

Monitoramento Distribuído de Coleta e Análise de Tráfego da rede em tempo real

Charles N. C. Freitas¹, Matheus M. M. Xavier¹, Thays Melo¹, Anderson Gomes¹,
Asaffe Carneiro¹

¹Departamento de Estatística e Informática – Universidade Federal Rural de
Pernambuco (UFRPE)
Caixa Postal CEP: 52171-900 – Recife – PE – Brasil
cnicollas21@hotmail.com, matheusmmx@gmail.com, thaysmelo.m@gmail.com,
gomes.ramos@gmail.com, asaffe.m@gmail.com

Abstract. *This paper presents the process of collecting and analyzing packets on the network in real time, featuring data stream and analyzing network traffic. Demonstrating a tool to monitor the network in a distributed manner, allowing greater control in the network traffic problem detections. In addition to discussing the concept and implementation of processing and distributed form of data storage.*

Resumo. *Este artigo apresenta o processo de coleta e análise de pacotes na rede em tempo real, caracterizando os fluxos de dados e analisando o tráfego na rede. Demonstrando uma ferramenta capaz de monitorar a rede de maneira distribuída e que possibilita um maior controle na detecção de problema de tráfego na rede. Além de discutir conceitos e implementações sobre processamento e armazenamento de dados de forma distribuída.*

1. Introdução

Este artigo apresenta o processo desenvolvido para elaboração de uma ferramenta que seja capaz de monitorar o tráfego na rede em tempo real. Desse modo se faz necessário à implementação de alguns componentes e algoritmos que auxiliaram no desenvolvimento do projeto. Seguindo as instruções do documento de especificação temos que a base da ferramenta é a implementação de diferentes pontos de coleta (Coletores) e classificação de tráfego que se comunica por meio de um protocolo com um gerente que monitora o estado dos coletores.

Os coletores utilizam comunicação indireta (filas) para enviar um *stream* de fluxos (de pacotes) classificados a um processador de fluxos que computará estatísticas sobre estes dados em tempo real. O processo de comunicação foi realizado via socket UDP, por ser um protocolo simples e com melhor desempenho para ser trabalhar com fluxos de dados em tempo real, além de permitir suporte a *broadcasting* e rápido processamento de pacotes.

O envio dos pacotes se dará pela comunicação indireta e assíncrona com o framework RabbitMQ, aonde será implementada as filas.

A análise dos pacotes enviados a fila se dá por meio da ferramenta de computação distribuída, Storm. Essa ferramenta será responsável por classificar os fluxos capturados por tamanho (ratos/elefantes), duração (tartarugas/libélulas), taxa (guepardo/caramujos) e calcular o percentual de cada classe de fluxos em cada tipo de aplicação.

2. Metodologia

A metodologia proposta, proveniente da implementação de uma Ferramenta de monitoramento da rede em tempo real, consiste em um ambiente integrado projetado para facilitar as análises dos dados que trafegam na rede. Conforme ilustra a Figura 1, a arquitetura geral do projeto.

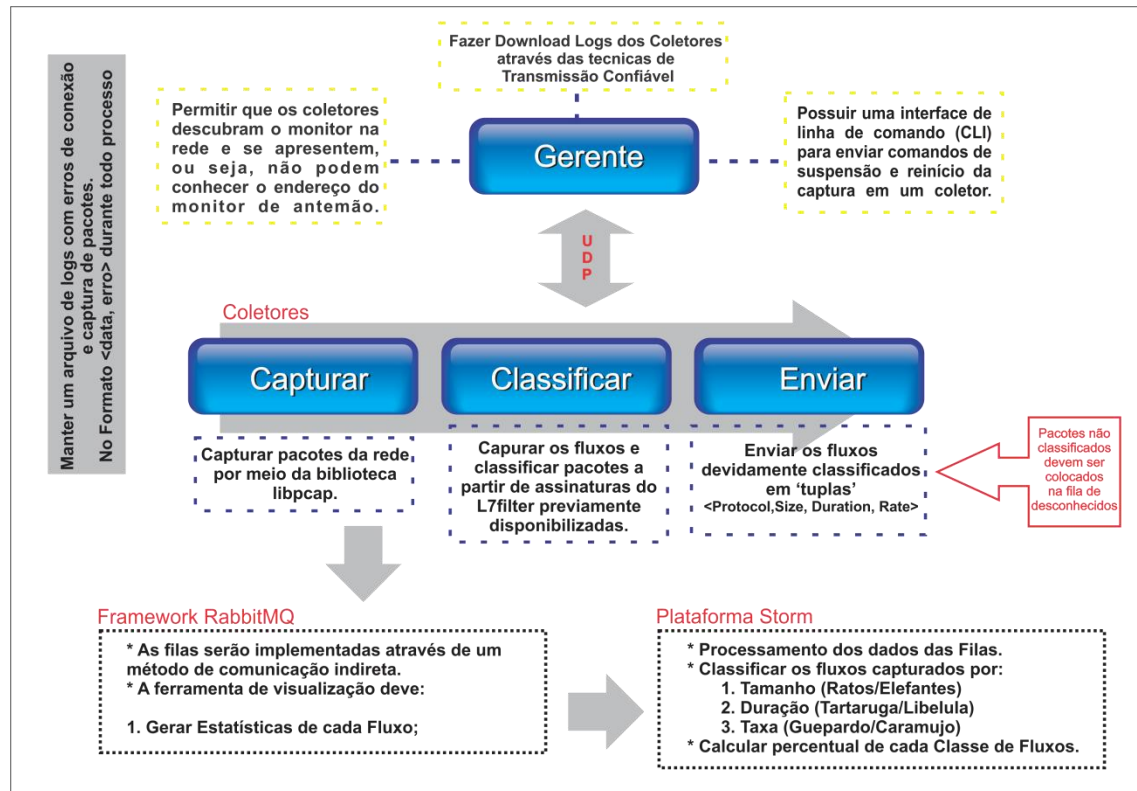


Figura 1 - Arquitetura Geral do Projeto

Para o desenvolvimento do projeto foi necessário à implementação de dois elementos principais, os Coletores e um Monitor. Sua utilização, implementação e componentes que as integram serão apresentadas e discutidas a seguir. Eles são utilizados em paralelo com as ferramentas de gerenciamento de filas e processamento de dados, RabbitMQ e Storm, respectivamente. Assim podendo ter uma análise em tempo real de transmissão e tráfego em uma rede. Para o desenvolvimento do projeto foram utilizadas as linguagens de programação Python e Java, de acordo as necessidades propostas.

2.1 Coletores

Os coletores são responsáveis por capturar pacotes na rede através da biblioteca libpcap e classifica-los a partir de assinaturas do L7filter previamente disponibilizadas e depois enviar informações para as filas específicas. O processo de captura se dar pelo esquema de fluxo contínuo dos dados, aonde a captura só para quando lhe é solicitado pelo Monitor.

2.1.1 Assinaturas L7filter e Capturas de Pacotes

Para o desenvolvimento do projeto foram utilizados apenas 6 (seis) protocolos bases, que foram utilizados para fazer a captura, classificação e análise, foram eles: bittorrent, dhcp, http, ssdp, ssh, ssl. Por questão de análise estatística todos os outros protocolos capturados, foram classificados como *unknown* (desconhecido). Abaixo encontra-se a implementação das assinaturas dos protocolos descritos acima.

```
def assinar_protocols(self, p1, p2, p3, p4, p5, p6):
    expr = p1[1]
    bittorrent = re.compile(expr)

    expr = p2[1]
    dhcp = re.compile(expr)

    expr = p3[1]
    http = re.compile(expr)

    expr = p4[1]
    ssdp = re.compile(expr)

    expr = p5[1]
    ssh = re.compile(expr)

    expr = p6[1]
    ssl = re.compile(expr)

    self.protocols = {"bittorrent":bittorrent,"dhcp":dhcp,"http":http,"ssdp":ssdp,"ssh":ssh,"ssl":ssl}
```

Função 1 - Assinatura Protocolos L7Filter

A partir dessa função é possível assinar os protocolos citados acima, para posteriormente classificar os pacotes capturados. Seguindo o processo de implementação, a função *capturar_pkts* é responsável pela captura dos pacotes na rede, além da capturar essa função é responsável por indicar e separar os pacotes recebidos ao dicionário de fluxo adequado, para posteriormente poder aplicar as métricas dos fluxos. Abaixo segue a função *capturar_pkts*:

```
def capturar_pkts(self):
    try:
        for ts, pkt in pcap.pcap():
            if self.start_captura is True:
                eth = dpkt.ethernet.Ethernet(pkt) #extraíndo dados do pacote
                ip = eth.data
                if isinstance(ip,dpkt.ip.IP):
                    ip_src = ip.src
                    ip_dst = ip.dst
                    protocol = ip.p
                    if isinstance(ip.data,dpkt.tcp.TCP) or isinstance(ip.data,dpkt.udp.UDP):
                        Port_src = ip.data.dport
                        Port_dest = ip.data.sport

                    #Gerar chave e tupla para enviar para o dic de fluxos
                    chave = (ip_src,ip_dst,Port_src,Port_dest,protocol)
```

```

        if not chave in self.fluxos:
            self.fluxos[chave] = [(ts,ip)]
            self.temporizador(chave)
        else:
            self.fluxos[chave].append((ts,ip))
            self.temporizador(chave)

        if isinstance(ip.data,dpkt.tcp.TCP):
            self.cTCP += 1
        elif isinstance(ip.data,dpkt.udp.UDP):
            self.cUDP += 1
        else:
            self.cNonIP += 1
    else:
        break

    print("IP Pkts:"+str(self.cTCP+self.cUDP))
    print("Non IP Pkts:"+str(self.cNonIP))

except:
    self.erro.setError(sys.exc_info()[1],self.nomeColetor)
    print self.erro.getError()

```

Função 2 - Captura de Pacotes

2.1.2 Métricas de Fluxo

Os cálculos estatísticos de cada fluxo são processados pela Ferramenta Storm que será descrito mais detalhadamente na seção 2.5, mas para que ele possa calcular e apresentar os resultados provenientes dos fluxos é preciso antes definimos a sumarização de cada fluxo. A sumarização é o processo de reunir e calcular os valores dos pacotes agrupados em cada fluxo, para isso calcularemos o valor de tamanho, duração e taxa, de cada fluxo pelo qual denominaremos Métricas de Fluxo. A seguir as funções que representa o cálculo das métricas de cada fluxo.

```

def calc_dur(self,pkts): #pkts is the packet list of a flow, and each element is a tuple (ts,pkt)
    first = pkts[0]
    last = pkts[-1]
    dur = last[0] - first[0]
    return dur

```

Função 3 - Métrica Duração

```

def calc_size(self, pkts): #pkts is the packet list of a flow, and each element is a tuple (ts,pkt)
    #print reduce(lambda x,y: x+y, pkts)
    lista=[]
    for i in pkts:
        lista.append(i[1].len)
    return sum(lista)

```

Função 4 - Métrica Tamanho

Para calcular a métrica da taxa é só pegar o tamanho e dividir pela duração. A partir das assinaturas dos protocolos feitas L7filter podemos ter a classificação de cada fluxo de acordo com o seu protocolo, a partir da função *classify* que segue abaixo.

```
def classify(self,pkts):
    for pkt in pkts:
        app = pkt[1].data.data.lower()
        Protocol_found = ""
        for p in self.protocols.items(): #Checar qual o protocolo
            if p[1].search(app):
                Protocol_found = p[0]
        if Protocol_found:
            return Protocol_found
    else:
        return "unknown"
```

Função 5 - Classificação do Fluxo pelo protocolo

Vale ressaltar mais uma vez como dito anteriormente, se os pacotes que forem capturados não for nenhum dos citados na seção 2.1.1, eles serão classificados como desconhecidos. Após todo o processo de captura, classificação e cálculos das métricas, os fluxos são então enviados para as respectivas filas de acordo com o protocolo adequado.

2.1.3 Logs

Os logs são responsáveis por captar todos os erros relacionados à conexão e captura dos pacotes. Eles são armazenados em um arquivo .txt relacionado a cada coletor. A classe responsável pela criação dos Logs é vista a seguir.

```
from datetime import datetime
import os.path

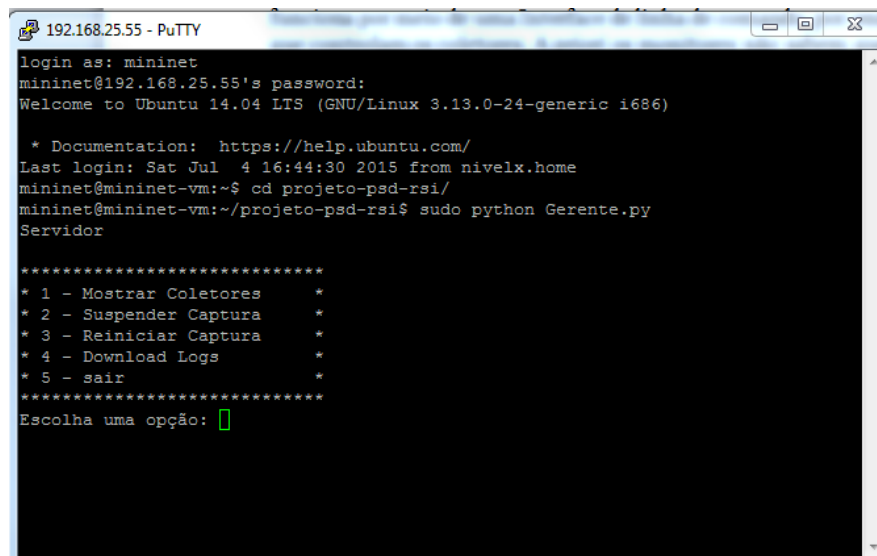
class Logs():
    nome_arq = ""
    def setError(self,mensagem,coletor):
        today = datetime.now()
        self.nome_arq = "logs_"+coletor+".txt"
        arquivo = open(self.nome_arq,"a+")
        arquivo.write(str(today)+" - "+str(mensagem)+"\n")
        arquivo.close

    def getError(self):
        if os.path.isfile(self.nome_arq):
            arquivo = open(self.nome_arq,"r")
            mensagem = arquivo.read()
            arquivo.close
            return mensagem
        else:
            return "arquivo nao encontrado"
```

Função 6 - Classe Log

2.2 Monitor

O Monitor ou Gerente é o responsável por controlar e gerenciar os Coletores. Ele funciona por meio de uma interface de linha de comando, por onde envia os comandos que controlam os coletores. A priori os monitores não sabem quem são os coletores, o descobrimento deles se dá por meio de mensagens em *broadcast* que é lançado na rede via sockets. Após o descobrimento o Monitor está pronto para enviar as mensagens de comando, que informam quando os coletores deverão parar de capturar pacotes, reiniciar a captura de pacotes e fazer o upload dos Logs de erros para o Monitor. Na imagem 2 é visto o menu de ações do Monitor.



```
192.168.25.55 - PuTTY
login as: mininet
mininet@192.168.25.55's password:
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic i686)

 * Documentation:  https://help.ubuntu.com/
Last login: Sat Jul  4 16:44:30 2015 from nivelx.home
mininet@mininet-vm:~$ cd projeto-psd-rsi/
mininet@mininet-vm:~/projeto-psd-rsi$ sudo python Gerente.py
Servidor

*****
* 1 - Mostrar Coletores      *
* 2 - Suspender Captura     *
* 3 - Reiniciar Captura     *
* 4 - Download Logs         *
* 5 - sair                  *
*****
Escolha uma opção: █
```

Figura 2 - Menu de Ações Monitor

2.2.1 Protocolo

Para que o processo de comunicação e recebimento de informações entre o Monitor e Coletores foi necessário à implementação de um simples protocolo, responsável por gerir a padronização da sintaxe, semântica e sincronização da comunicação, além de controlar a transferência de dados entre os dois componentes. Abaixo segue a tabela que representa a sintaxe e semântica adotada para nosso protocolo de comunicação.

Sintaxe	Semântica
Status	Demonstra o status atual de cada coletor (Coletando ou Suspenso).
Suspender	Suspender a captura de pacotes.
Reiniciar	Reiniciar a captura de pacotes.
Download	Faz download dos Logs de Erros dos Coletores.
Identificação	Identifica quais são os coletores da rede.

Tabela 1 - Sintaxe e Semântica do Protocolo

2.3 Comunicação entre Monitor e Coletores

A comunicação entre os Coletores e o Monitor se dar pela comunicação via socket UDP. Através do protocolo citado acima. O UDP ou Protocolo de Datagramas é um protocolo de transferência de pacotes de um hospedeiro para outro, sendo não orientado a conexões, seu objetivo é acelerar o processo de envio dos dados, assim ele se preocupa apenas em enviar os dados ao seu destinatário, sem se preocupar se as mensagens foram realmente recebidas. Portanto o UDP foi escolhido devido ao seu desempenho para trabalhar com *stream* de dados da rede em tempo real.

2.3.1 Socket e SocketError

Como já dito para a comunicação entre os componentes foram utilizados sockets UDP, para desenvolvimento e aplicação do projeto. Sockets são os pontos finais de um canal de comunicação bidirecional. Eles podem se comunicar dentro de um processo, entre processos na mesma máquina, ou até mesmo entre processos em diferentes continentes. A biblioteca *socket* tomada para utilização no projeto fornece classes específicas para a manipulação dos transportes comuns, bem como uma interface genérica para manusear o resto. Abaixo segue um exemplo de socket implementado no projeto.

```
self.AMOUNT_BYTES = 1024
self.BROADCAST_PORT = 9000
self.BROADCAST_LISTEN = ''
self.BROADCAST_SEND = '<broadcast>'
self.bsock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM);
self.bsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
self.bsock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
self.bsock.bind((self.BROADCAST_LISTEN, self.BROADCAST_PORT))
self.bsock.settimeout(2)
```

Função 7 - Exemplo de Socket UDP

Como visto acima usamos a função de socket disponível no módulo socket para criar um objeto de socket. Um objeto de socket é então usado para chamar outras funções para configurar um servidor de socket. Logo após é chamado à função bind (hostname porta) para especificar uma porta para o seu serviço. As funções 'recvfrom' e 'sendto' são as responsáveis por transmitir e receber as mensagens trocadas pelos dois sistemas.

Para emular perdas aleatórias no processo de transmissão confiável, descrito na seção 2.3.2, os coletores foram implementados com a biblioteca socketError, que possui as mesmas funções do módulo principal, mas que contem característica diferenciais, que foram feitas para induzir o erro no processo de transmissão de pacotes. Abaixo encontra-se a implementação da biblioteca socketError.

```

import socket, random, time

class socketError(socket.socket):
    errorProb = 0.0

    def setErrorProb(self, p):
        self.errorProb = float(p)

    def getErrorProb(self):
        return self.errorProb

    def sendWithError(self, s):
        if (self.type == socket.SOCK_DGRAM):
            u = random.random()
            if (u > self.errorProb):
                self.send(s)
        else:
            self.send(s)

    def recvWithError(self, n):
        if (self.type == socket.SOCK_DGRAM):
            data = self.recv(n)
            u = random.random()
            if (u > self.errorProb):
                return data
            else:
                raise socket.timeout
        else:
            return self.recv(n)

```

Função 8 - Implementação SocketError

2.3.2 Transmissão Confiável

Para o desenvolvimento do projeto foi necessário à implementação de um processo de transmissão confiável, já que como já citado há uma transmissão de dados entre os Coletores e o Monitor, no processo de envio e recebimento dos arquivos de Logs dos Coletores. Para isso, o arquivo de log foi dividido em parte proporcionais, para ser enviados como pacotes ao Monitor, sem perdas ou duplicatas de dados. No projeto Consideraremos apenas o caso de transferência unidirecional de dados, ou seja, do lado do remetente para o lado do destinatário.

O processo de transmissão confiável funciona da seguinte forma, o emissor envia o pacote e aguarda pela resposta do receptor. O receptor avisa explicitamente ao emissor que o pacote foi recebido corretamente (ACKs) ou que os pacotes têm erros, e assim faz o emissor retransmitir o pacote. Contudo para se ter um controle maior sobre os dados que estão sendo enviado e evitar duplicatas, o emissor acrescenta um numero de sequencia em cada pacote. Assim caso algum pacote der erro ou tenha sido perdido,

o emissor reenvia o último pacote novamente. O receptor por sua vez descarta (não passa para aplicação) pacotes duplicados.

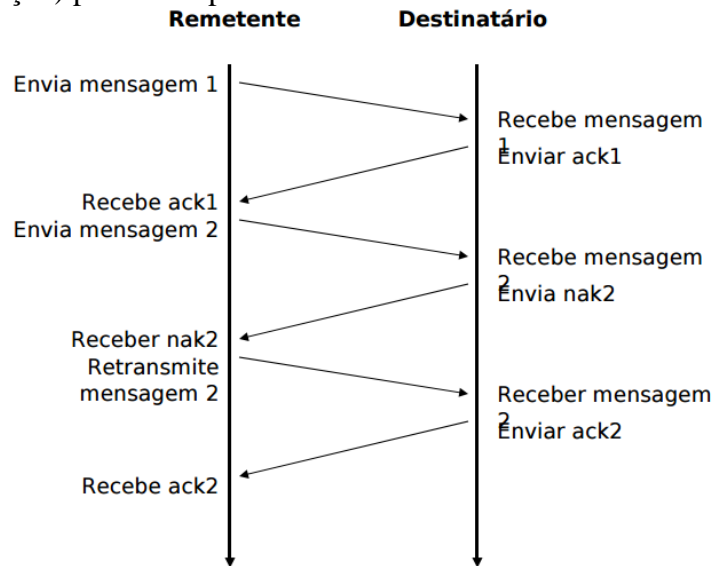


Figura 3 - Processo Transmissão Confiável

2.3.3 Flow de Dados

Os fluxos depois de calculados estão prontos para ser enviados para fila. Pelo processo de envio de tuplas, segue abaixo o modelo de implementação. As tuplas são formadas da seguinte forma, <protocolo,duração,tamanho,taxa> que corresponde a cada fluxo. A duração é em segundos, o tamanho em Kilobytes e a taxa é em Kb/s. O envio é feito pela biblioteca Pika implementa a conexão com o RabbitMQ, para o envio das tuplas na fila que corresponde a cada protocolo.

```

def send_tupla(self,nome_protocolo,msg):

    credentials = pika.PlainCredentials('server', 'server123')
    connection = pika.BlockingConnection(pika.ConnectionParameters(
        'localhost', 5672, 'grupo1', credentials))
    channel = connection.channel()

    channel.exchange_declare(exchange='topic_logs',
                             type='topic')

    routing_key = nome_protocolo if len(msg) > 1 else 'anonymous.info'
    print routing_key
    message = str(msg) or 'Hello World!'
    channel.basic_publish(exchange='topic_logs',
                          routing_key=routing_key,
                          body=message)
    print " [x] Sent %r:%r" % (routing_key, message)
    connection.close()
  
```

Função 9 - Envio da Tupla para fila do RabbitMQ

2.4 RabbitMQ

O RabbitMQ costumeiramente utiliza dois conceitos importantes básicos. O Produtor é o programa que necessariamente irá enviar as mensagens, que neste caso são armazenadas em filas. Cada fila pode armazenar uma quantidade ilimitada de mensagens. O Receptor é o programa que permanece em espera aguardando as mensagens. Para que o Receptor seja capaz de receber as mensagens faz-se necessário executar o código do Receive. O exemplo segue a seguir:

```
Máquina $: python Receive.py  
[ * ] Esperando por mensagens. Para sair pressione CTRL + C
```

Figura 4 Inicializando o Receptor no RabbitMQ em Python

A partir disso, ao executar um código referente ao produtor, será recebido toda e qualquer mensagem enviada por quaisquer produtores que tenham o receptor em questão como destinatário da mensagem e parâmetros definidos.

Para o desenvolvimento do projeto fez-se necessário utilizar a diretiva de tópicos que o RabbitMQ oferece. Esta opção permite que baseado em um tópico específico de cada mensagem enviada seja diferenciado para uma fila diferente. A partir da fila criada, as próximas mensagens da fila irão ser encaminhadas para a fila correspondente ao tópico criado. No arquivo de Receive foi configurado que a cada mensagem recebida é analisada o tipo de tupla a que este pertence. Ou seja, as filas são criadas de acordo com a necessidade da tupla.

2.5 STORM

O Storm é um sistema open-source utilizado na área de sistemas distribuídos em tempo real. O Storm é semelhante ao Hadoop, porém enquanto o primeiro se baseia no conceito de topologias e processa dados ilimitados de forma confiável, em tempo real e continuamente, o Hadoop se baseia no modelo arquitetural MapReduce e realiza o processamento em lotes. Segundo Apache Software Foundation, o Storm é capaz de processar mais de um milhão de tuplas por segundo em cada nó. Isso é possível devido ao fato dele utilizar streaming de dados em paralelo com um cluster.

A estrutura do Storm consiste em nós trabalhadores e nó mestre. O nó mestre executa o Nimbus. Este realiza o monitoramento das falhas no sistema, distribui as tarefas aos nós trabalhadores que executam o Supervisor. Estes iniciam e realizam processamento dos dados que foram enviados pelo Nimbus.

A comunicação entre o Nimbus e o Supervisor é desempenhada pelos nós que executam o Zookeeper. Ele também efetua a coordenação do cluster. Caso um nó trabalhador falhe, por exemplo, o Zookeeper notifica ao Nimbus para que este possa atribuir a tarefa antes destinada ao nó que falhou a outro nó. A Figura 4 ilustra a comunicação entre o Nimbus, Zookeeper e o Supervisor.

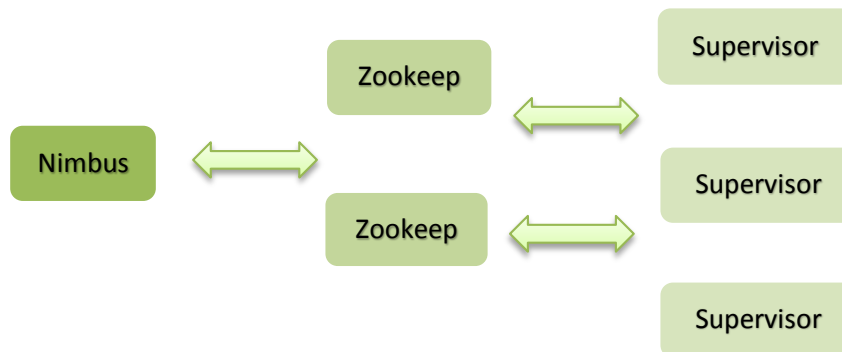


Figura 4- Estrutura do Storm

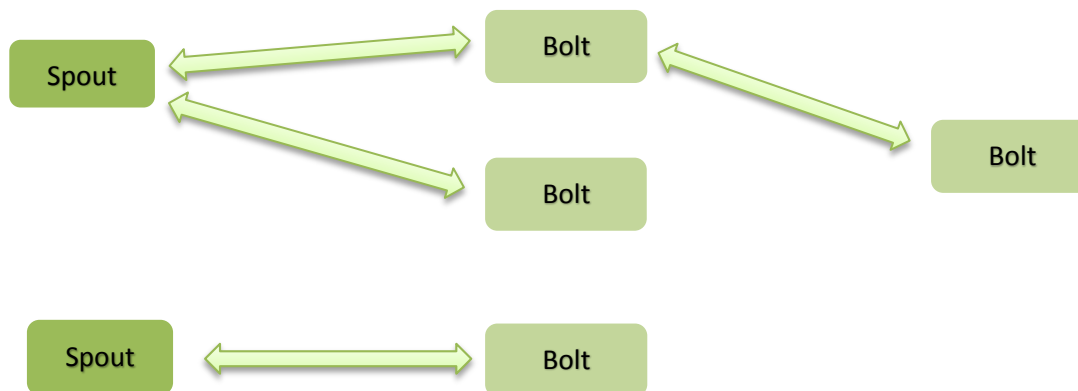


Figura 5 - Topologia spout e bolt

A Figura 5 ilustra a topologia de spouts e bolts. Os spouts emitem o fluxo das tuplas. Um Spout pode conectar-se a uma API, ou como no caso deste projeto recebe os dados do RabbitMQ. Os bolts consomem as tuplas emitidas pelos spouts e realizam o processamento dos dados, tendo a possibilidade de emitir novos fluxos de dados para outros bolts dependendo das informações que se pretende obter dos fluxos. Os fluxos são enviados a partir do RabbitMQ para o Storm.

2.6 Integração RabbiMQ-Storm-Charts4J

A integração do RabbiMQ para o Storm foi implementada através das classes *AMQPRReceiverSpout*, na qual englobam o conceito do receiver do storm, e ao mesmo tempo, fazem a integração com o conceito de spouts do storm, de forma que cada fila terá um spout correspondente na topologia.

Na topologia do Storm implementada, fez-se necessário a criação de sete spouts (uma para cada fila do Rabbit) e três bolts, uma para processamento de tamanho (ratos e elefantes), outro para processar tempo (libélula e tartarugas) e o terceiro para processar fluxo (guepardo e caramujo). A topologia final pode ser representada pela figura 7 abaixo:

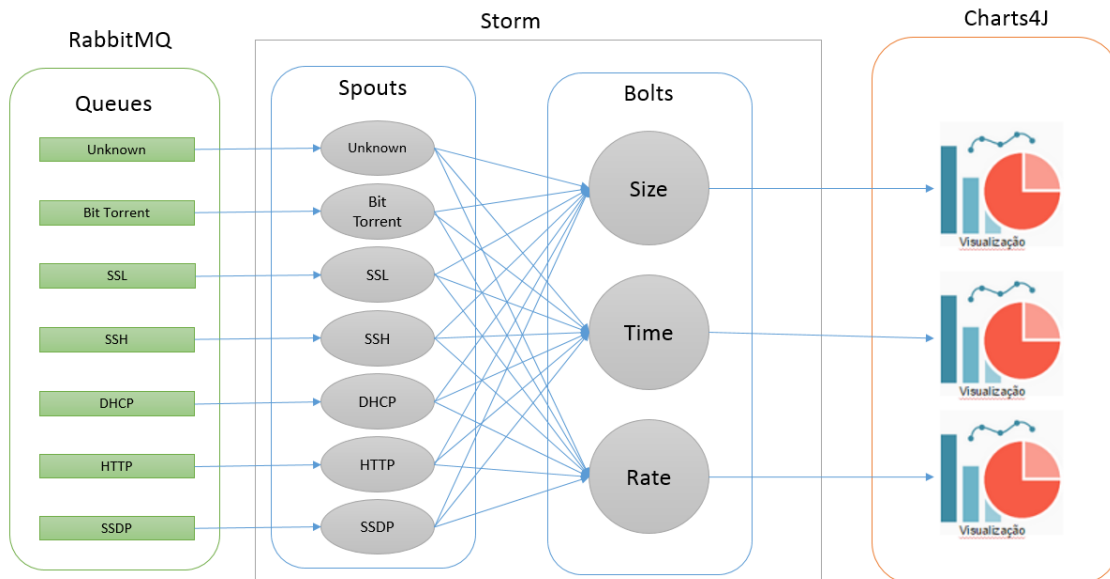


Figure 6 – Representações da Integração

Cada bolt criado é responsável por um processamento diferente, na qual recebe os dados do spout por pacote e realiza as seguintes análises:

Size Bolt: Processa se a tupla recebida se trata de um elefante ou de um rato, ou seja, compara a média de tamanho (kB) até o momento ($\text{TamanhoTotal} / \text{NumeroDePacotes}$) para identificar se tratasse de um elefante (acima da média) ou um rato (abaixo da média).

Time Bolt: Processa se a tupla recebida se trata de uma libélula ou de uma tartaruga. Ou seja, verifica o tempo médio (seg) até o momento ($\text{TempoTotal} / \text{NumeroDePacotes}$) para identificar se tratasse de uma libélula (acima da média) ou uma tartaruga (abaixo da média).

Rate Bolt: Processa se a tupla recebida se trata de um guepardo ou de um caramujo, ou seja, compara a média do fluxo (kB/s) até o momento ($\text{FluxoTotal} / \text{Numero de Pacotes}$) para verificar se tratasse de um guepardo (acima da média) ou de um caramujo (abaixo da média).

Observações:

- A média varia de acordo com o número de tuplas processadas, sendo atualizada sempre que uma nova análise é realizada.
- Para análise inicial, foi estabelecida uma média inicial fixa, e apenas após a verificação do primeiro item, ela passa a ser uma média móvel.
- Por orientação do instrutor, não foi aplicado o desvio padrão para cálculo da média.
- Os valores de média fixa iniciais adotados foram: 15kB para tamanho, 4 segundos para tempo e 500 kB/s para fluxo.

Ao final, todos os dados são processados pela API Chats4J, uma biblioteca criada para integração do Java com o Google Charts. Durante intervalos definidos, o sistema irá coletar os dados processados e plotar os devidos gráficos para cada parâmetro (tamanho, tempo e fluxo), conforme ilustrado na figura 8.

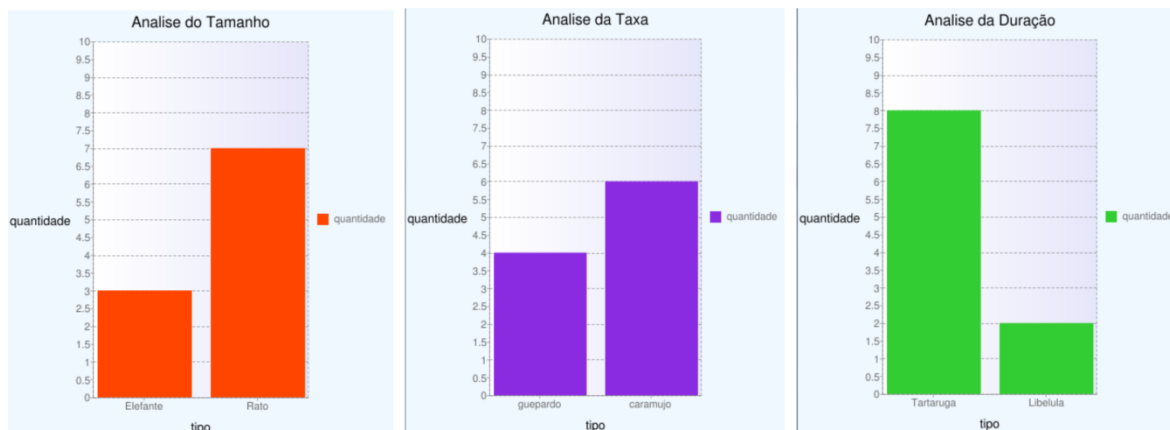


Figura 5 – Exemplo Gráfico de Analise de Tamanho, Taxa e Duração.

3. Principais Dificuldades

Para o desenvolvimento desse projeto o grupo teve dificuldades relacionadas à implementação, entendimento dos requisitos e integração com as ferramentas auxiliares. Que atrapalharam e atrasaram o desenvolvimento e progresso do projeto. Em redes a principal dificuldade foi implementar a transferência confiável e criação dos fluxos de dados. Na parte de sistemas distribuídos a principal dificuldade foi a integração das ferramentas RabbitMQ e Storm, que são as responsáveis pelo armazenamento das filas e processamento dos fluxos, respectivamente. Contudo aprendemos muito na trajetória do projeto, adquirindo experiências e conhecimentos necessários para o entendimento das duas disciplinas.

4. Referências

Apache Storm. Storm Documentation. Disponível em:
<<https://storm.apache.org/documentation/Tutorial.html>> Acesso em 3 de julho de 2015.

Sempre Update. Apache Storm está pronto para estrear. Disponível em:
<<http://sempreupdate.org/apache-storm-esta-pronto-para-estrear/>>. Acesso em 5 de julho de 2015.

Charts4J: Let the computer in the cloud build your charts. Disponível em:
<<https://code.google.com/p/charts4j/>>. Acesso em 5 de julho de 2015.

JavaWorld. Open Source Java Projects: Storm. Disponível em:
<<http://www.javaworld.com/article/2078672/big-data/open-source-tools-open-source-java-projects-storm.html?page=2>>. Acesso em 5 de julho de 2015.

Microsoft Azure. Introdução ao Apache Storm no HDInsight: análise em tempo real para o Hadoop. Disponível em: < <https://azure.microsoft.com/pt-br/documentation/articles/hdinsight-storm-overview/>> Acesso em 5 de julho de 2015.

RabbitMQ. Get Started. Disponível em: <<https://www.rabbitmq.com/getstarted.html>>. Acesso em 5 de julho de 2015.

HEIDEMANN J. ; LAN K.C. A measurement study of correlations of Internet flow characteristics, United States, 2005.

Programming with pcap Disponível em: < <http://www.tcpdump.org/pcap.html> >. Acesso em 3 de julho de 2015.

Dpkt Api - Disponível em: < <https://dpkt.readthedocs.org/en/latest/>>. Acesso em 3 de julho de 2015.

Low-level networking interface - Disponível em: < <https://docs.python.org/2/library/socket.html>>. Acesso em 3 de julho de 2015.

VAINGAST S. Beginning Python Visualization, Editora: Apress, 2009.

SAXENA S. Real-time Analytics with Storm and Cassandra, Editora: Packt, 2015.