experimento de Crowd

Para os atributos cujos valores se modificaram durante o experimento, seguem os gráficos por atributo na figura 6.1. Cada série representa um nodo, demonstrando essa progressão de valores no tempo.

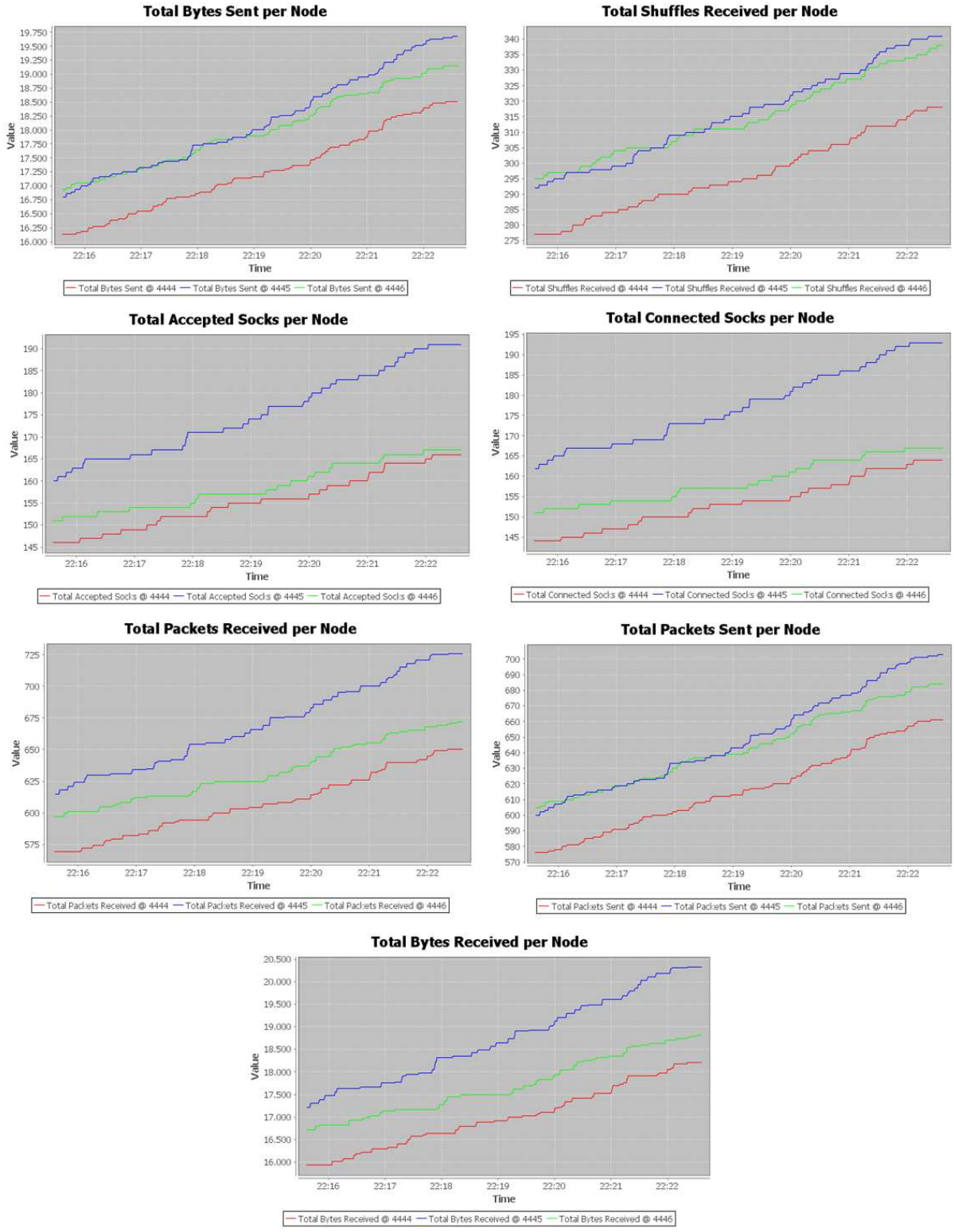


Figura 6.1: Gráficos por atributo monitorado durante experimento de Crowd

Um dos conjuntos de consultas utilizadas para gerar esses gráficos pode ser vista na figura 6.2. Note como as consultas utilizam atributos com prefixo “Total”, nos quais, segundo a seção 4.1, apresentam somente os valores totais de cada atributo. Três consultas precisaram ser feitas, já que cada uma retorna os dados de um nodo

específico. Todas as três são plotadas no mesmo gráfico, gerando uma das 7 imagens vistas anteriormente (no caso, a primeira, mais a esquerda).

```
/*Total Bytes Sent */
SELECT strftime('%s', datetime), TotalBytesSent FROM csv
WHERE substr(hostname, length(hostname)-3) = '4444';
/* per */
SELECT strftime('%s', datetime), TotalBytesSent FROM csv
WHERE substr(hostname, length(hostname)-3) = '4445';
/* Node*/
SELECT strftime('%s', datetime), TotalBytesSent FROM csv
WHERE substr(hostname, length(hostname)-3) = '4446'
```

Figura 6.2: Exemplo de consulta utilizada para gerar um dos gráficos analisados após experimento de Crowd

6.1.2 Análise dos dados mais primitivos

Como todos os gráficos apresentam um comportamento crescente, algumas perguntas que podem ser feitas são:

- Por onde começar a explorar os dados?
- Quando acontece cada incremento?
- Qual a quantidade deste incremento?
- Existe alguma relação deste incremento com alguma outra informação?

A resposta à primeira pergunta seria a definição de uma abordagem de exploração inicial. Um bom ponto de partida são os dados mais primitivos que temos, ou seja, a quantidade de bytes enviada e recebida pelo nodo ou conjunto de nodos. Com esses dados, conseguiremos entender o padrão de troca de dados da aplicação e criar melhores hipóteses sobre “de onde estão vindo esses dados”.

Dessa forma, a segunda pergunta pode ser respondida pelo gráfico 6.3, que consulta o atributo especial sem o prefixo “Total”. Esses atributos especiais foram explicados anteriormente na seção 4.1, a qual descreve a especificação para o módulo de monitoramento.

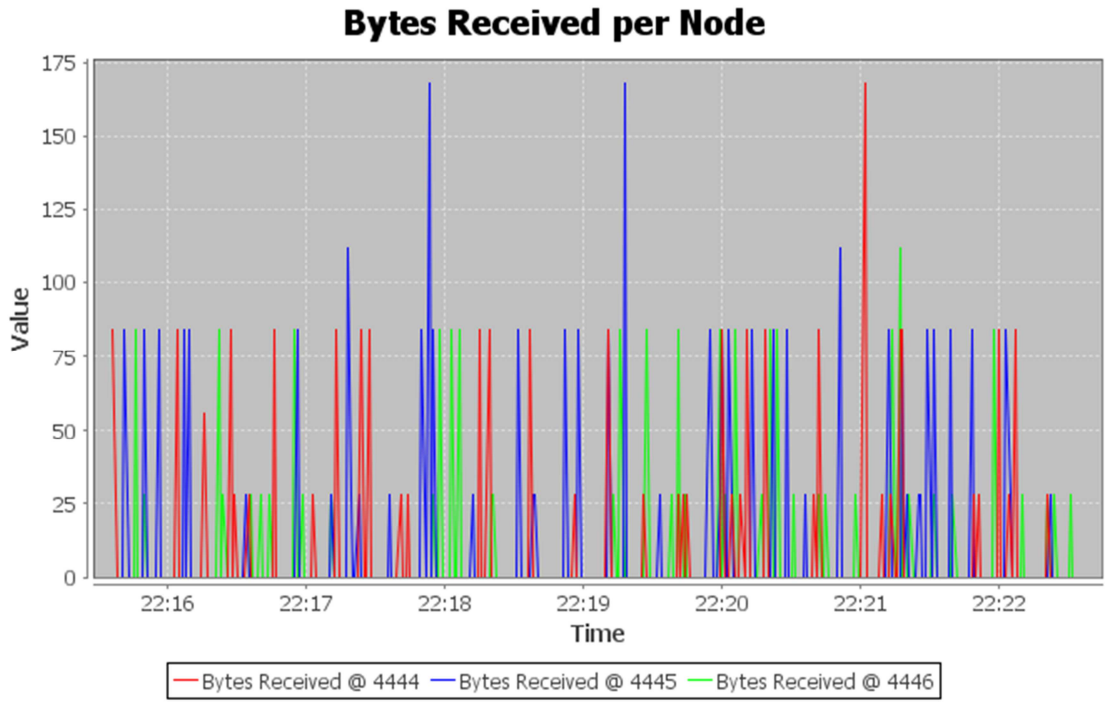


Figura 6.3: Gráfico demonstrando o atributo “Bytes Recebidos por nodo” ao final de experimento de Crowd

Segundo a figura 6.3, cerca de 75 bytes são recebidos a cada “incremento” nos Bytes Recebidos. Às vezes apenas 25 são recebidos, e noutras 125 ou até 175 são lidos ao mesmo tempo.

A terceira pergunta pode ser respondida somente ampliando uma faixa de tempo específica. A figura 6.4 expande o primeiro minuto do experimento.

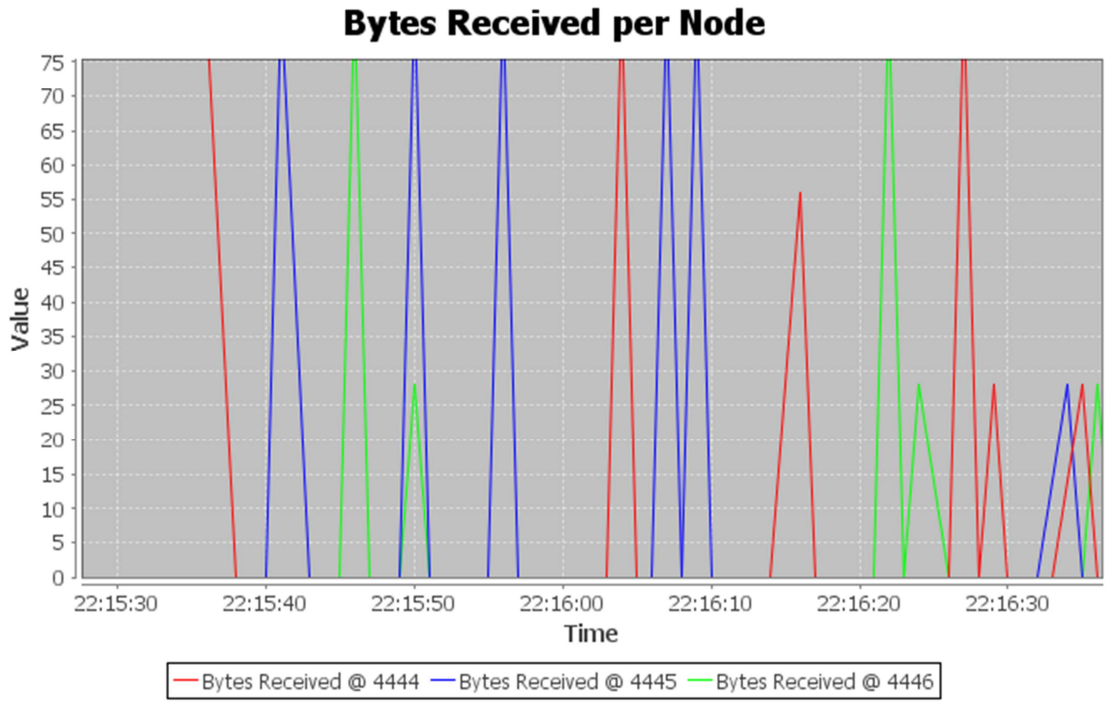


Figura 6.4: Gráfico demonstrando o primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” ao final de experimento de Crowd

Pela figura 6.4, acima, o nodo 4444 recebeu dados perto de 22:15:35, perto de 22:16:05 e perto de 22:16:15, o que nos dá um período entre recebimento de dados entre 15s e 30s. Isso não é muito preciso, já que essa análise acaba tendo de ser feita pelo olho humano.

Podemos voltar ao módulo de análise e começar a gerar consultas que nos ajudem a ver os dados de uma maneira mais analítica. O método de análise visual já nos ajudou, a detectar áreas de interesse.

A figura 6.5 possui uma consulta que evidencia os dados que queremos. A consulta exibe a hora em que cada amostra foi gerada e os bytes recebidos naquele momento. Porém, nós só queremos os momentos de tempo em que algum dado foi recebido (primeira cláusula depois do trecho “WHERE 1=1”) e somente oriundo de um nodo específico (segunda cláusula depois do “WHERE 1=1”). Por fim, as duas últimas cláusulas restringem o tempo analisado para o primeiro minuto de experimento.

```
SELECT time(datetime) as time, BytesReceived FROM csv
WHERE 1=1
AND BytesReceived > 0
AND substr(hostname,length(hostname)-3) = '4444'
AND datetime(datetime) > datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch')
AND datetime(datetime) <= datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch', '+60 seconds')
```

time	BytesReceived
22:15:36	84
22:16:04	84
22:16:16	56
22:16:27	84
22:16:29	28
22:16:35	28

Figura 6.5: Consulta para filtrar os dados do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo”

Agora sabemos que os dados começaram a ser recebidos por este nodo no tempo 22:15:36 e foram sendo recebidos nos tempos relativos de 28 segundos, 12 segundos, 11 segundos, 2 segundos e 6 segundos.

Para entender melhor esse período, podemos olhar para os bytes enviados no mesmo período, como na figura 6.6:

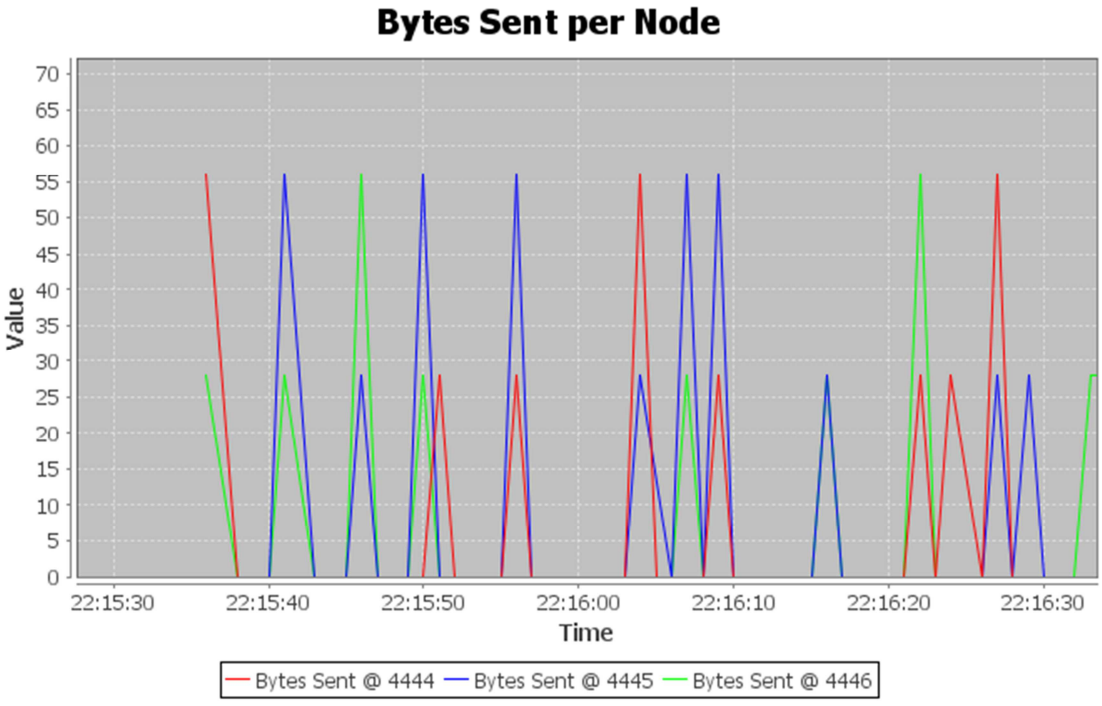


Figura 6.6: Gráfico demonstrando o primeiro minuto de amostras do atributo “Bytes Enviados por nodo” ao final de experimento de Crowd

Note que parece haver muito mais eventos de envio que de recebimento. Notem, também, que a média de bytes enviados fica em cerca de 30 bytes por vez, nunca menos que isso. De forma similar a análise dos bytes recebidos, os dados sobre bytes enviados em formato tabular melhoram essa aproximação, como mostra a figura 6.7.

```
SELECT time(datetime) as time, BytesSent FROM csv
WHERE 1=1
AND BytesSent > 0
AND substr(hostname,length(hostname)-3) = '4444'
AND datetime(datetime) > datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch', '+0 seconds')
AND datetime(datetime) <= datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch', '+60 seconds')
```

time	BytesSent
22:15:36	56
22:15:51	28
22:15:56	28
22:16:04	56
22:16:09	28
22:16:22	28
22:16:24	28
22:16:27	56

Figura 6.7: Consulta para filtrar os dados do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo”

Note que, coincidentemente, os momentos de envio de bytes parecem coincidir com os de recebimento de bytes. Parece também coincidir a quantidade de bytes nesses momentos: quando o envio é de grande quantidade, o recebimento também. Uma análise gráfica pode deixar essas semelhanças mais claras.

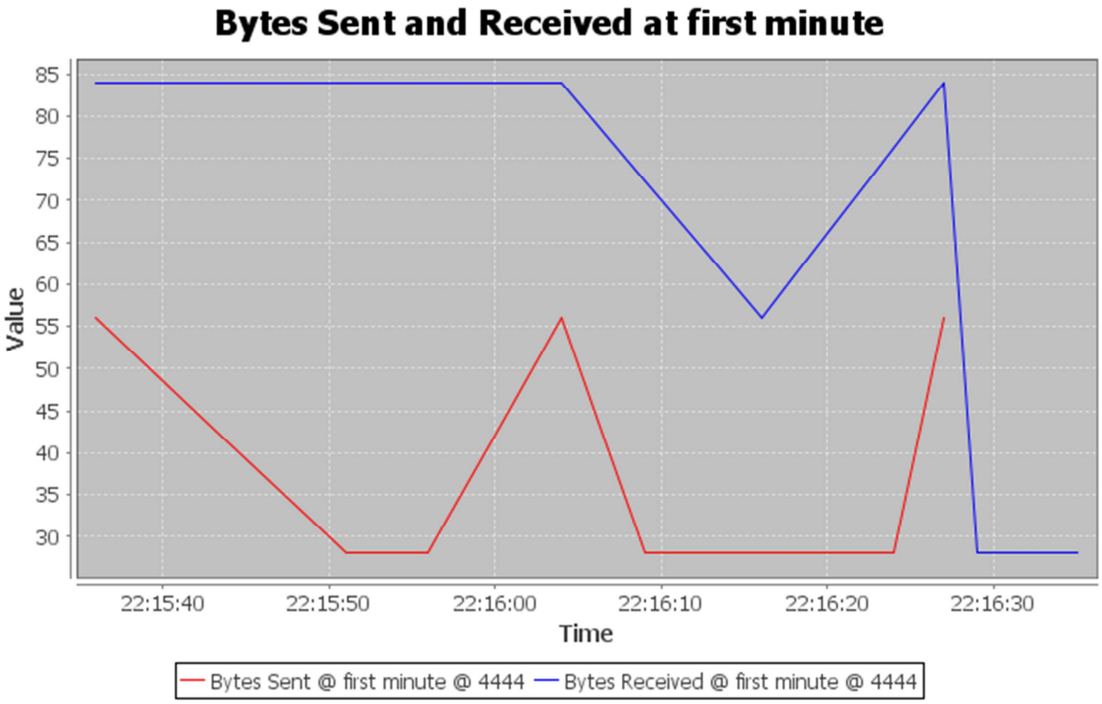


Figura 6.8: Gráfico demonstrando uma associação das amostras do primeiro minuto de amostras do atributo “Bytes Recebidos por nodo” com “Bytes Enviados por nodo”

Como visto na figura 6.8, não parece haver uma associação muito clara entre os eventos de recebimento e envio de bytes. A figura 6.9 tenta melhorar essa impressão expandindo o momento de experimento sendo analisado para 5 minutos:

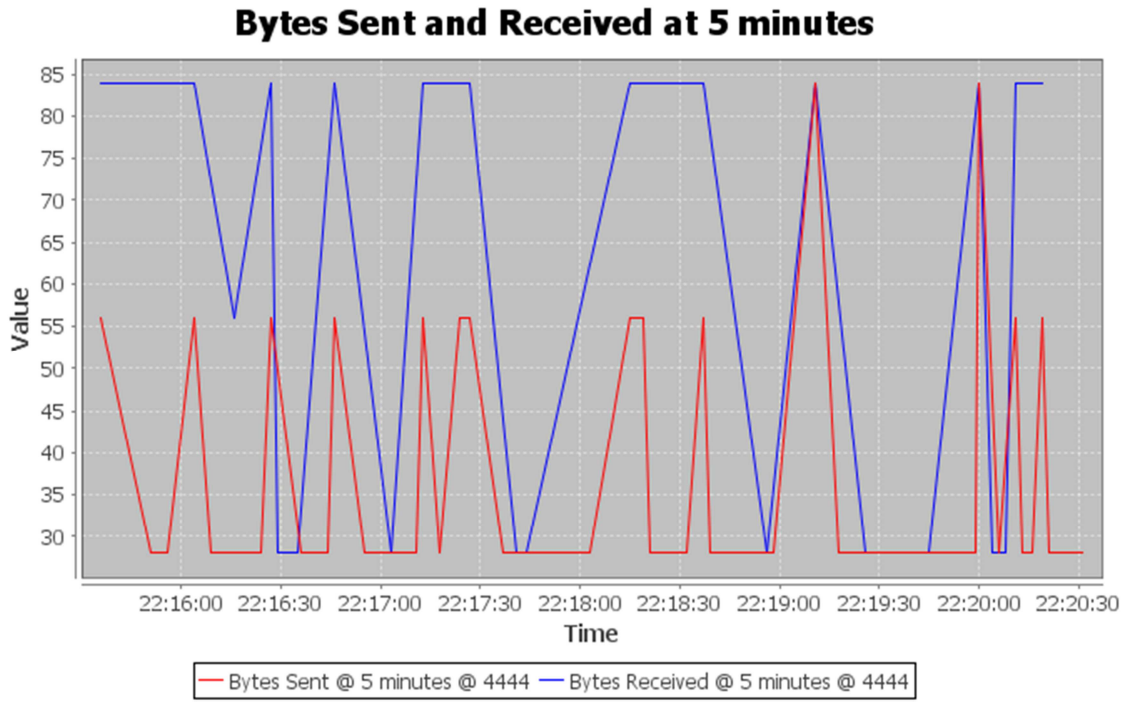


Figura 6.9: Gráfico demonstrando uma associação das amostras dos primeiros cinco minutos de amostras do atributo “Bytes Recebidos por nodo” com “Bytes Enviados por nodo”

A figura acima evidencia melhor uma possível relação entre bytes enviados e recebidos, mais especificamente nos momentos por volta de 22:16:45 e 22:17:15. Ainda, nos momentos por volta de 22:19:15 e 22:20 parece haver uma relação bem direta entre esses eventos.

6.1.3 Análise semanticamente mais rica

Para tentar melhorar nosso entendimento e responder a quarta pergunta da seção anterior, talvez faça mais sentido analisar eventos semanticamente mais ricos e tentar relacioná-los com o tráfego de bytes. Um exemplo é o evento de envio de mensagem de Shuffle (quando um nodo dissemina um vizinho para outro nodo conhecê-lo) com o tráfego de bytes enviados, como mostra a figura 6.10.

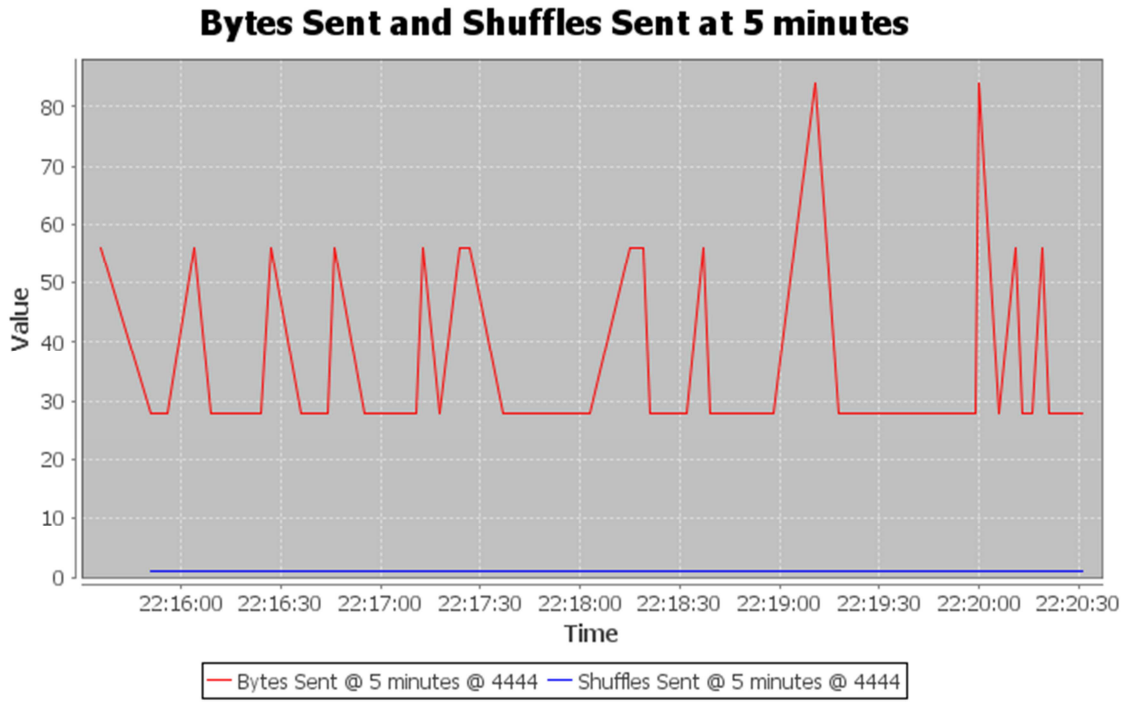


Figura 6.10: Gráfico demonstrando uma associação das amostras dos primeiros cinco minutos de amostras do atributo “Shuffles Enviados por nodo” com “Bytes Enviados por nodo”

A quantidade de eventos de envio Shuffle é uma medida diferente do que vínhamos usando (quantidade de bytes enviados). Portanto, existe a necessidade de se modificar ligeiramente os dados a fim de torná-los melhor comparáveis.

Como o acréscimo de eventos de Shuffle acontece sempre em 1 unidade, podemos voltar a representar graficamente os momentos em que nenhum evento ocorreu, o que elimina a 2ª cláusula após o termo “WHERE 1=1” das consultas que vínhamos realizando, como pode ser visto na figura 6.11. Podemos, ainda, fazer com que esses acréscimos de 1 unidade sejam representáveis em forma gráfica como se fosse uma quantidade bem maior (como, por exemplo, 10 unidades), afim de dar mais destaque a eles. A figura 6.11 mostra como fazer isso: usando um multiplicador no atributo ShuffleSent.

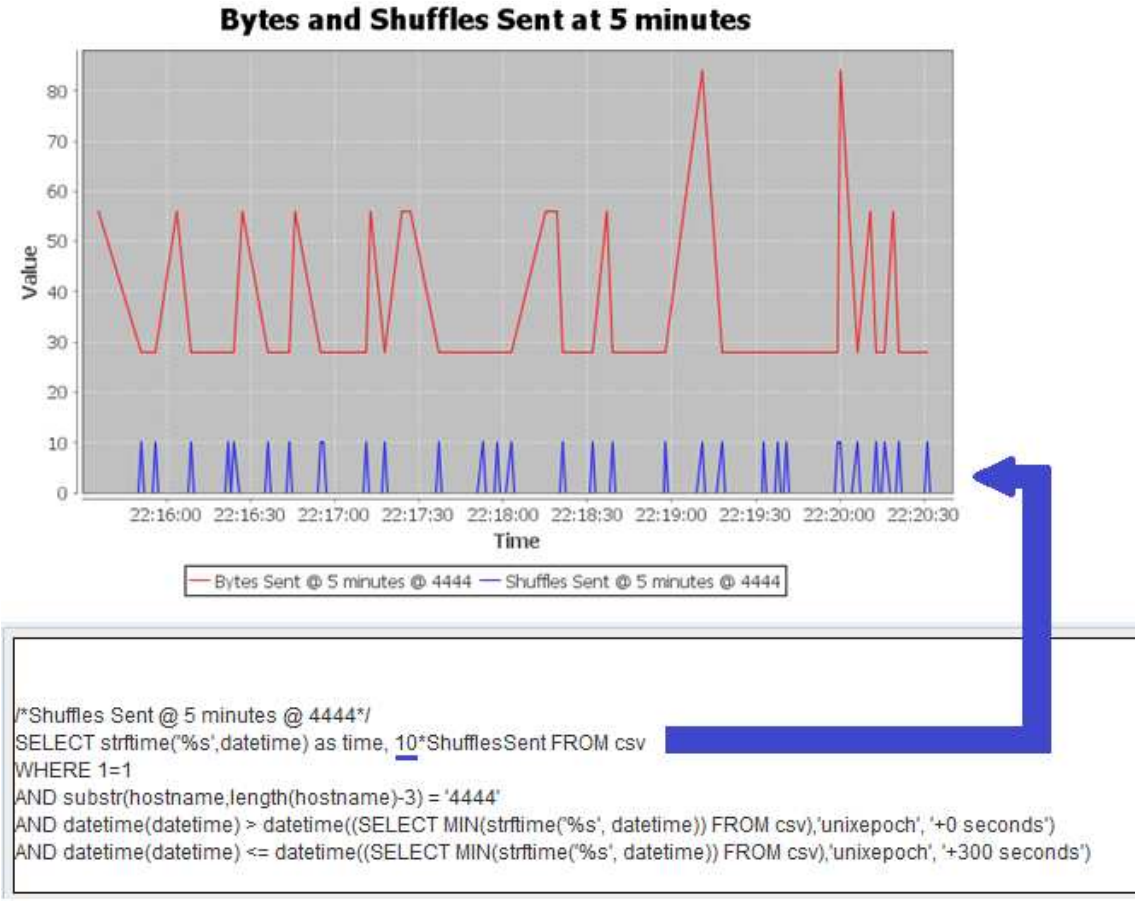


Figura 6.11: Consulta para evidenciar os picos e vales da série de amostras do atributo “Shuffles Enviados por nodo”

A figura acima deixa a série azul muito abaixo da outra, o que aumenta a dificuldade de se relacionar os “picos e vales” (acréscimos e decréscimos, respectivamente, nos valores a série). Podemos trazer essa série para cima se somarmos um valor ao atributo ShufflesSent, como mostrado na figura 6.12.

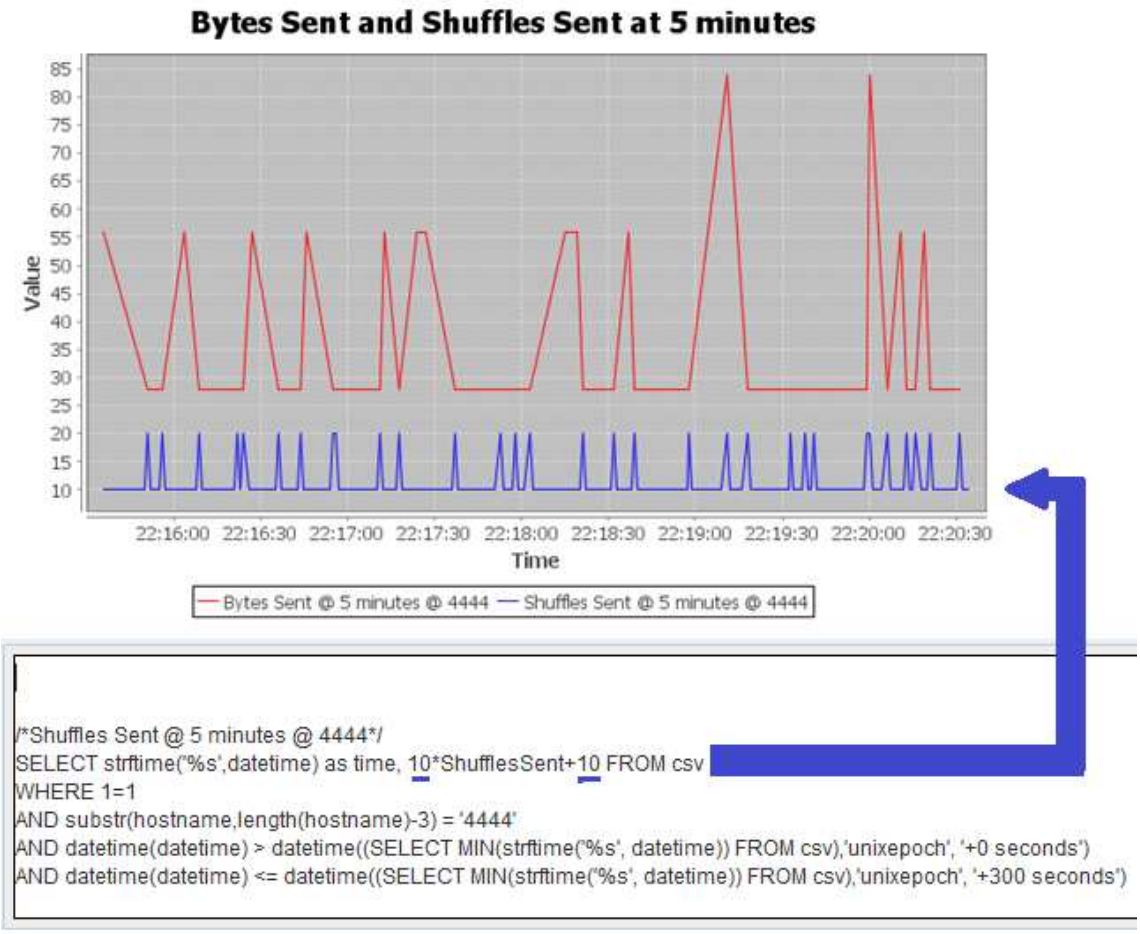


Figura 6.12: Consulta para evidenciar os picos e vales da série de amostras do atributo e aumentar seu *offset* “Shuffles Enviados por nodo”

Agora pode-se começar a tentar explicar as relações entre os dois eventos mostrados. Um último ajuste seria retirar da consulta que gera a série vermelha (bytes enviados) a mesma restrição que havia sido retirada da série azul, o que faria serem representados os momentos sem a ocorrência de envios de bytes.

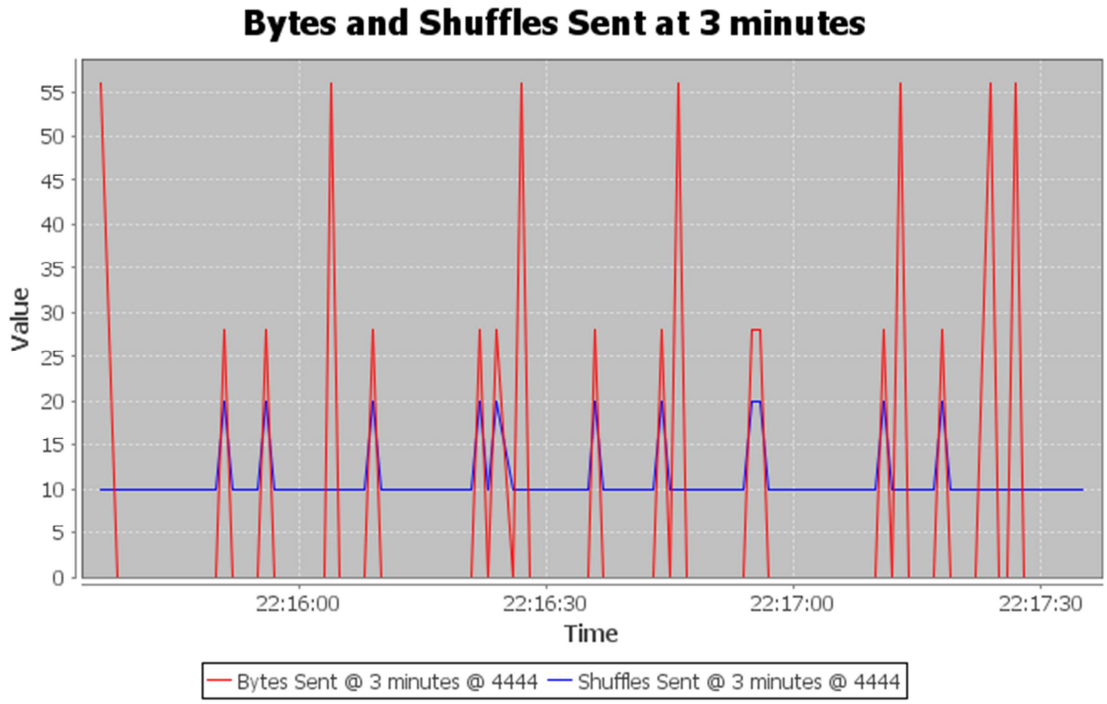
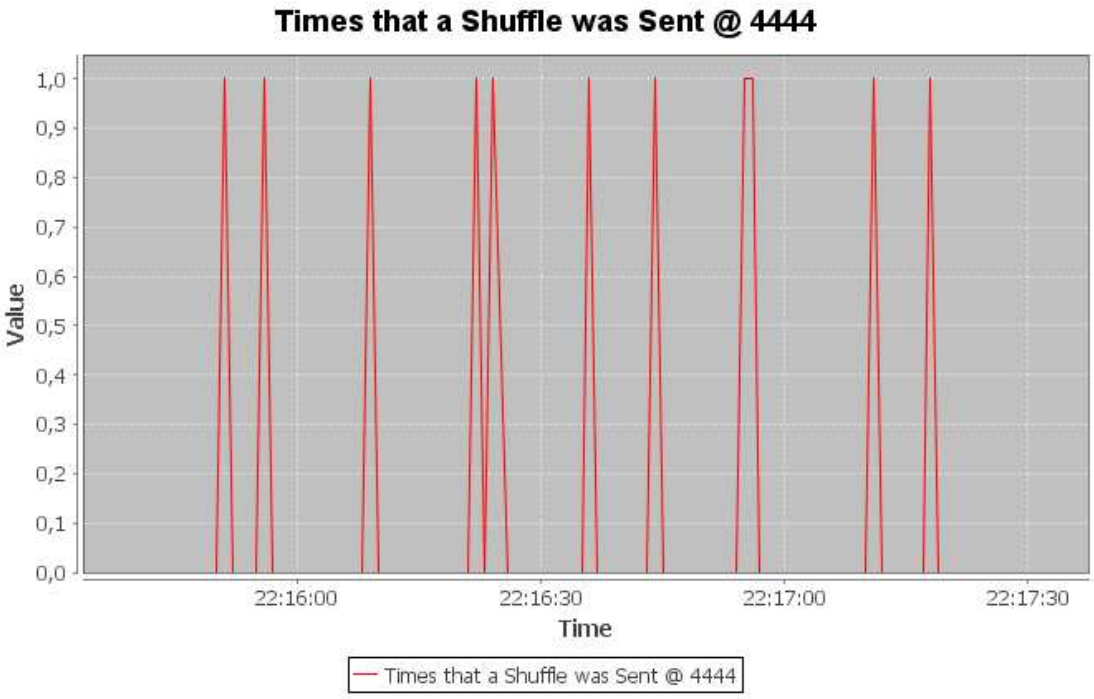


Figura 6.13: Gráfico comparativo dos Bytes e Shuffles enviados num período de 3 minutos de experimento

Finalmente se nota claramente pela figura 6.13 que os eventos de shuffle sent (picos azuis) causam o envio de alguns poucos bytes (mais precisamente, 28 bytes, segundo os dados tabelados). Ainda, uma quantidade maior de bytes é enviada (mais precisamente, 56 bytes, segundo os dados tabelados) logo depois, provavelmente representando o restante de cada mensagem de shuffle que não havia sido enviada anteriormente.

Outra análise que pode ser feita é para averiguar se o ShufflePeriod (constante, com valor de 10 segundos, durante o experimento) está sendo respeitado. Ainda, devemos lembrar que o envio de uma mensagem de Shuffle não depende somente do ShufflePeriod, mas também da quantidade de Joins recebidos (nula, no nosso caso) e de Shuffles Recebidos (mas apenas em 50% das vezes). Ainda, o NEeM manda uma mensagem de Shuffle a cada vizinho do nodo; podemos teorizar que os vizinhos do nodo 4444 sempre foram o 4445 e o 4446, mas talvez isso não tenha sido sempre verdade em todo o período do experimento.

Com esses questionamentos em mente, a figura 6.14 demonstra a quantidade de Shuffles enviados num período de 2 minutos de experimento, em forma gráfica e em forma tabular.




```
/*Times that a Shuffle was Sent @ 4444*/
SELECT time(datetime) as time FROM csv
WHERE 1=1
AND ShufflesSent > 0
AND substr(hostname,length(hostname)-3) = '4444'
AND datetime(datetime) > datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch', '+0 seconds')
AND datetime(datetime) <= datetime((SELECT MIN(strftime('%s', datetime)) FROM csv),'unixepoch', '+120 seconds')
```

time
22:15:51
22:15:56
22:16:09
22:16:22
22:16:24
22:16:36
22:16:44
22:16:55
22:16:56
22:17:11
22:17:18

Figura 6.14: Dados contendo os Shuffles enviados pelo nodo 4444 num período de 3 minutos de experimento

Nota-se pela análise dos dados tabulares da figura 6.14, que sempre que 2 mensagens de Shuffle eram enviadas, uma terceira era enviada num determinado tempo depois. A figura 6.15 tenta cruzar essas ocorrências “isoladas” com eventos de recebimento de mensagens de Shuffle.

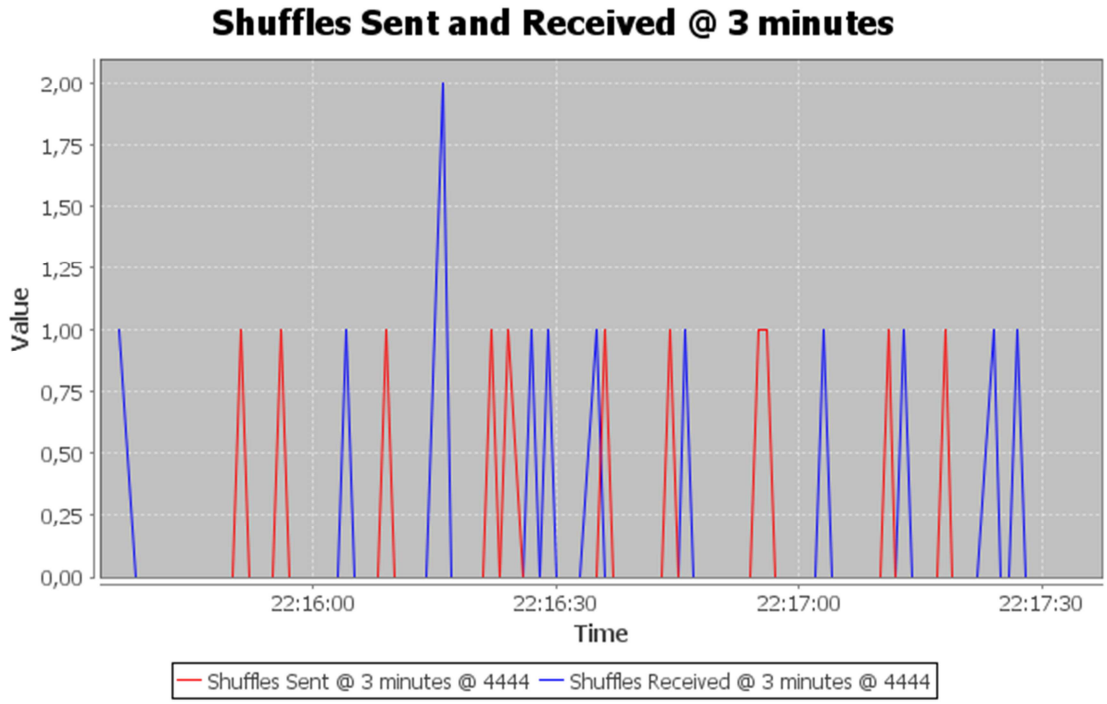


Figura 6.15: Gráfico comparativo dos Shuffles enviados e recebidos pelo nodo 4444 num período de 3 minutos de experimento

Dentre as mensagens de Shuffle recebidas, exibidas na figura 6.15, parece que a que ocorreu perto de 22:16:00 realmente causou o envio de uma mensagem de Shuffle. Isso não ocorreu com as duas mensagens de Shuffle recebidas perto de 22:16:15, mas parece ter voltado a acontecer com as mensagens recebidas em 22:16:30 e as 22:16:45, dado a distância curta entre os eventos de recebimento e envio de Shuffle.

6.1.4 Ponderações finais sobre a análise do experimento com o programa Crowd

A resposta para a terceira pergunta, feita no início desse experimento, permanece, portanto em aberto. Foram feitas ponderações e hipóteses somente manipulando os dados, mas ainda se pode investigar bastante a relação entre os atributos (por exemplo, analisando-se os bytes enviados por um nodo e recebidos por outros). Também se possibilita modificar os atributos a serem monitorados, afim de melhor representar cada evento sendo realizado (por exemplo, quando se recebe uma mensagem de Shuffle e gera-se um envio de Shuffle por causa dela, poderia ser um evento contabilizado separadamente dos eventos de envio de Shuffle periódicos).

6.2 Testes com programa demonstrativo de Chat do NEeM

Após o desenvolvimento do protótipo ter sido considerado estável o suficiente, começaram a ser feitos testes utilizando a classe Chat, disponibilizada pelos criadores do NEeM no pacote de aplicações demonstrativas. Essa aplicação cria uma “rede de conversação”, onde cada nodo conversa (recebe e envia mensagens) com todos os outros via mensagens inseridas por linha de comando.

6.2.1 Visão geral

Ao iniciar o programa de Chat, o usuário recebe o identificador daquele nodo, como mostrado na figura 6.16. Esse identificador será utilizado para identificar as mensagens provenientes desse nodo e para que outros nodos, ausentes da "conversação", se juntem a ela.

Figura 6.16: Exemplo de identificador de um nodo, conforme ele é fornecido ao usuário

O programa de Chat inicial, disponibilizado pelos autores do NEeM, somente repassava a mensagem a todos os interessados. Para melhor acompanhar as mensagens e conseguir analisar seus efeitos em todos os nodos da rede, o programa utilizado nesse trabalho adicionou as seguintes informações a cada mensagem:

- Identificador do nodo emissor da mensagem, entre os caracteres "<" e ">";
- Data e hora na qual a mensagem foi emitida pela primeira vez, entre os caracteres "[" e "]";

A figura 6.17 exemplifica uma mensagem enviada por um nodo e recebida de um nodo remoto. Note como a mensagem recebida nos permite saber o momento exato em que ela foi enviada e a identificação do seu emissor.

Figura 6.17: Exemplo de mensagens, conforme elas são fornecidas ao usuário

O experimento foi realizada com três nodos, que, da mesma forma que no experimento da seção 6.1, são nomeados conforme suas portas de comunicação: 53197, 53187, 53182. Na figuras 6.18 seguem completas as conversas que foram monitoradas, como vista pelo nodo 53182, 53187 e 53197.

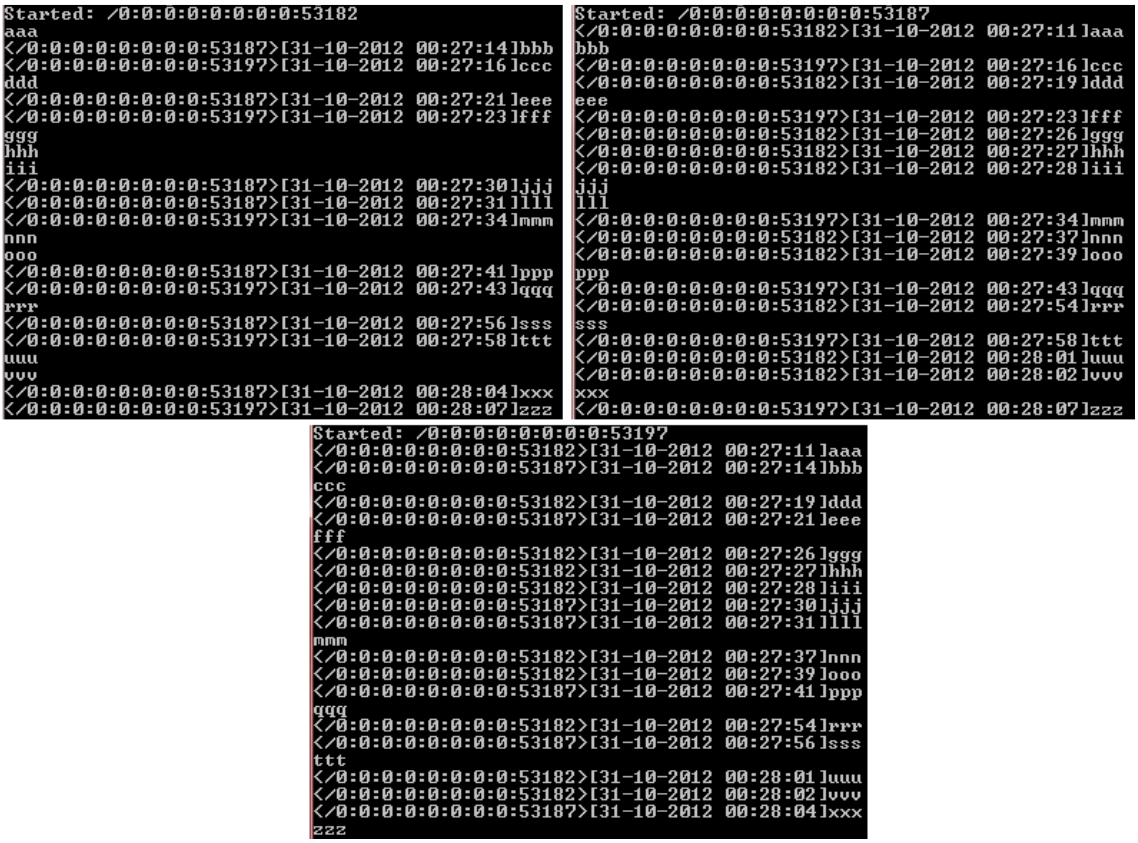


Figura 6.18: Conversação monitorada, conforme vista pelo nodo 53182 (topo, a esquerda), 53183 (topo, a direita) e 53197 (centro)

Como já foi explorado o efeito das mensagens de Shuffle na análise do experimento da seção anterior, gostaríamos de focar nas mensagens Multicast enviadas e recebidas pelos nodos monitorados. Para tanto, foi reduzida a duração desto experimento, com o objetivo de gerar o menos tráfego de mensagens de Shuffle possível, sem pará-lo totalmente.

A tabela 6.2 mostra o comportamento dos atributos monitorados nessa seção de experimento:

Atributos	Comportamento
MaxIds	**Constante (valor 100)**
Delivered	**Crescente**
Multicast	**Crescente**
DataSent	**Crescente**
HintsSent	**Constante (valor 0)**
PullSent	**Constante (valor 0)**
QueueSize	**Constante (valor 10)**
BytesSent	**Crescente**
GossipFanout	**Constante (valor 11)**
PushTimeToLive	**Constante (valor 2)**
MinPullSize	**Constante (valor 64)**

PullPeriod	**Constante (valor 120)**
DataReceived	**Crescente**
HintsReceived	**Constante (valor 0)**
PullReceived	**Constante (valor 0)**
ShufflePeriod	**Constante (valor 10)**
OverlayFanout	**Constante (valor 15)**
JoinRequests	**Constante (dependente do nodo)**
PurgedConnections	**Constante (valor 0)**
ShufflesReceived	**Crescente**
ShufflesSent	**Crescente**
BufferSize	**Constante (valor 1024)**
AcceptedSocks	**Crescente**
ConnectedSocks	**Crescente**
PacketsReceived	**Crescente**
PacketsSent	**Crescente**
BytesReceived	**Crescente**
TimeToLive	**Constante (valor 6)**

Tabela 6.2: Comportamento de cada atributo monitorado durante experimento de Crowd

Com o intuito de restringir o escopo da nossa análise, vamos focar nos atributos mostrados na figura 6.19.

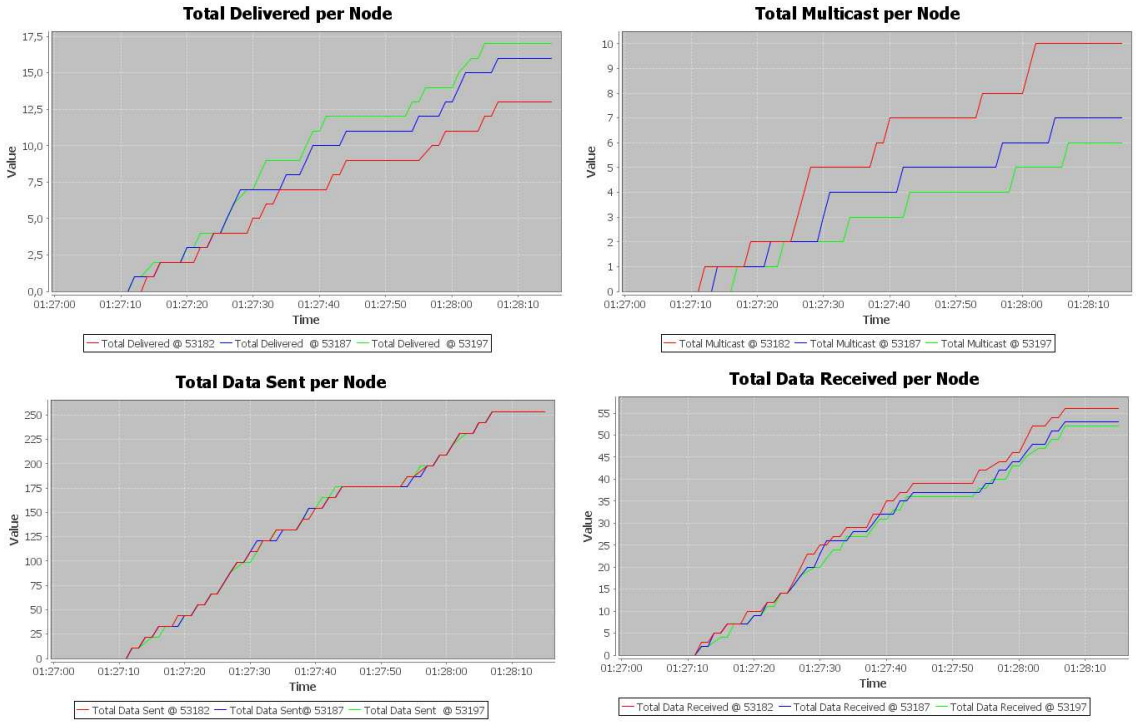


Figura 6.19: Gráficos por atributo monitorado durante experimento de Chat

6.2.2 Análise dos dados mais primitivos

Para começar nossa análise do programa de Chat, vamos primeiro tentar entender o comportamento das mensagens enviadas e recebidas.

Consultando a tabela 6.2,sabemos que nenhum dos nodos da aplicação Chat enviou avisos de *ack* ou *nack* (visto que os atributos PullSent, PullReceived, HintsSent e HintsReceived estão sempre zerados), ou seja, sempre usaram a estratégia de enviar a mensagem completa. Isso acontece porque todos os nodos estão interligados e, portanto, a apenas 1 *hop* de distância, o que é sempre menor que o parâmetro *Gossip.PushTTL* (com valor padrão de 2).

Sabendo disso, nos resta analisar os atributos DataSent e DataReceived e explicar quais mensagens poderiam ter causado um acréscimo no valor deles. A figura 6.20 compara se os dados enviados por um nodo qualquer realmente chegaram aos outros nodos.

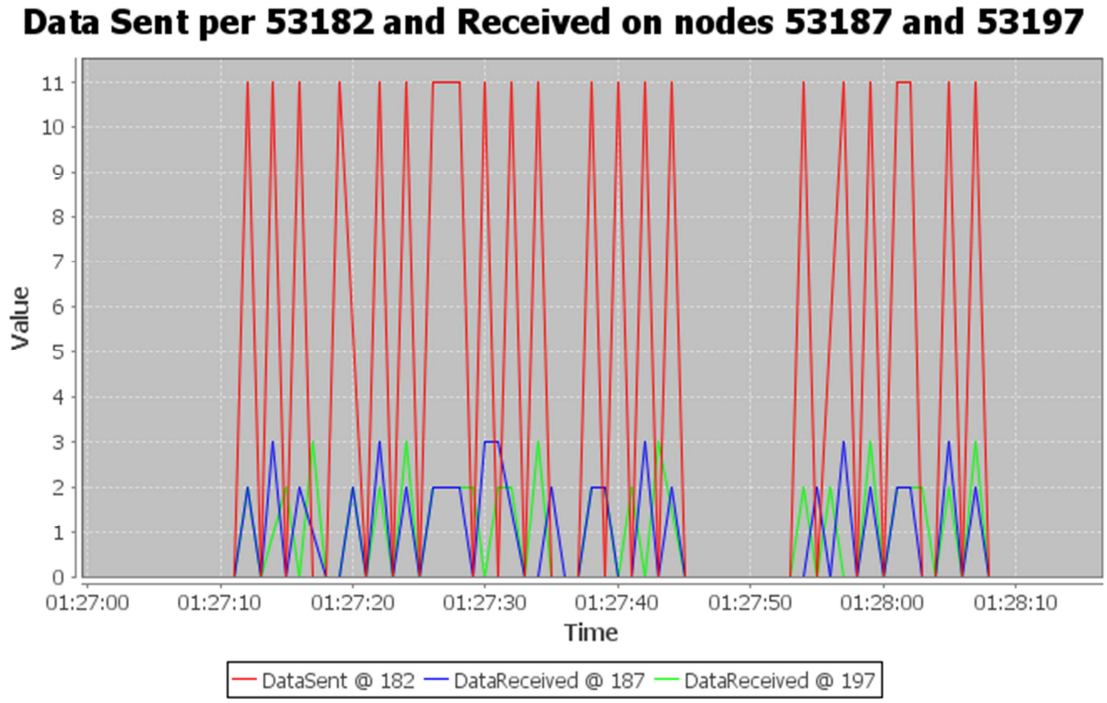


Figura 6.20: Gráfico demostrando as mensagens enviadas pelo nodo 53182 e recebidas pelos outros nodos durante experimento de Chat

Pela figura 6.20 acima, vemos que houve um momento de silêncio de quase 10 segundos na conversação, entre o momento 00:27:45 e o momento 00:27:54. Voltando a figura 6.18, nota-se que esse momento realmente aconteceu, mais exatamente entre as mensagens “qqq”, enviada pelo nodo 53197, e “rrr” enviada pelo nodo 53182.

Ainda pela figura 6.16, vemos que sempre 11 mensagens parecem ser enviadas, e 2 ou 3 recebidas pelos outros nodos. Esse número grande de mensagens enviadas não é por acaso. Sempre que o atributo *Gossip.dataOut* é incrementado, o valor desse incremento é o parâmetro *Gossip.fanout*. O NEeM 0.7 considera que todo envio de

mensagens é feito para todos os vizinhos e esse número é expresso pelo *fanout*, mesmo que haja menos conexões ativas do que esse valor.

Já que a medida de DataSent apresenta esse erro na hora de incrementar, nos resta explorar um pouco melhor quando as mensagens são recebidas e tentar descobrir quais são elas. A figura 6.21 mostra os dados brutos de um dos nodos.

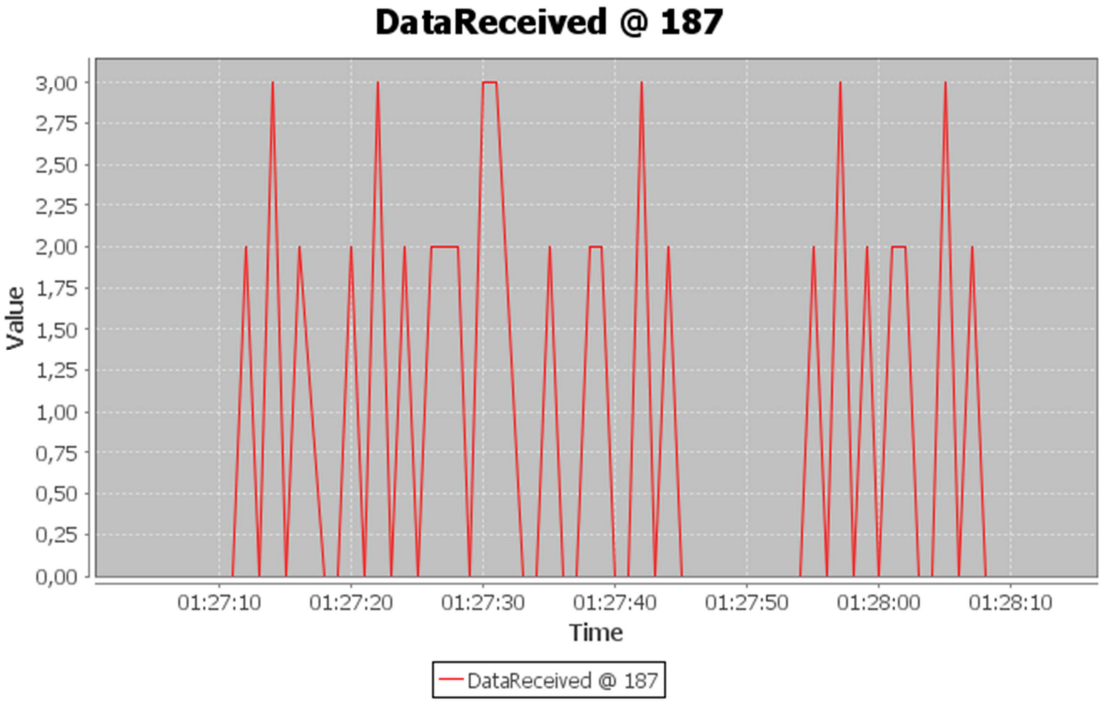


Figura 6.21: Gráfico demostrando as mensagens recebidas pelo nodo 53187 durante experimento de Chat

Olhando o código do NEeM, descobrimos que, quando uma mensagem é recebida no canal de dados, talvez ela seja na verdade um Multicast por parte do usuário da biblioteca. Ainda, talvez essa mensagem recebida tenha sido entregue ao usuário (*delivered*). A figura 6.22 tenta explorar essas possibilidades.

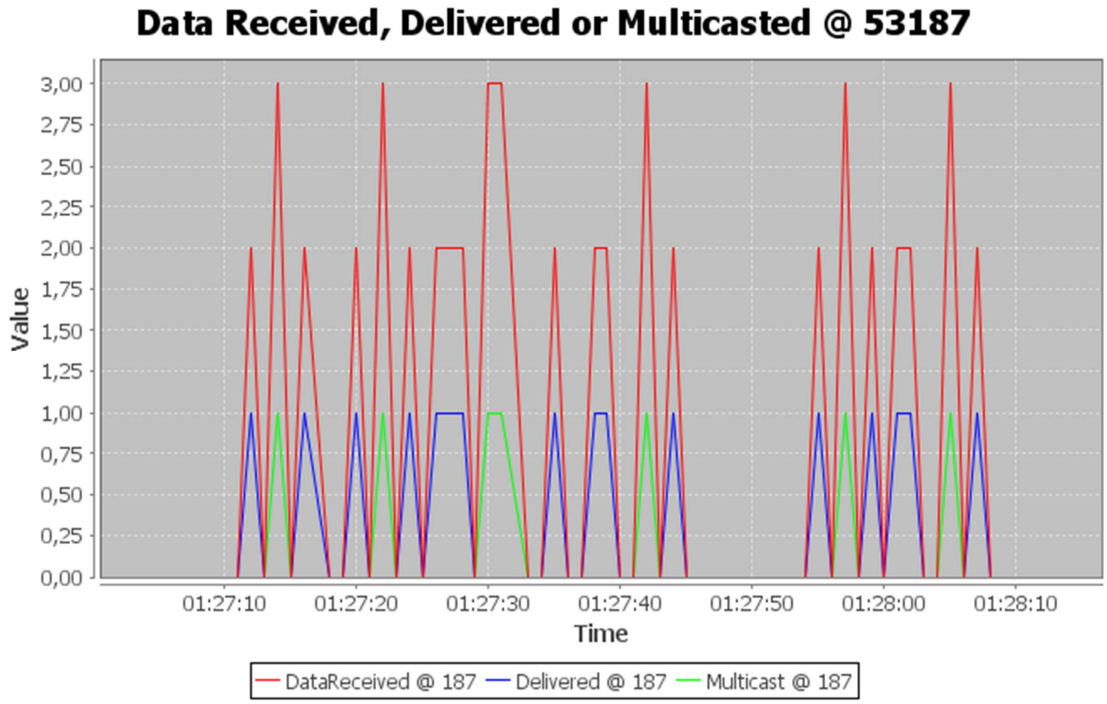


Figura 6.22: Gráfico demostrando as mensagens recebidas pelo nodo 53187 e a relação delas com eventos de Deliver e Multicast, durante experimento de Chat

Note pela figura 6.22 que todos os eventos de Data Received foram ou por causa de uma mensagem que o usuário solicitou o envio ou foi entregue a ele. Note, ainda, que todos os eventos de solicitação de multicast causaram 2 outras mensagens (além da própria de multicast) a passar pelo objeto Gossip. Por outro lado, as mensagens que acabaram sendo entregues ao usuário final causaram uma outra mensagem (além da que foi entregue) a passar por esse objeto. A figura 6.23 faz a subtração desses dados para termos uma melhor visualização de escala.

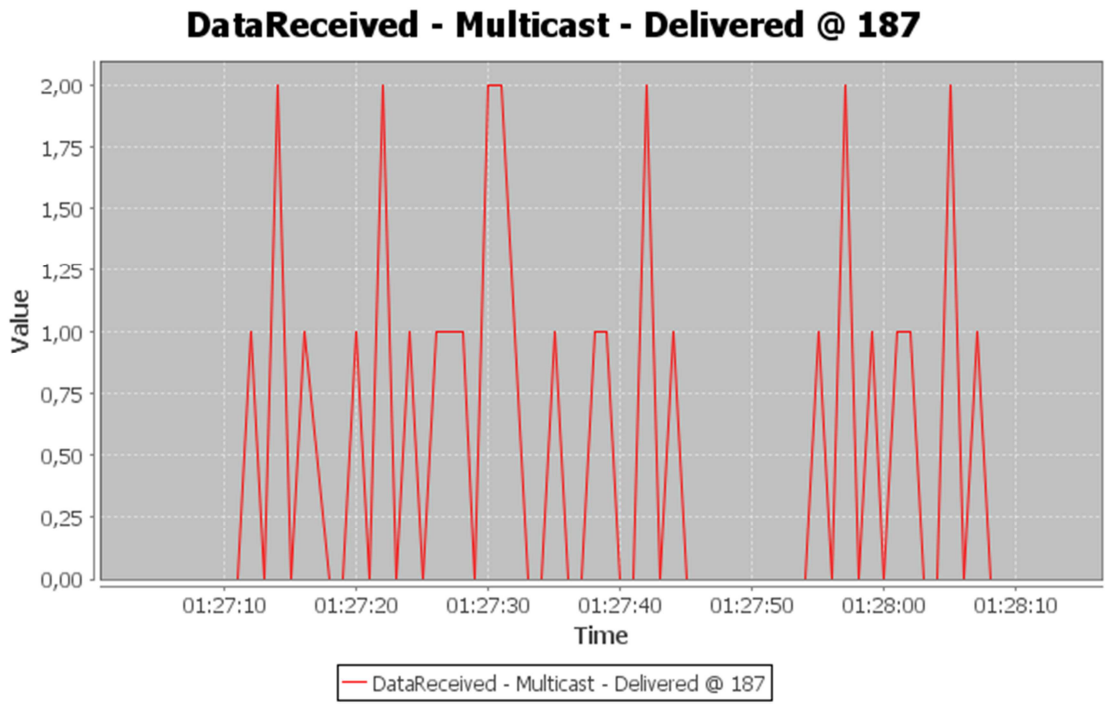


Figura 6.23: Gráfico demostrando as mensagens recebidas pelo nodo 53187, já excluindo aquelas que foram delivered ao usuário ou multicasted, durante experimento de Chat

A próxima sessão retorna a conversa entre os usuários, explicitada na figura 6.18, para melhor entender de onde vieram essas mensagens restantes recebidas por cada nodo.

6.2.3 Análise semanticamente mais rica

Para melhor comparar o comportamento monitorado com aquele visível pelo usuário do programa de Chat (e mostrado na figura 6.18), podemos começar a analisar quais mensagens foram Delivered (entregues). A figura 6.24 mostra os momentos exatos em que cada mensagem foi entregue ao usuário, em formato tabular.

```
/*Total Delivered @ 53182*/
SELECT time(datetime) as datetime, Delivered FROM csv
WHERE 1=1
AND substr(hostname, length(hostname)-4) = '53182'
AND Delivered > 0
```

datetime	Delivered
00:27:14	1
00:27:16	1
00:27:22	1
00:27:24	1
00:27:30	1
00:27:32	1
00:27:34	1
00:27:42	1
00:27:44	1
00:27:57	1
00:27:59	1
00:28:05	1
00:28:07	1

Figura 6.24: Consulta demostrando os momentos nos quais as mensagens foram entregues ao usuário do nodo 53182 durante experimento de Chat

Comparando os dados da figura 6.18 e 6.24, notamos que a maioria dos momentos de tempo estão compatíveis, embora alguns dados possuam uma ligeira diferença de 1s.

Essa diferença acontece por causa da diferença entre o tempo que é mostrado ao usuário na tela de comando e o tempo em que a amostragem dos dados foi feita em cada nodo. Ainda, como cada nodo possui uma *thread* que obtém as amostras de dados de maneira independente das outras, a ordem em que esses eventos são amostrados e realizados pode ser diferente, como evidenciado na figura 6.25.

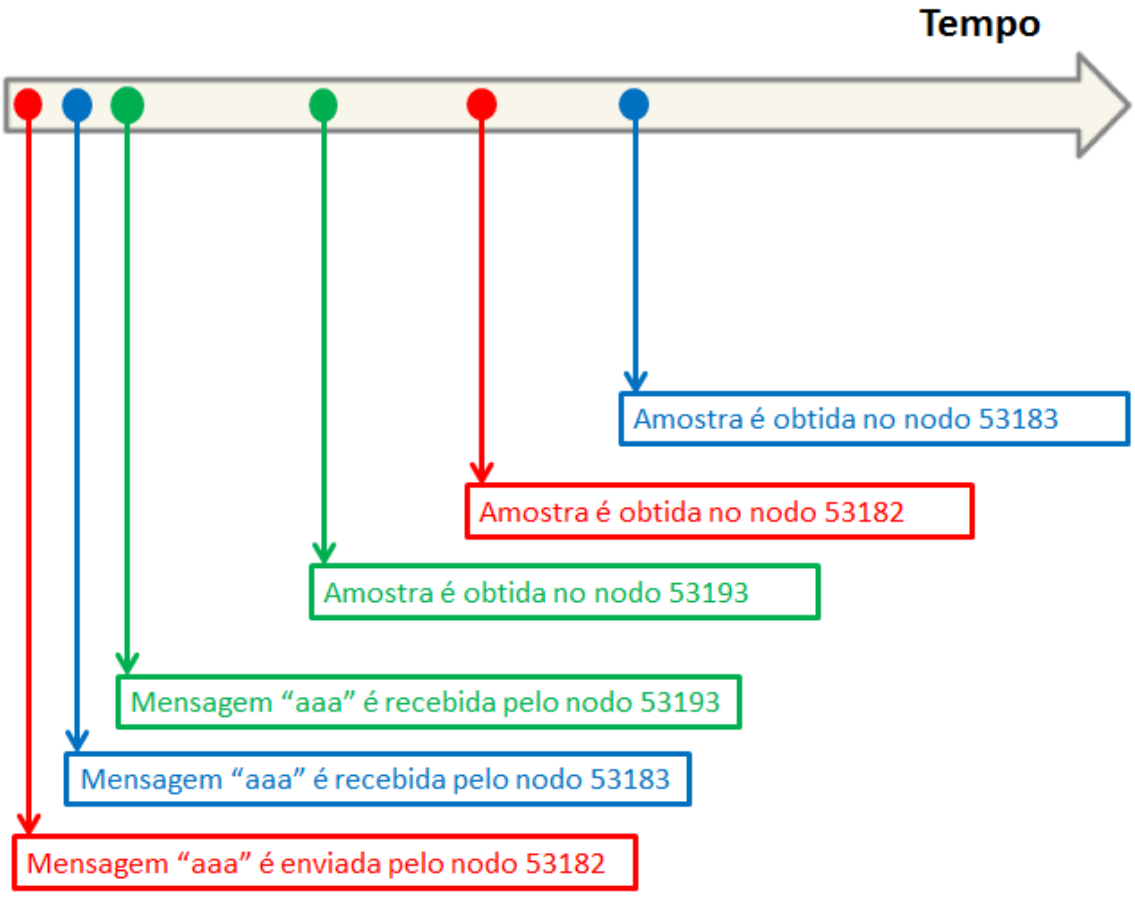


Figura 6.25: Exemplo de como a ordem dos eventos e a ordem de obtenção de amostras pode variar

Os tempos que são utilizados nas análises de dados do LUsIR são os tempos em que as amostras foram obtidas. Os momentos de tempo em que os eventos realmente aconteceram só podem ser estimados, já que o NEeM não envia para o programa de monitoramento esses dados. Programas que utilizam o NEeM precisam prover essa informação, se assim desejarem (como o programa de Chat faz). Ainda assim, o programa de monitoramento (o LUsIR) não tem como conhecer esse timestamp.

Esse problema fica evidente ao tentar cruzar as informações de mensagens Multicast com eventos Delivered. Seria interessante conseguir cruzar esses dados, para analisar como as mensagens Multicast de um nodo geram eventos Delivered em todos os outros.

A figura 6.26 mostra uma situação em que uma mensagem Multicast gerou 2 eventos Delivered, em momentos de tempo sequenciais, exatamente como a aplicação de Chat se comportou. A consulta retorna amostras ordenadas no tempo, com a informação de que nodo gerou aquele dado. Note como o nodo 53182 gerou uma mensagem de Multicast e, alguns milisegundos depois, aconteceram eventos de Delivered nos nodos 53187 e 53197.

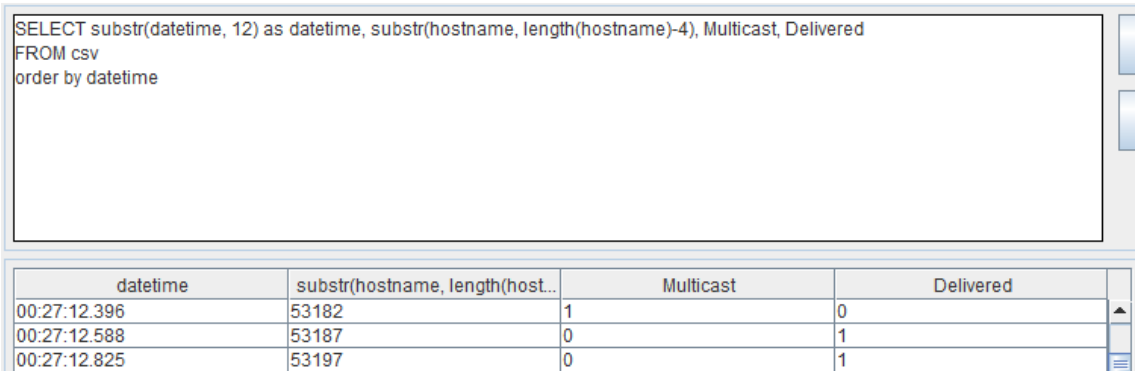


Figura 6.26: Situação em que o momento de tempo no qual as amostras foram obtidas reflete a ordem dos eventos na aplicação

Já na figura 6.27, a amostra que detecta a mensagem de Multicast (disparada pelo nodo 53187) foi obtida num determinado momento de tempo após um evento de Delivered ter sido gerado (no nodo 53182). Logo depois, a próxima amostra capturou um novo evento de Delivered no nodo restante.

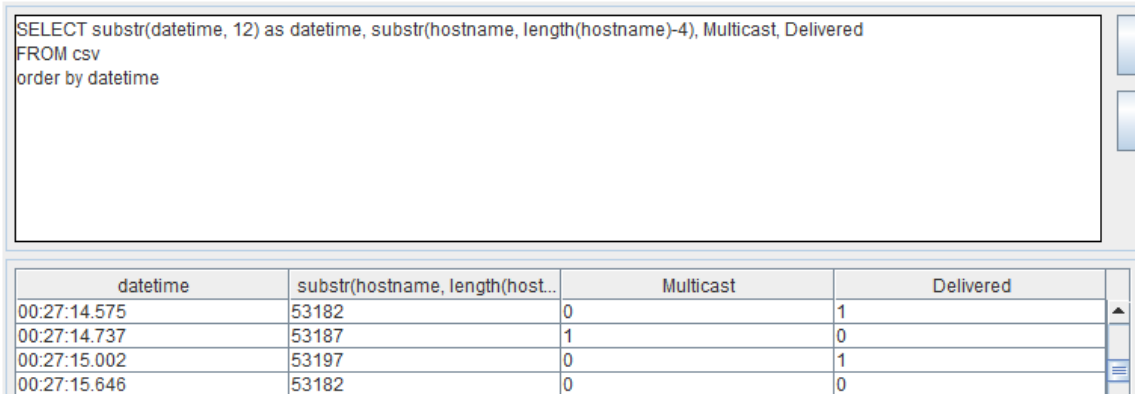


Figura 6.27: Situação em que o momento de tempo no qual as amostras foram obtidas não reflete a ordem dos eventos na aplicação

6.2.4 Ponderações finais sobre a análise do experimento com o programa Chat

Assim como o experimento com o programa Crowd, o que resta na análise do programa Chat são perguntas em aberto. Como relacionar corretamente os momentos de tempo nos quais o usuário solicitou o multicast de uma mensagem à biblioteca NEeM e o momento de tempo em que seus vizinhos efetivamente receberam essa mensagem a partir da interface com essa mesma biblioteca ? A figura 6.18 mostra que isso efetivamente aconteceu, mas o programa de monitoramento não obteve dados suficientemente ricos para mostrar isso.

7 CONCLUSÃO

Os capítulos anteriores se preocuparam em:

- Introduzir o conceito de protocolos epidêmicos e como eles serviram de base para a criação do NEeM, o primeiro protocolo epidêmico que se preocupou em expor uma interface (no caso, JMX) para instrumentação remota;
- Descrever essa interface JMX em detalhes e comparar outras formas de monitoramento existentes;
- Criar um protótipo (no caso, o LUsIR) que pudesse auxiliar tanto no monitoramento quando na análise dos dados gerados por diversos nodos, de forma integrada e que facilitasse a exploração dos dados;
- Executar experimentos demonstrando a utilização e eficácia do LUsIR.

A partir desses itens descritos, se mostrou a utilidade de uma ferramenta gráfica na análise de dados monitorados. Ainda, existe o diferencial de o LUsIR ser usado, de forma integrada, tanto para análise quanto para monitoramento, diferente de outras ferramentas, que desempenham apenas um papel.

Porém, desde o início foi criada uma dependência do LUsIR com o NEeM. Essa dependência se mostrou útil ao criar a oportunidade de se entender profundamente o funcionamento do protocolo e como suas ações influenciavam os atributos a serem monitorados e ao direcionar e guiar os esforços de monitoramento para a interface JMX.

Ao tentar atacar essa dependência e se modificar o protocolo a ser monitorado, alguns desafios podem aparecer. Um deles é a necessidade de se informar quais atributos são contadores acumuláveis (como o BytesSent do LUsIR), pois o programa precisa saber que deve controlar tanto o atributo lido quando suas modificações entre amostras, e guardar esses dados de maneira diferente. Um segundo seria modificar qual a classe possui uma interface JMX (no caso do NEeM, a classe Protocol é a única monitorável). Na ausência de uma, deve-se criá-la, o que envolve também definir quais atributos serão monitorados.

Felizmente, apenas a classe Sensor precisa ser modificada no LUsIR. Precisa-se também, respeitar a interface dela com a classe LiveMonitorFrame.

A classe Sensor que hoje existe já sabe lidar com atributos não-monitoráveis (ou seja, aqueles que não podem ser representados por números inteiros) e descobrir o nome de nodos e atributos a serem monitorados, conseguindo, portanto, lidar com um número variável de atributos e nodos. As classes que lidam com os dados (CSVExporter, CSVImporter e DBHandler) também estão preparadas para lidar com essa variabilidade.

Outros pontos de melhoria no LUsIR, que ficam como sugestão para trabalhos futuros, são:

- Um meio do usuário informar, via interface gráfica, quais IPs e Portas a serem monitorados via RMI (hoje, essa informação deve ser informada no código fonte do programa);

- A possibilidade de se visualizar vários atributos durante o experimento (hoje, apenas um deles pode ser visualizado por vez);
- A possibilidade de invocar métodos remotos para re-configurar parâmetros em algum nodo, enquanto ele é monitorado (no NEeM, poderiam ser modificados os valores dos buffers de mensagens, ou o *fanout* do protocolo, por exemplo);
- Um meio mais amigável para exploração dos dados (como um *wizard* que ajudasse a criar consultas);
- Facilidades no editor de consultas, como *syntax highlight*, evidência do nível de tabulação de todas as linhas, funções de “desfazer digitação” ou “refazer digitação” ou “autocompletar”, etc;
- Possibilidade de exibição de dados em outras formas gráficas (como a possibilidade de se criar gráficos em colunas ou em torta, por exemplo), o que cria o desafio de criar consultas-templates adequadas para formatar corretamente os dados nessas formas gráficas;

Por fim, segue uma listagem final das qualidades do LUsIR como ferramenta de monitoramento e análise do protocolo NEeM:

- Integração plena com a última versão do protocolo, contendo os Oráculos que foram introduzidos em Pereira (2011);
- Facilidade na gerência e manipulação dos dados, devido à fácil comunicação entre módulo de monitoramento e análise;
- Facilidade na análise de dados genéricos, devido ao uso pleno do formato CSV;
- Liberdade na exploração dos dados para usuários com conhecimentos em SQL;
- Facilidade de exploração dos dados em formato gráfico e tabular, de maneira interconectada;

REFERÊNCIAS

A. FINAMORE; M. MELLIA; M.MEO; M.M. MUNAFÒ; D. ROSSI. **Live Traffic Monitoring with Tstat: Capabilities and Experiences.** WWIC 2010, LNCS 6074, p.290–301, 2010.

ALAN DEMERS; MARK GEALY; DAN GREENE; CARL HAUSER; WES IRISH; JOHN LARSON; SUE MANNING; SCOTT SHENKER; HOWARD STURGIS; DAN SWINEHART; DOUG TERRY; DON WOODS. **Epidemic Algorithms for Replicated Database Maintenance.** Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, p.1–12, 1987.

EUGSTER, P. T.; GUERRAOUI, R.; KERMARREC, A. M.; MASSOULIÉ, L. **Epidemic information dissemination in distributed systems.** Computer. v. 37, p.60–67, 2004.

FREY, D.; GUERRAOUI, RACHID; KERMARREC, A.-M.; KOLDEHOFE, B.; MOGENSEN, M.; MONOD, M.; QUÉMA, V. **Heterogeneous gossip.** Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. , Middleware '09.. p.3:1–3:20. New York, NY, USA: Springer-Verlag New York, Inc. Retrieved October 12, 2012, from http://dl.acm.org/citation.cfm?id=1656980.1656984, 2009.

J. PEREIRA; L. RODRIGUES; M. J. MONTEIRO; R. OLIVEIRA; A.-M. KERMARREC. **NEEM: Network-friendly Epidemic Multicast.** NEEM: Network-friendly Epidemic Multicast. ,2003.

JOSÉ PEREIRA; RUI OLIVEIRA; LUÍS RODRIGUES. **Efficient Epidemic Multicast in Heterogeneous Networks.** . Montpellier, France, 2006.

KENNETH P. BIRMAN; MARK HAYDEN; OZNUR OZKASAP; ZHEN XIAO; MIHAI BUDIU; YARON MINSKY. **Bimodal multicast.** ,1999.

KUROSE, J. F.; ROSS, K. W. **Computer networking**. Pearson/Addison Wesley. Retrieved September 9, 2012, from http://www.aw.com/info/kurose/preface.pdf, 2005.

LEITAO, J.; PEREIRA, J.; RODRIGUES, L. **HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast.** Dependable Systems and Networks, 2007. DSN

’07. 37th Annual IEEE/IFIP International Conference on. p.419–429. doi: 10.1109/DSN.2007.56, 2007.

M. MELLIA; R. LO CIGNO; F. NERI. **Measuring IP and TCP behavior on edge nodes with Tstat.** Computer Networks. p.1–21, 2005.

MAGNUS ALMGREN; ULF LINDQVIST. **Application-Integrated Data Collection for Security Monitoring.** p.22–36, 2001.

NUNO CARVALHO; JOSÉ PEREIRA; RUI OLIVEIRA; LUÍS RODRIGUES. **Emergent structure in unstructured epidemic multicast.** p.481–490. Washington, DC, USA, 2007.

P. TH. EUGSTER; R. GUERRAOUI; S. B. HANDURUKANDE; P. KOUZNETSOV. **Lightweight probabilistic broadcast.** ,2001.

R. KARP; C. SCHINDELHAUER; S. SHENKER; B. VOCKING. **Randomized Rumor Spreading.** ,2000.

RIZZO, L. **Dummynet: a simple approach to the evaluation of network protocols.** ACM SIGCOMM Computer Communication Review. v. 27, p.31–41. Retrieved September 9, 2012, from http://dl.acm.org/citation.cfm?id=251007.251012, 1997.

SUN MICROSYSTEMS INC. **Java Management Extensions (JMX) Specification, version 1.4**. November 2004. Disponível em: <http://docs.oracle.com/javase/7/docs/technotes/guides/jmx/JMX_1_4_specification.pdf >. Acesso em: 16 dezembro 2012.

WILGES, P.; LOVISON, H. D. C. ; CECHIN, S. L. ; WEBER, T. S. ; MORAES, R. L. O. **Serviço de Presença sobre uma Estrutura Gossip em Cloud**. Escola Regional de Redes de Computadores, Pelotas, RS. p. 61-64, 2012.