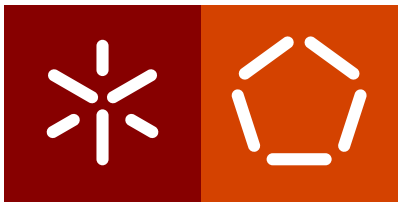
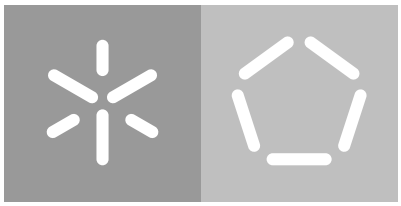


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ricardo César Cangueiro Mendonça

Um Sistema P2P para Detecção de Anomalias de Rede

Maio 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ricardo César Cangueiro Mendonça

Um Sistema P2P para Detecção de Anomalias de Rede

Dissertação de Mestrado
Mestrado em Engenharia Informática

Dissertação efetuada sob a orientação do
Professor Pedro Nuno Sousa

Maio 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

AGRADECIMENTOS

Esta dissertação resultou de um longo trabalho e dedicação. Contudo, não seria possível sem a ajuda e os conselhos que me foram dados durante todo o trabalho.

Quero agradecer a disponibilidade e orientação académica do Professor Doutor Pedro Nuno Sousa. Também agradeço ao meu pai, Nilton Mendonça e à minha namorada, Juliana Cortez, por todo o apoio que me deram ao longo deste percurso.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração. Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

RESUMO

Uma monitorização precisa e eficiente é vital para garantir que uma rede funcione de acordo com o comportamento pretendido, tal como atuar rapidamente face aos problemas encontrados. A tarefa de monitorização de uma rede torna-se complexa com o aumento do tamanho e heterogeneidade da rede.

Este trabalho tem como objetivo continuar o desenvolvimento de um sistema P2P para deteção de anomalias de rede, a fim de ajudar os administradores de redes no processo de monitorização. O sistema deverá ser facilmente utilizado por administradores de redes.

Inicialmente, será apresentado o trabalho de investigação do estado da arte, onde estarão destacadas várias tecnologias relacionadas com o objetivo desta dissertação. Expor-se-ão vários requisitos de sistemas de monitorização de redes e descrita a aplicabilidade de um sistema P2P neste tipo de sistemas. Também serão apresentados e comparados alguns projetos relacionados com o tema e objetivos deste projeto.

Posteriormente, irão ser descritos todos os processos de criação do sistema e especificadas as entidades existentes e como é efetuada a sua comunicação, sendo apresentadas as várias formas de interação do administrador com o sistema. De seguida expor-se-á o processo de definição de uma metalinguagem com o objetivo de possibilitar aos administradores a configuração e pré-programação da rede de forma eficaz e variada. Descrever-se-á o processo de otimização do sistema, com o objetivo de reduzir o tráfego gerado pela rede P2P. Será também descrito como o sistema se tornou tolerante a falhas, recuperando o estado caso alguma entidade do sistema tenha um problema. Depois, expor-se-á a implementação de cada um dos mecanismos desenvolvidos e da arquitetura do sistema, onde estarão referidas as tecnologias utilizadas e justificados certos caminhos escolhidos para alcançar os objetivos da dissertação.

O sistema será testado num emulador de redes e os dados resultantes dos mecanismos criados serão analisados a fim de validar o correto funcionamento do que foi desenvolvido.

Palavras-Chave: P2P, monitorização de redes

ABSTRACT

Precise and efficient monitoring is vital to ensure that a network works according to the intended behavior, as well as quickly acting to the problems found. The task of monitoring a network becomes complex with increasing network size and heterogeneity. The available network monitoring and management solutions are not only costly but also difficult to use, configure and maintain.

This work aims to continue the development of a P2P system to detect network anomalies to help network administrators. The system should be easily used by ISP network administrators.

Initially, state-of-the-art research will be presented, where various technologies related to the objective of the dissertation will be highlighted. Multiple requirements of monitorization systems will be exposed and there will be a description of the applicability of a P2P system in a monitorization system.

Afterward, all the processes of creation of the system and the existing entities will be described as well as their communication capabilities. All the ways an administrator can interact with the system will also be presented. Then, the process of definition of a metalanguage will be exposed, to allow the administrators to configure and pre-program the network in an effective and varied manner. The process of optimization of the monitoring system will be described, to reduce the traffic of the P2P network. It will be also described how the system became fault-tolerant, recovering its state if any entity has a problem. Subsequently, the implementation of each one of the developed mechanisms and the architecture of the system will be exposed, where it's explained which technologies were used and justified some paths chosen to achieve the dissertation objectives.

The system will be tested in a network emulator and the resulting data from the created mechanisms will be analyzed to validate the correct behavior of what was developed.

Keywords: P2P, network monitoring

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

SIGLAS

A

API Application Progamming Interface.

AST Abstract syntax tree.

B

BNF Backus-Naur form.

C

CDN Content Delivery Networks.

CLI Command-Line Interface.

CORE Common Open Research Emulator.

CPU Central Processing Unit.

CSS Cascading Style Sheets.

D

DDOS Distributed Denial-of-Service.

DHT Distributed Hash Table.

DNS Domain Name System.

G

GUI Graphical User Interface.

H

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

I

ICMP Internet Control Message Protocol.

IP Internet Protocol.

ISP Internet Service Provider.

M

MIB Management Information Base.

MTR My Traceroute.

N

NMS Network Management Station.

NTP Network Time Protocol.

O

OSPF Open Shortest Path First.

OWD One Way Delay.

P

P2P Peer-to-Peer.

PDU Protocol Data Unit.

PRTG Paessler Router Traffic Grapher.

Q

QOS Quality of Service.

R

RIP Routing Information Protocol.

RON Resilient Overlay Network.

RTT Round Trip Time.

S

SMTP Simple Mail Transfer Protocol.

SNMP Simple Network Management Protocol.

SSL Secure Sockets Layer.

T

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TTL Time To Live.

U

UDP User Datagram Protocol.

UTP Unshielded twisted pair.

V

VOIP Voice Over IP.

1

INTRODUÇÃO

1.1 ENQUADRAMENTO E MOTIVAÇÃO

As redes de comunicação deparam-se algumas vezes com anomalias que afetam o serviço oferecido aos utilizadores finais. Algumas dessas anomalias relacionam-se com falhas temporárias ou permanentes de equipamento físico da rede (e.g. *links*, *routers*, etc.) que implicam uma degradação do nível de serviço oferecido. Outras anomalias resultam de situações de congestão originada por níveis excessivos de tráfego em alguns pontos críticos da rede. A prática atual de monitorização de rede depende, em grande parte, de operações manuais e, assim, as empresas gastam uma parte significativa dos seus orçamentos na parte que monitoriza as suas redes [8]. As soluções de monitorização e gestão de rede disponíveis não são apenas caras, mas também difíceis de usar, configurar e manter [9]. Este projeto visa a especificação e desenvolvimento de um sistema *Peer-to-Peer (P2P)* que possibilite a deteção de anomalias de rede, podendo ser usado pelos administradores dos *Internet Service Provider (ISP)*. Existem vários esquemas diferentes para criar uma rede *overlay* P2P, sendo este centralizado [10]. O sistema será composto por um conjunto *peers* distribuídos na rede e um nó coordenador responsável pela gestão da rede P2P, pela coleta de informação de anomalias ocorridas na rede do ISP, e por disponibilizar uma interface com o administrador onde poderá iniciar a monitorização em *links* de uma rede com alguns tipos de medições, podendo alterar vários parâmetros. A rede P2P deverá disponibilizar alarmes ao administrador para situações tais como: falta de conectividade em determinadas *paths edge-to-edge*; situações de alteração de rotas; degradação crítica do atraso, perdas de pacotes, congestão, etc. Serão explorados com especial foco os seguintes aspetos relativos à operação da rede *overlay* de monitorização:

- Redução do tráfego gerado pela rede P2P através da composição dos resultados obtidos por diferentes *peers* (e.g. possibilidade de agregação/composição dos resultados obtidos por diferentes *peers* para obter informação relativa a uma determinada *path*);
- Estratégias de programação da rede P2P para tomar determinadas ações autónomas e possibilitar a adaptação a diferentes cenários;

- Procedimentos para tornar o sistema tolerante a falhas, onde serão contempladas situações de injeção de falhas (permanentes ou temporárias) nos *links*/*routers* da topologia e análise de resultados.

1.2 OBJETIVOS

Este projeto centra-se no desenvolvimento de uma rede *overlay* para monitorização e detecção de anomalias da rede, que fornecerá em tempo real ao administrador do ISP um sumário dos eventos anómalos e quando estes ocorreram. Como ponto de partida do projeto será utilizado um protótipo com algumas funcionalidades básicas já implementadas. A fim de atingir os objetivos do projeto, algumas tarefas de trabalho serão:

- Investigação preliminar nas áreas gerais das redes *overlay*/P2P e em questões relacionadas com monitorização da rede;
- Definição da arquitetura geral e das regras de operação da rede P2P;
- Desenvolvimento/adaptação de um protótipo da rede P2P;
- Implementação e definição de uma metalinguagem que permite a programação da rede P2P. Otimização do tráfego gerado pelo sistema e torná-lo tolerante a falhas;
- Demonstração do protótipo e das funcionalidades definidas num emulador de redes.

1.3 ESTRUTURA DA DISSERTAÇÃO

Este documento é composto por seis capítulos que estão organizados da seguinte forma:

- Introdução: neste capítulo foi feito o enquadramento e contextualização do trabalho, onde é exposto o tema geral, os seus objetivos e a motivação.
- Estado da Arte: base teórica necessária para desenvolver o sistema. São descritas tecnologias de monitorização de rede existentes sendo referidas algumas ferramentas e metodologias. Posteriormente neste capítulo são apresentados vários requisitos de sistemas de monitorização de redes e de seguida são descritos redes e sistemas P2P. Por fim, são apresentados alguns trabalhos relacionados.
- Arquitetura do Sistema e Mecanismos Desenvolvidos: Neste capítulo é detalhada a arquitetura do sistema, especificando os vários componentes existentes nas diferentes entidades que serão apresentadas, posteriormente a comunicação entre estas entidades será exposta. Serão também descritos os passos para a criação de uma metalinguagem para ajudar o administrador a pré-programar a rede e iniciar *probings* mais

rapidamente, como também todos os casos que ocorreram na otimização do sistema e como o sistema se tornou tolerante a falhas.

- Implementação do Sistema Desenvolvido: Neste capítulo será descrito o processo de criação das várias entidades e sua comunicação, da metalinguagem, da otimização do sistema e da transformação do sistema para se tornar tolerante a falhas. Também serão descritas que ferramentas foram utilizadas na implementação do sistema e a razão da sua escolha.

- Análise de resultados: Neste capítulo é utilizado um emulador de redes para testar os mecanismos desenvolvidos, onde serão analisados os dados resultantes a fim de validar o correto funcionamento do que foi desenvolvido.

- Conclusões: Por último, será feito um resumo do trabalho desenvolvido, onde serão apresentadas as dificuldades encontradas e feita uma análise dos capítulos e principais contribuições da dissertação. Serão também apresentadas algumas possíveis melhorias do sistema para trabalho futuro.

2

ESTADO DA ARTE

Neste capítulo serão descritas algumas tecnologias de monitorização de rede existentes na secção 2.1, sendo detalhadas ferramentas e metodologias utilizadas. Posteriormente serão enunciados requisitos de sistemas de monitorização de redes na secção 2.2, com foco nas necessidades dos ISPs, de seguida serão descritas redes e sistemas P2P na secção 2.3. Por fim, serão apresentados alguns trabalhos relacionados com este projeto na secção 2.4.

2.1 TECNOLOGIAS DE MONITORIZAÇÃO

Nesta secção será feita uma descrição de algumas tecnologias de monitorização como *packet probing*, *packet sniffing*, *Simple Network Management Protocol (SNMP)* e monitorização por *flows*. Também serão abordadas ferramentas, metodologias e aplicabilidade de cada uma.

2.1.1 *Packet Probing*

Packet probing é uma técnica de monitorização de redes que permite desenvolver soluções eficazes para localizar falhas de redes [11]. Através de um emissor são enviados pacotes de sonda que atravessam uma rede sendo estes recebidos num recetor, posteriormente poderão ser feitas observações sobre o comportamento dos pacotes, sendo esta uma medição ativa. A Figura 2.1 ilustra o funcionamento de *packet probing*. Para efetuar este tipo de medição

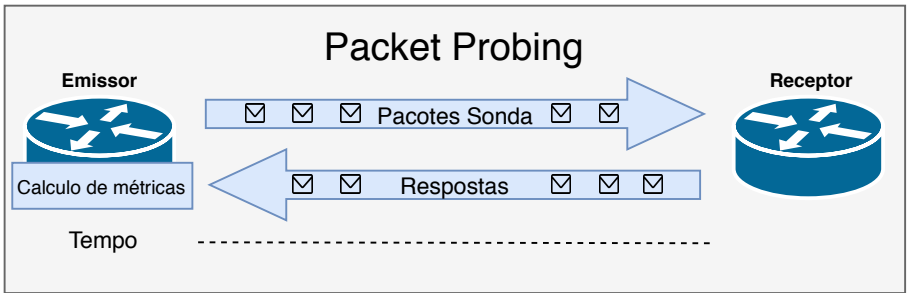


Figura 2.1: Exemplo de *Packet Probing*.

não é necessário acesso total a recursos da rede como *routers* e é uma estratégia possível de usar em situações onde não é possível selecionar os pontos de captura livremente. Quando é feita captura de informação não se captura nenhuma informação privada, não havendo problemas com privacidade. As medições podem ser efetuadas a qualquer altura, quando necessárias.

Os pacotes que são gerados são artificialmente criados quando é necessário fazer um *probing*, contendo normalmente só bits aleatórios no *payload* dos pacotes, resultando numa injeção de tráfego adicional na rede que causa carga adicional. Para ter visibilidade suficiente na rede é preciso ter vários *probes* em localizações especificas. Outro ponto a ter em conta com este tipo de medição é o tráfego artificial que está a ser injetado, nem sempre reflete o comportamento que iria ter no mundo real, pois ao usar protocolos como *Internet Control Message Protocol (ICMP)* e *User Datagram Protocol (UDP)*, estes, podem ser colocados em prioridade mais baixas [12].

Como exemplo de ferramentas de sondas de pacotes existe o *ping*, que ao enviar mensagens ICMP a outro computador via *Internet Protocol (IP)* verifica a conectividade a nível IP, permite saber o *Round Trip Time (RTT)* e conectividade de dispositivos. O *Traceroute* permite saber a rota que os pacotes tomaram e verificar se houve alguma mudança na rede, enviando uma sequência de pacotes UDP incrementando o *Time To Live (TTL)* a uma porta no destino, implementado por Van Jacobson em 1988 [13]. A ferramenta *iperf*, permite verificar a velocidade da conexão enviando vários pacotes com tamanhos de pacotes, *timings*, *buffers* e protocolos diferentes [14].

2.1.2 *Packet Sniffing*

O *Packet sniffing* é normalmente utilizado por administradores de redes para monitorizar e validar o tráfico de uma rede, podendo também ser utilizado para intenções maliciosas como por *hackers*, podendo estes ficar à escuta numa rede por exemplo [15].

Funciona intercetando tráfego que passa numa rede através de um *software* ou *hardware*, cada pacote é capturado e vários campos são expostos que podem ser utilizados para retirar várias informações relevantes, a Figura 2.2 ilustra este tipo de medição.

Este tipo de medição é mais indicado para situações onde os pontos de captura podem ser livremente selecionados, permitindo o tráfego ser capturado em qualquer ponto [16]. Esta técnica coleciona grandes volumes de dados que podem ser analisados para obter vários tipos de informação.

Ao contrário da medição ativa não injeta nenhum tráfego adicional na rede e adiciona muito pouco *overhead* ao *hardware* da rede, a Figura 2.3 ilustra este ponto.

De seguida, serão detalhadas duas aplicações que utilizam *packet sniffing*, *Wireshark* e *Paessler Router Traffic Grapher (PRTG)*.

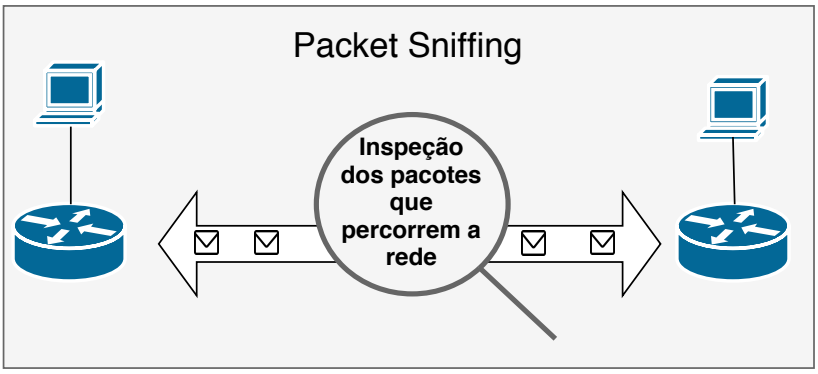


Figura 2.2: Exemplo de *Packet Sniffing*.

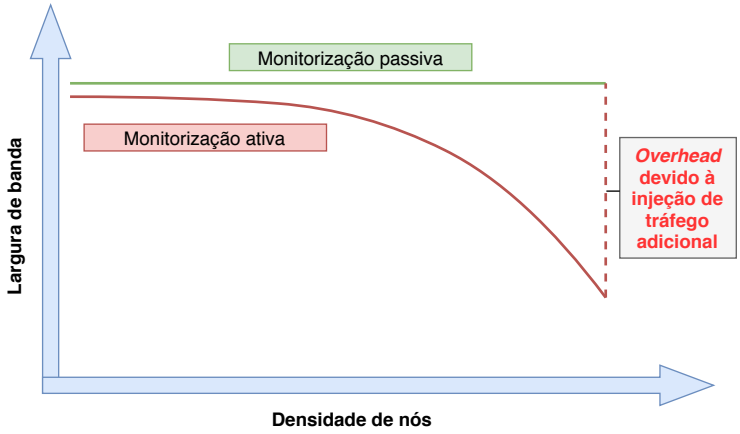


Figura 2.3: Ilustração do *overhead* que a monitorização ativa causa.

O *Wireshark*, é *open-source* e está disponível gratuitamente. Tem a funcionalidade de escutar interfaces de rede locais usando a *Application Progamming Interface (API) pcap*, permitindo analisar pacotes que passem nessas interfaces, sendo usado por: administradores de redes para verificar problemas de redes, engenheiros de redes para verificarem problemas de segurança, programadores para fazer *debug* a implementações de protocolos e aprendizagem [17]. Inicialmente era chamado *Ethereal* e foi lançado em 1998, sendo posteriormente mudado para o nome atual em Maio de 2006. Uma ferramenta semelhante ao *Wireshark* é o *tcpdump* onde não existe interface gráfica.

O PRTG é um software de monitorização sem agentes da *Paessler AG* [18]. Pode monitorizar e classificar condições de sistemas como a utilização da largura de banda e o tempo de utilização sem falha de dispositivo por exemplo, e permite colecionar estatísticas de *switches routers* e servidores. Tem várias ferramentas

que possibilitam a monitorização da rede, sendo estas ferramentas separadas por módulos chamados sensores, incluindo um sensor *packet sniffer*.

2.1.3 *SNMP*

O protocolo simples de gestão de rede (SNMP), lançado em 1988, pertence à camada de aplicação, e é maioritariamente utilizado em sistemas de administração de rede para monitorizar o comportamento de vários dispositivos. Plataformas que integrem este protocolo têm a possibilidade de obter dados e estatísticas dos dispositivos da rede, como o *throughput*, tempo de resposta e utilização do *Central Processing Unit (CPU)*/Memória dos dispositivos da rede [19]. Também têm a possibilidade de mostrar alertas. Os administradores têm a capacidade de decidir que parâmetros querem monitorizar e como a informação é mostrada. O SNMP pode ser utilizado em redes de qualquer tamanho, mas onde se tira melhor partido é em redes de larga escala.

Existem quatro componentes principais numa rede gerida por SNMP, o agente SNMP, os dispositivos a serem geridos (podem ser *routers*, *switches*, impressoras, etc.), o gerente SNMP denominado por *Network Management Station (NMS)* e a *Management Information Base (MIB)* [20] [21].

O agente SNMP corre no *hardware* ou serviço a ser monitorizado, colecionando vários dados e estatísticas dos dispositivos, enviando esta informação quando for feito um pedido pelo NMS, o agente também pode pro-ativamente notificar o NMS se ocorrer um erro.

O NMS é o nó central onde os agentes enviam a informação que colecionaram, pede ativamente aos agentes por informação.

Os NMSs comunicam com os agentes através da MIB, que é uma coleção de informação que é organizada hierarquicamente e composta por objetos geridos, identificados por identificadores de objeto. Como ilustração do funcionamento do protocolo SNMP ver a Figura 2.4.

Como exemplos de aplicações que utilizam o protocolo SNMP existe *Nagios* e *Ntop*.

Nagios®

Feito para correr no sistema operativo *Linux*, o *Nagios* é uma aplicação popular de monitorização de rede de código aberto que pode monitorizar dispositivos que corram os sistemas operativos *Linux*, *Windows* e *Unix*. Tem uma arquitetura modular que permite os utilizadores desenvolverem módulos personalizados para melhorar as funcionalidades do sistemas de várias formas [22].

O *Nagios*, que originalmente tinha o nome de *NetSaint*, foi criado por Ethan Galstad. Este ainda faz a manutenção da aplicação, juntamente com uma equipa de programadores.

Os parâmetros críticos dos recursos de aplicação, rede e servidores são periodicamente verificados pelo *Nagios*. Como exemplo disso, esta aplicação pode monitorizar o número

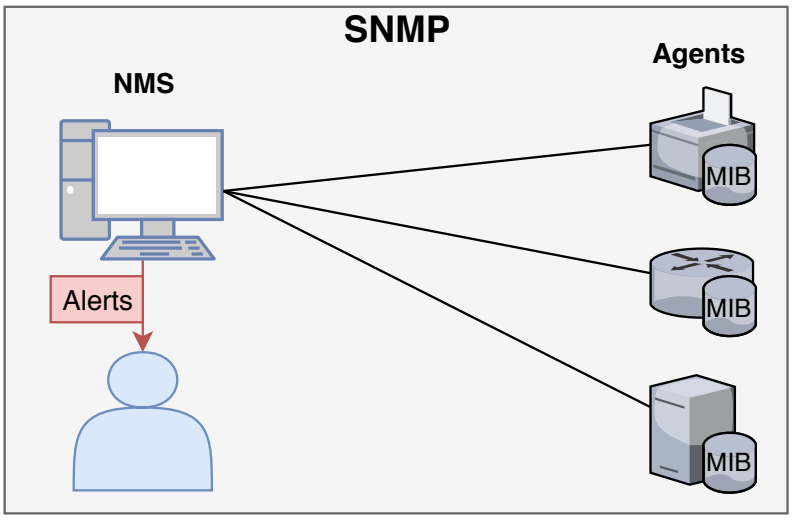


Figura 2.4: Ilustração do SNMP.

de processos a decorrer, bem como os ficheiros de registos, a utilização de disco e de memória. Pode monitorizar também serviços que utilizam protocolos como o *Hypertext Transfer Protocol (HTTP)* e o *Simple Mail Transfer Protocol (SMTP)*.

Em algumas versões do *Nagios*, os utilizadores podem escolher entre trabalhar com uma *Graphical User Interface (GUI)* ou uma *Command-Line Interface (CLI)*. O painel de controlo fornece uma visão geral sobre os parâmetros críticos e, baseado nisso, se forem alcançados níveis críticos é enviado um alerta que pode ser por mensagem ou por email.

O *Ntop* é uma ferramenta *open source* para monitorizar e gerir redes de computadores. Além de ter muitos recursos que demonstram tudo através de gráficos e informações detalhadas que permitem com que haja interação entre utilizadores, também tem suporte para ser executado em vários sistemas operativos e monitoriza e gera relatórios sobre o tráfego e suporte dos *hosts* por vários protocolos. Normalmente as bibliotecas para captura de pacotes oferecem pequenos *buffers* internos, fazendo com que as aplicações não sejam capazes de lidar com tráfego em *bursts*. Para resolver esse problema e reduzir a perda de pacotes, o *Ntop* faz *buffer* aos pacotes capturados [23]. O *Ntop* monitoriza e gera relatórios sobre o tráfego e suporte dos *hosts* pelos seguintes protocolos: *Transmission Control Protocol (TCP)*/UDP/ICMP, *(R)ARP, IPX, DLC, DECnet, AppleTalk, Netbios* TCP/UDP [24].

Alguns dos principais objetivos do *Ntop* são:

- O resultado da análise da rede deverá ser fácil de ler e rico em conteúdo;

- Capacidade de apresentar dados tanto num terminal baseado em caracteres como num *web browser*;
- Portabilidade entre plataformas baseadas e não baseadas em *Unix*;
- Capacidade de monitorizar e gerir a rede remotamente sem a necessidade de recorrer a outras aplicações para obter informação sobre o tráfego;

2.1.4 *Monitorização por flows*

A monitorização por *flows* é primariamente utilizada para monitorizar tráfego em redes de alta velocidade adicionando pouco *overhead* [1]. Gera informação como o endereço de IP dos emissores e recetores, as portas que comunicaram, a data em que a conversação ocorreu, o tempo que demorou e a informação que foi transferida. Comparando com monitorização por pacotes é normalmente mais escalável pois a análise é por *flows* em vez de ser por pacotes. Um *flow* é um conjunto de pacotes IP que passam por um ponto de observação na rede durante um certo intervalo de tempo, onde todos os pacotes têm um conjunto de propriedades em comum [25]. Estas propriedades podem incluir campos de cabeçalho de pacote, como endereços IP de origem e destino, números de porta, conteúdo dos pacotes e meta-informação. A maior parte dos *routers* oferecem a capacidade de coletar estes *flows* para análise. É até possível monitorizar a utilização da rede com nenhum custo de medição.

Há várias formas de configurar monitorização por *flows*, a Figura 2.5 ilustra algumas formas de configuração, onde a secção IV + V trata da observação de pacotes, medição de *flow* e exportação, estas etapas normalmente são combinadas num único dispositivo, a secção VI trata da coleção de dados e a secção VII trata da análise dos dados obtidos, podendo ser feita uma análise manual ou automática.

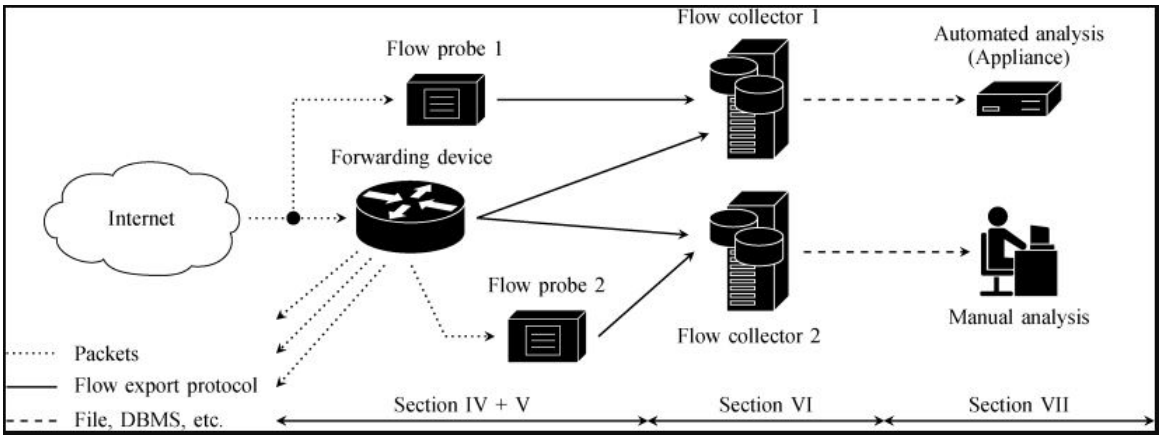


Figura 2.5: Ilustração de monitorização por *flows* [1].

Normalmente as ferramentas de monitorização de *flows* de rede conseguem mapear tendências históricas para planeamento da capacidade de uma rede e também pro-ativamente identificar problemas de segurança. Estas ferramentas têm a capacidade de:

- Monitorização em tempo-real da largura de banda e mapear tendências históricas dos utilizadores, permitindo aos administradores identificar interfaces, ligações, aplicações, utilizadores e protocolos que estejam a usar a largura de banda.
- Permitir aplicar políticas de *Quality of Service (QoS)*. Por defeito cada canal da rede opera dando a cada aplicação a mesma prioridade, seja uma chamada *Voice Over IP (VoIP)* importante ou um vídeo. Estas políticas garantem que aplicações críticas tenham os requisitos suficientes para serem utilizadas corretamente.
- Identificar tendências históricas, analisando padrões de tráfego e utilização sobre um certo período de tempo, estas ferramentas conseguem identificar tendências na largura de banda e potenciais *bottlenecks*. A informação histórica pode ajudar os administradores a planear a capacidade das redes.
- Identificar utilização anormal da largura de banda, ao monitorizar em tempo-real a utilização da largura de banda e as tendências históricas da largura de banda, estas ferramentas podem pro-ativamente identificar problemas de segurança como ataques *Distributed Denial-of-Service (DDoS)*, *downloads* não autorizados e outros potenciais maliciosos comportamentos na rede.

Os protocolos bastante utilizados para monitorização por *flows* são *NetFlow*, *sFlow*, *JFlow*, que são suportados em *switches* e *routers* de vendedores como Cisco, HP, etc [26][27].

2.2 Requisitos de Sistemas de Monitorização de Redes

Os sistemas de monitorização de redes têm vários requisitos a serem alcançados para garantir um sistema robusto e com várias funcionalidades para os administradores manterem as suas redes em bom estado. É necessário ter em mente o que os clientes atualmente mais precisam. Estes requisitos serão descritos em mais detalhe nas próximas secções, começando nas métricas, passando para funcionalidades e recursos que sejam precisos.

2.2.1 *Métricas*

Em sistemas de monitorização de redes existem várias métricas importantes para manter uma rede em bom estado e ter noção do funcionamento geral. Saber o valor de algumas dessas métricas é relativamente simples, mas noutras são necessários algoritmos mais sofisticados e mudanças nas infraestruturas. O *ping* é normalmente utilizado para medir

algumas dessas métricas utilizando o protocolo ICMP, outro protocolo também bastante utilizado para o mesmo fim é o UDP. O *router* pode dar pouca prioridade a pedidos ICMP resultando em resultados que podem não representar a realidade pois o tempo de resposta é afetado.

De seguida apresenta-se uma listagem de métricas importantes a serem recolhidas com uma breve explicação do funcionamento e captura de cada uma:

- O RTT é o tempo que é necessário para um pacote percorrer de uma origem até um destino mais o tempo de volta, é uma métrica relativamente fácil de obter, o *One Way Delay (OWD)* é só o tempo de ida, sendo mais difícil de obter este valor pois há bastantes desafios relacionados com a sincronização de relógios entre entidades diferentes e na existência de *links* assimétricos. O atraso é uma métrica com bastante importância pois afeta a qualidade de vários serviços existentes na Internet.

- *Jitter* também conhecido por variação no atraso, é importante para aplicações como VoIP, pois uma variação muito grande no atraso dos pacotes produz uma receção não regular dos pacotes o que causa distorção do som ou vídeo, mesmo que as outras métricas estejam em bom estado. Pode ocorrer quando os pacotes tomam rotas diferentes ou sofrem balanceamento de carga. Uma maneira de melhorar problemas relacionados com *jitter* é criar um *buffer* na recepção dos dados.

- Perda de pacotes, quando um ou mais pacotes falham a chegar ao destino ocorre perda de pacotes. O que pode fazer com que uma conexão fique lenta, com interrupções ou perder a conexão completamente. Ocorre quando existe congestão na rede, problemas com *hardware* ou *bugs* de *software*, podendo ser amenizado detetando rapidamente que existe perda de pacotes e voltar a enviar os pacotes perdidos, prevenindo degradação da qualidade.

- *Throughput* geralmente definido como a quantidade de dados transferidos que um *hop* ou *path* pode fornecer num determinado espaço de tempo, podendo ser afetado por diversos fatores como latência e protocolo a ser utilizado. O *Throughtput* é bastante importante para medir a capacidade e utilização dos recursos das ligações. Influencia bastante a qualidade de vários serviços existentes na Internet, como *download* de vídeos ou *upload*. É uma das métricas mais importantes que deve ser mantida em boa qualidade pelos administradores dos ISPs.

- Largura de Banda, é a capacidade máxima de transmissão de dados de um emissor a um recetor em um determinado tempo, ao contrário do *Throughput*, não entram os fatores que possam afetar a qualidade do *link*.

  A Figura 2.6 ilustra algumas destas métricas e como são medidas.

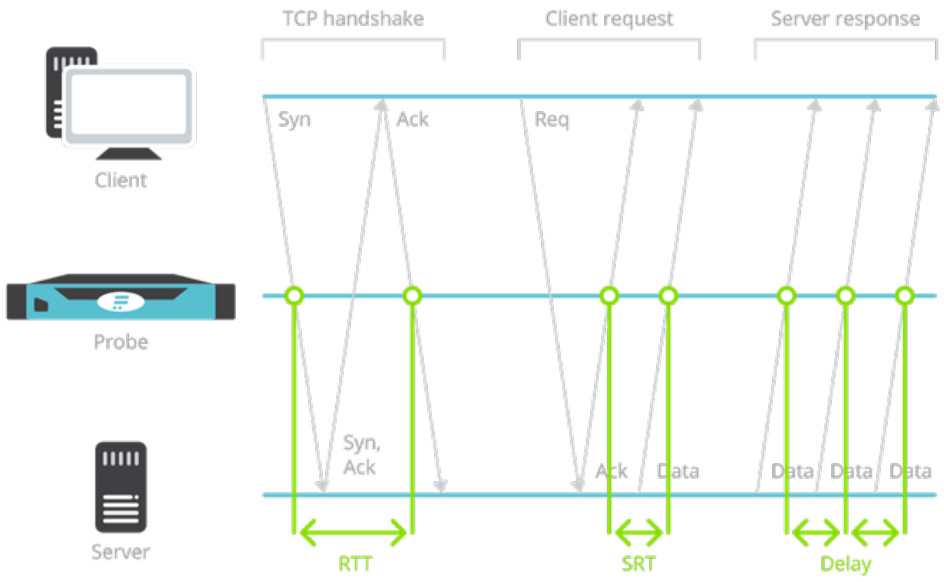


Figura 2.6: Ilustração da medição de algumas métricas [2].

2.2.2 *Funcionalidades*

Ao desenvolver um sistema de monitorização é necessário ter em conta várias funcionalidades que se podem implementar para tornar a monitorização aos utilizadores mais fácil e útil. De seguida serão apresentadas várias funcionalidades importantes a considerar num sistema de monitorização.

Os alertas são uma das funcionalidades mais importantes para um sistema de monitorização de redes. Com estes, os administradores de redes conseguem rapidamente saber se houve algum problema, localiza-lo e potencialmente corrigi-lo. Um sistema de alarmes deve fornecer várias funcionalidades, uma delas é a possibilidade de configuração de perfis de alarme para diferentes administradores, também devem permitir escolha de prioridade, para garantir que os alarmes mais importantes são recebidos mais rapidamente. Uma parte importante ao configurar um sistema de alarmes é existirem vários canais de notificação ao administrador, por e-mail, sms, irc, etc. em vez de ser só por consola. Um administrador também deverá poder configurar diferentes perfis de alarme para datas distintas, podendo ser configurado só receber alarmes de alta prioridade no fim de semana por exemplo. O isolamento das falhas também é crucial para rapidamente se corrigirem os problemas que os alarmes reportam. Uma arquitetura recomendada é ilustrada na Figura 2.7, onde é apresentado um sistema de alarme que faz a análise dos resultados de vários monitores, sendo um administrador notificado através de um sistema de notificações [3].

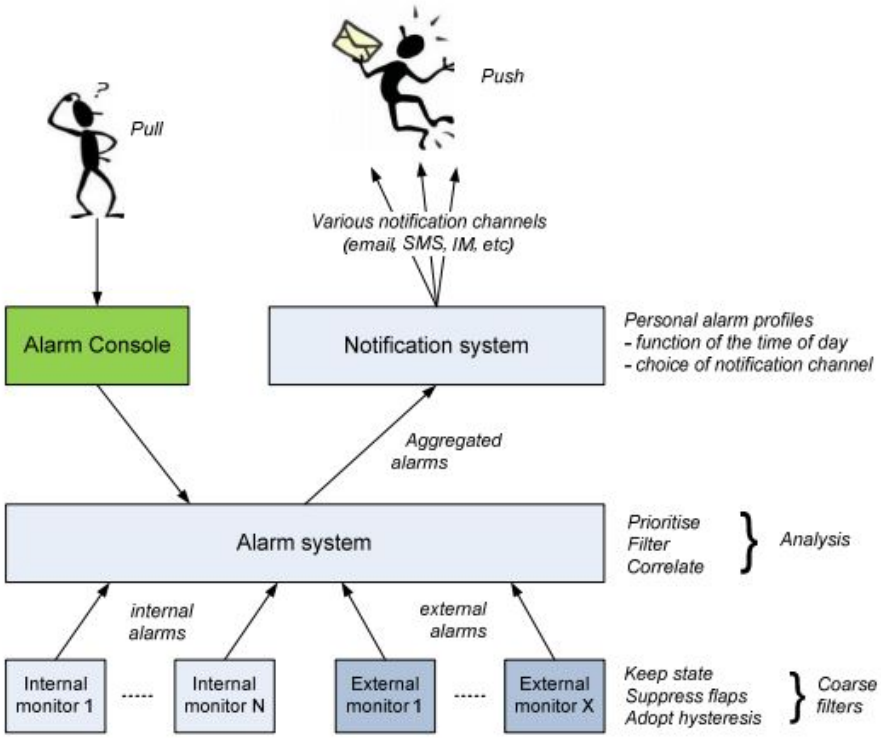


Figura 2.7: Ilustração de um sistema de alarme e notificações [3].

A descoberta de dispositivos automática é uma funcionalidade bastante útil, um sistema que automaticamente se adapta à rede real faz com que não seja necessário mudar o sistema manualmente sempre que existe uma mudança física na rede.

A facilidade de utilização é um fator que afeta a produtividade dos administradores. A interface deve ser bem pensada e intuitiva, é preciso levar em conta o tempo de treino necessário até um administrador dominar o sistema. Muitas soluções existentes têm demasiadas funcionalidades que os administradores não precisam, ver Figura 2.8, fazendo com que o preço das soluções seja demasiado elevado pois existe desperdício de funcionalidades. Uma solução é criar o sistema por vários módulos e com diferentes versões dependendo das funcionalidades que cada administrador irá precisar.

Integrar o sistema com uma base de dados possibilita analisar de várias maneiras as informações obtidas enquanto o sistema monitoriza a rede, reduz o tempo que se gasta a gerir a informação, podendo ser útil para verificar certos padrões de comportamento na rede que tenham acontecido e ajuda a prevenir outros problemas futuros que possam ocorrer utilizando mineração de dados. As bases de dados de grafos são tipos de bases de dados ideais para guardar dados relacionados com redes, pois as redes são grafos. O desempenho da base de dados é constante à medida que os dados aumentam ao contrário

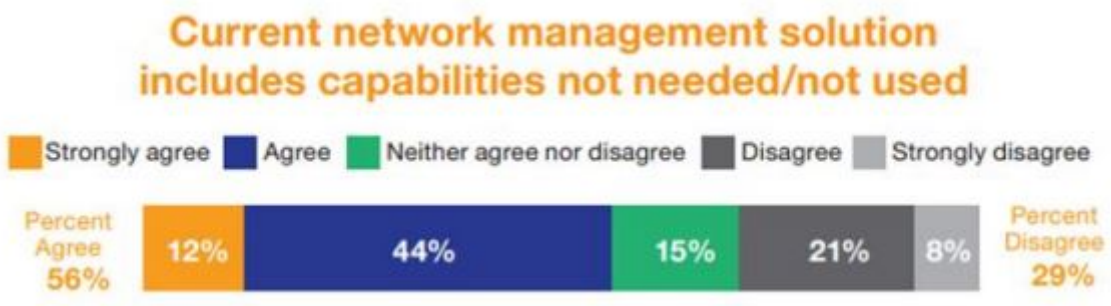


Figura 2.8: Inquérito realizado sobre funcionalidades dos sistemas que não são utilizados [4].

das bases de dados tradicionais e são mais flexíveis, permitindo facilmente adicionar à estrutura já existente, sem precisar de modelar previamente.

Um exemplo de plataforma de bases de dados de grafos é o *Neo4j* [28], lançado inicialmente em 2007 e escrito em Java. Permite escalabilidade alta de escrita e leitura, a adjacência sem índices diminui o tempo de leitura e fica ainda melhor à medida que a complexidade dos dados aumenta. Usa a linguagem declarativa de consultas de grafos, *Cypher* [29].

A escalabilidade do sistema também é um fator importante para lidar com o crescimento de nós no sistema, é recomendado criar um sistema que tenha em conta o futuro e a possível expansão da rede e que potenciais medidas tomar para permitir escalabilidade.

2.3 REDES *overlay* P2P

Nesta secção será feita uma descrição da aplicabilidade de um sistema P2P para monitorização de redes e suas vantagens, posteriormente será feita uma comparação de redes centralizadas, descentralizadas, estruturadas e não estruturadas.

2.3.1 *Aplicabilidade*

Uma rede *overlay* é uma rede virtual construída em cima de uma rede física. Muitas redes *overlay* modernas são construídas por cima da Internet como a rede subjacente. A maior parte destas redes existem para fornecer diretamente funcionalidade a nível de aplicação que está fora do alcance da rede subjacente como por exemplo *Content Delivery Networks (CDN)*s e *Resilient Overlay Network (RON)*. As redes *overlay* são utilizadas em

várias aplicações relacionadas com jogos, monitorização e medição de redes e distribuição de conteúdo.

Um sistema de monitorização de rede pode beneficiar utilizando redes P2P. Com redes *overlay* os programadores podem desenvolver e implementar novos algoritmos de *routing* e gestão de pacotes rapidamente e facilmente por cima da Internet. Não é necessário modificar protocolos da Internet e ter que passar por problemas técnicos e políticos, os ISPs podem criar estes novos serviços e funcionalidades sem precisarem de qualquer mudança na rede subjacente, nem de uma adaptação universal dos ISPs.

As redes P2P são escaláveis se cada utilizador estiver a partilhar a sua parte de computação, mais utilizadores significa mais capacidade. Adicionar novos *peers* é fácil e se um dos *peers* falha a rede continua a funcionar corretamente. Como a computação pode ser distribuída por diferentes *peers* os custos também são reduzidos.

É apresentada uma taxonomia de arquiteturas P2P baseadas em sistemas existentes que foram desenvolvidos na Figura 2.9.

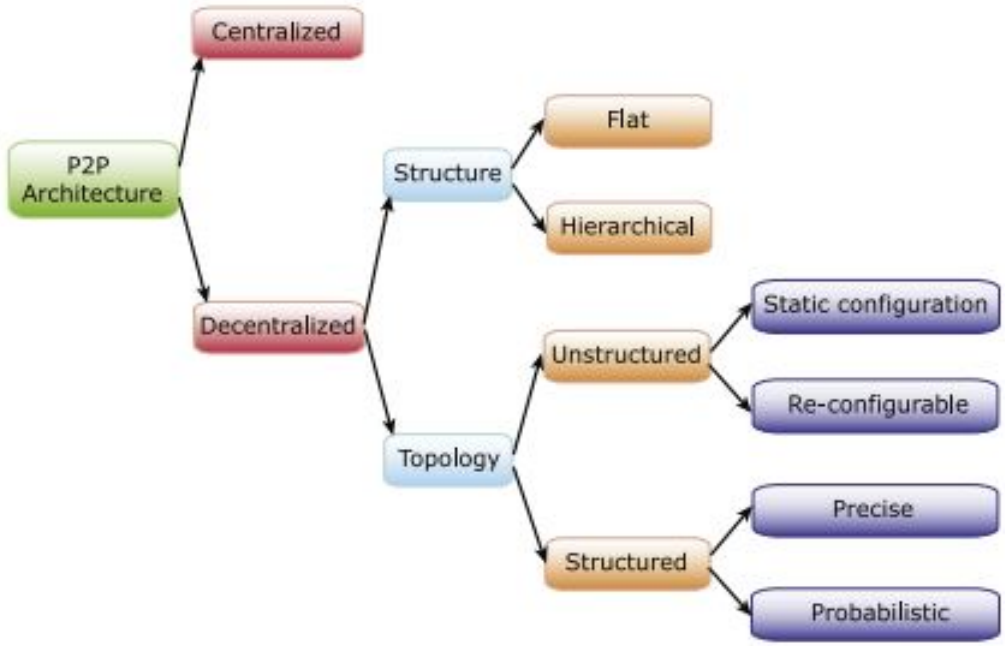


Figura 2.9: Sistemas P2P [5].

Num extremo alguns sistemas P2P são suportados por servidores centralizados, noutro, sistemas P2P puros são completamente descentralizados. No meio destes dois extremos existem os sistemas híbridos onde os nós são organizados em duas camadas, os servidores de camada superior e os nós comuns de camada inferior. Alguns desses sistemas serão descritos nas subsecções seguintes.

2.3.2 *Arquitetura Centralizada*

Os sistemas centralizados P2P misturam as caraterísticas de arquiteturas centralizadas (ex cliente-servidor) e descentralizadas [6]. Num sistema cliente-servidor há um ou mais servidores centrais, o que ajuda os *peers* a localizar os seus recursos ou funcionar como um gestor de tarefas para coordenar todos os *peers*. Para localizar recursos, um *peer* pode enviar mensagens para o servidor central para determinar o endereço de *peers* que contenham determinados recursos. Mas, como um sistema descentralizado, quando um *peer* tem a informação que necessita, pode comunicar diretamente com outros *peers* sem passar pelo servidor. Como em todos os sistemas centralizados, esta categoria de sistemas P2P é susceptível a ataques maliciosos e pontos únicos de falha. Um servidor centralizado pode ser um *bottleneck* para um número grande de *peers*, que potencialmente degrada o desempenho drasticamente, sendo necessários mais servidores. Este tipo de sistema carece de escalabilidade e robustez sendo necessário de implementar maneiras de amenizar estas fraquezas, a Figura 2.10 ilustra a arquitetura centralizada. Como exemplo desta arquitetura existe o Napster e BOINC [30][31].

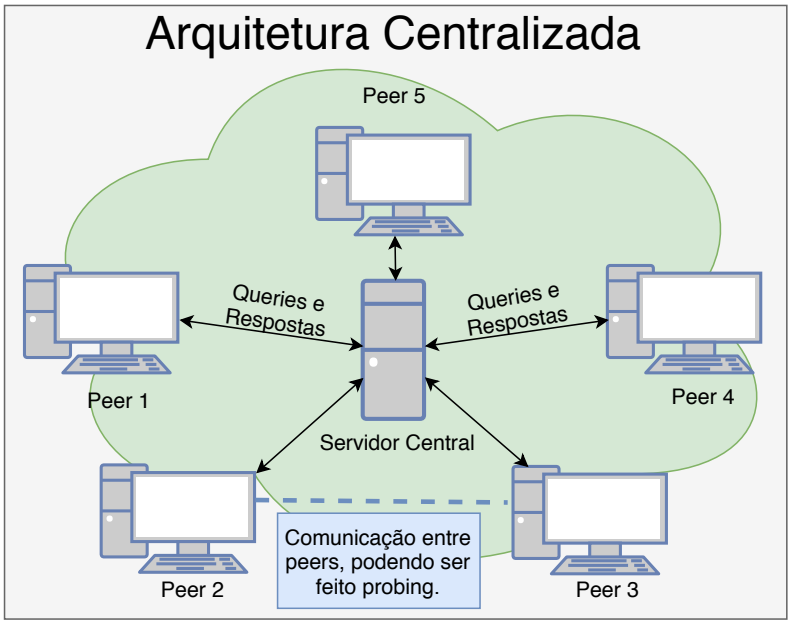


Figura 2.10: Ilustração de uma arquitetura P2P centralizada.

2.3.3 *Arquitetura Descentralizada*

Num sistema descentralizado P2P, os *peers* têm responsabilidades e direitos iguais. Cada *peer* tem uma vista parcial da rede P2P e oferece serviços e dados que podem ser relevantes para só algumas *queries* ou *peers*. É então critico e desafiante localizar rapidamente estes *peers* que oferecem serviços e dados. Estes sistemas têm bastantes vantagens, são imunes a pontos de falha singulares e potencialmente têm alta performance, escalabilidade e robustez. Como ilustrado na Figura 2.9, existem duas dimensões no design de um sistema P2P descentralizado. Uma estrutura de rede pode ser plana ou hierárquica. Numa estrutura plana, a funcionalidade e o carregamento são distribuídos de maneira uniforme sobre os nós participantes. A maior parte dos sistemas descentralizados existentes não são hierárquicos. Por outro lado, um design hierárquico naturalmente oferece certas vantagens incluindo isolação de faltas e segurança, *caching* e utilização de banda larga eficaz, armazenamento hierárquico, etc. Numa estrutura hierárquica existem várias camadas de estruturas de *routing*. Por exemplo, a nível nacional, pode existir uma estrutura para conectar distritos, dentro de cada distrito pode existir outra estrutura de *routing* para universidades dentro do distrito, e dentro de cada universidade pode existir ainda outro nível que liga departamentos e por aí em diante.

Na dimensão que diz respeito à rede *overlay*, pode ser estruturada ou não estruturada. A diferença entre estes tem a ver sobre como cada *query* é enviada a outros nós. Num sistema P2P cada *peer* é responsável pela sua própria informação e mantém informação sobre os vizinhos que podem enviar *queries*. Não há mapeamento estrito entre os identificadores de objetos desses *peers*, o que significa que localizar informação neste tipo de sistemas pode ser desafiante pois é difícil prever precisamente que *peers* mantêm os dados em que foram feitas *querys*, também não há garantia de que as respostas estejam completas, a não ser que a rede inteira seja procurada.

As plataformas conhecidas que implementaram sistemas P2P não estruturados são *FreeNet* e o *Gnutella* original.

O *FreeNet* aplica mecanismos *unicast* de procura para localizar recursos, que é ineficiente em termos de tempo de resposta, mas eficiente em respeito à utilização da largura de banda e o número de mensagens utilizadas [32].

O *Gnutella* utiliza estratégias de *routing* e de *flooding*, o que é eficiente em termos de tempo de resposta mas ineficiente em termos de utilização de largura de banda e número de mensagens utilizadas (pois a rede é inundada com um número exponencial de mensagens) [33]. Um problema das redes não estruturadas P2P é a obtenção dos vizinhos. Estes vizinhos podem ser pré-determinados estaticamente e fixados.

Num sistema P2P estruturado, a colocação dos dados está sob controlo de estratégias pré-definidas, normalmente uma *Distributed Hash Table (DHT)*, existindo mapeamento entre

dados e os peers. O objetivo destes sistemas é retirar os benefícios de pesquisa numa topologia estruturada enquanto se mantém um bom grau de autonomia, mantendo o custo baixo como numa topologia não estruturada. O *Chord* é um protocolo e algoritmo desenvolvido, baseado em DHTs [34].

Por fim, existem os sistemas híbridos P2P que se aproveitam das vantagens de ambas arquiteturas centralizadas e descentralizadas. A vantagem principal de um sistema centralizado P2P é que consegue fornecer um sistema que localiza rapidamente e solidamente os recursos a serem pesquisados, tendo por limitação a escalabilidade que é afetada pelo uso de servidores, sistemas P2P descentralizados são melhores neste aspeto mas perdem na parte de localizar os recursos. Os sistemas híbridos P2P para manterem a escalabilidade semelhante a um sistema descentralizado, não têm servidores. Os *peers* que têm mais capacidade que outros podem ser selecionados como servidores ou *super peers*. Desta maneira a localização dos recursos pode ser feita por técnicas descentralizadas ou centralizadas.

Uma tabela a comparar estes sistemas P2P que foram descritos pode ser vista na Figura 2.11.

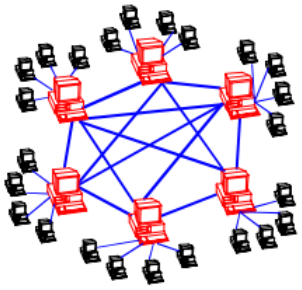
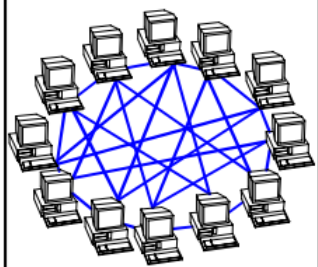
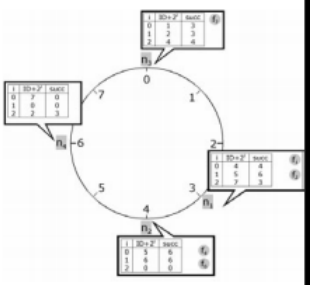
Unstructured P2P		***Structured P2P***
Hybrid P2P	*Pure P2P*	*DHT-Based*
1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → dynamic central entities Example: Gnutella 0.6, JXTA	1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities Examples: Gnutella 0.4, Freenet	1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities 4. Connections in the overlay are "fixed" Examples: Chord, CAN
		

Figura 2.11: Comparação dos vários tipos de sistemas P2P (adaptação de [6]).

2.4 PROJETOS RELACIONADOS

Nesta secção são apresentados quatro projetos relacionados com o tema e objetivos deste projeto, o *RIPE Atlas*, *perfSONAR*, *SamKnows* e *NLNOG Ring*, no fim desta secção será feita uma comparação entre eles.

O RIPE Atlas é uma rede global de *probes* que medem a conectividade da Internet e a sua acessibilidade, fornecendo informação sobre o estado da Internet em tempo real [35]. Este projeto fornece monitorização contínua de vários pontos do globo. Permite investigar e resolver problemas com a rede com verificações de conectividade rápidas e flexíveis. Também tem a funcionalidade de criar alarmes. Os *probes* do *RIPE Atlas* são pequenos dispositivos que se podem ligar a uma porta Ethernet num *router* através de um cabo de rede *Unshielded twisted pair (UTP)*, fazendo vários tipos de medições e enviando esta informação a uma entidade central, o *RIPE NCC*, onde os dados são agregados com o resto dos dados da rede. Estes *probes* usam pouca largura de banda e não conseguem determinar informação do conteúdo que passa nos computadores dos *hosts*. Estes *probes* efetuam estes tipos de medições: *ping*, *traceroute*, *Secure Sockets Layer (SSL)*/*Transport Layer Security (TLS)*, *Domain Name System (DNS)*, *Network Time Protocol (NTP)* e HTTP a alvos selecionados. O RIPE Atlas também é constituído por *anchors* que são *probes* do RIPE Atlas com capacidade de medição maior e contêm informação valiosa sobre a conectividade local e regional e a acessibilidade da Internet.

O projeto *perfSONAR* é um kit de ferramentas para correr testes na rede em vários domínios, sendo utilizado em investigação de rede e educação [7]. Existe uma infraestrutura com várias instâncias pelo mundo, em que muitas delas estão disponíveis para testes do desempenho da rede. Esta infraestrutura ajuda a identificar e isolar problemas, criando o papel de suportar utilizadores da rede mais facilmente para as equipas de engenheiros e aumentar a produtividade quando se utilizam recursos da rede. Este projeto fornece uma interface uniforme que permite agendar medições, guardar dados em formatos uniformes, e métodos escaláveis para adquirir dados e gerar visualizações. Também permite modificar o sistema para suportar novas métricas.

A *SamKnows* foi fundada em 2008 com o objetivo de criar uma metodologia standard para medir o desempenho da Internet e da rede [36]. A empresa opera internacionalmente, tem o objetivo de monitorizar e fornecer informações sobre as conexões dos utilizadores. Atualmente a *SamKnows* tem à volta do mundo mais de 40000 *probes* de medição. Fornece os dados colecionados do desempenho da rede a dois grupos de consumidores, aos utilizadores que tenham os *probes*, chamados *whiteboxes*, e a outro grupo, os ISPs, governo e reguladores. As *whiteboxes* são pequenas caixas que quando são conectadas a um *router* ou

modem, medem o desempenho da Internet, medindo vários aspetos como latência, perda de pacotes, *download*, *upload*, etc.

O *NLOG RING* é uma colaboração de um número em crescimento de organizações com o objetivo de depurar e solucionar problemas de redes de uma maneira eficiente e flexível [37]. O *RING* fornece operadores com uma vista de fora e várias ferramentas. O *RING* é construido inteiramente por voluntários da comunidade e por organizações que se podem juntar sem cobrança adicional, sendo o *RING* bastante único.

Na Figura 2.12 encontra-se uma comparação dos vários projetos referidos nesta secção.

	perfSONAR	RIPE Atlas	SamKnows	NLNOG RING
Description	Network measurement toolkit designed to provide federated coverage of network paths. It provides an interface that allows for the scheduling of measurements, storage of data and generate visualizations	Active measurement network from the RIPE NCC. It consists of measurement probes that run measurements in the RIPE Atlas system and report results to the central data collection components	Probes for the measuring end-user broadband performance (fixed-line and mobile). They execute a series of software tests over their broadband connection they are connected to. The results of these tests are reported securely up to hosted backend infrastructure.	The infrastructure of (virtual) machines available to its participants. It offers ssh access to all servers which are part of the project to run custom scripts executing commands on all or a subset of the servers. These scripts run from own machine or from one of other nodes.
Type of measurements	Throughput (TCP and UDP), RTT, One-way delay, One-way packet loss, Traceroute	RTT to the first and second hops, Ping to predetermined destinations, Traceroute to predetermined destinations, DNS queries to root DNS servers, SSL queries to predetermined destinations	Multi-threaded HTTP download speed, Multi-threaded HTTP based upload speed, Availability of the connection, Jitter, Latency (ICMP and UDP), Packet loss (ICMP and UDP), DNS query resolution time, DNS query failure rate, Web page loading time, Web page loading failure rate, Video streaming performance	RTT, Traceroute, ssh and system tools from predetermined destinations to any other host
User-defined scheduled measurements	yes	yes (limited by # of so called credits)	no	no
On-demand measurements	yes	yes (limited by # of credits)	no	yes
Incoming measurements control	yes	no	no	no
Measurements target control	between any pefSONAR host	to predefined RIPE hosts, to other probe	between predefined hosts	between any RING host
Type of distribution	software	hardware	software, hardware	software
Measurement data storage distribution	local or centralised	centralised	centralised	N/A
Measurement data storage architecture	data stored in user infrastructure	data stored in service provider infrastructure	data stored in service provider infrastructure	N/A
Access to archive measurements	local or central interface	central web interface service	central web interface service	N/A

Figura 2.12: Comparação dos vários projetos relacionados com o tema [7].

2.5 SUMÁRIO

Este capítulo foi crucial para desenvolver um sistema P2P para deteção de anomalias de redes, explorando as diversas tecnologias e programas existentes para monitorizar uma rede, sendo enumerados também vários requisitos que estes sistemas necessitam para se obter um sistema robusto e com várias funcionalidades para os administradores de redes. Por fim, foram estudadas várias formas de implementar um sistema P2P e sua aplicabilidade e foram apresentados quatro projetos relacionados com o tema proposto.

3

ARQUITETURA DO SISTEMA E MECANISMOS DESENVOLVIDOS

Neste capítulo será detalhada a arquitetura do sistema, especificando os vários componentes existentes nas diferentes entidades que serão apresentadas na secção 3.1, posteriormente a comunicação entre estas entidades será exposta na secção 3.2. Serão também descritos os passos para a criação de uma metalinguagem para ajudar um administrador a pré-programar a rede de monitorização na secção 3.3, como também todos os casos que ocorreram na otimização do sistema na secção 3.4. Por fim, será exposto como se tornou o sistema tolerante a falhas na secção 3.5.

3.1 ARQUITETURA

A arquitetura do sistema P2P de monitorização é composta por duas entidades principais, coordenador e peer. O coordenador, tem a função de coordenar todos os *peers* através de vários comandos, podendo iniciar *probings* com diversos parâmetros opcionais [38]. Esta entidade, também permite verificar os resultados dos *probings*, que podem ser posteriormente analisados numa interface gráfica, onde se poderá verificar se algum dos alarmes definidos pelo administrador disparou. A entidade *peer*, está encarregue de recolher informações sobre a rede e enviá-las ao coordenador. Num *probing*, existe um *peer* origem, que recebe os comandos do coordenador e envia os dados recolhidos da *path* entre esse *peer* origem e um *peer* destino, definidos ao iniciar um *probing* no coordenador. Na Figura 3.1, dois probings podem ser observados entre o *peer* A e B, e entre o *peer* A e C, onde os resultados estão a ser enviados a uma porta de transferência de dados do coordenador para posterior análise.

3.1.1 *Coordenador*

O coordenador da rede tem várias portas de comunicação que servem para controlar o sistema, estas, permitem enviar diversos comandos para os *peers* existentes na rede. O coordenador também é composto por várias portas de transferência de dados que permitem receber as várias medições que os *peers* estão a coletar ativamente, sendo estas medições

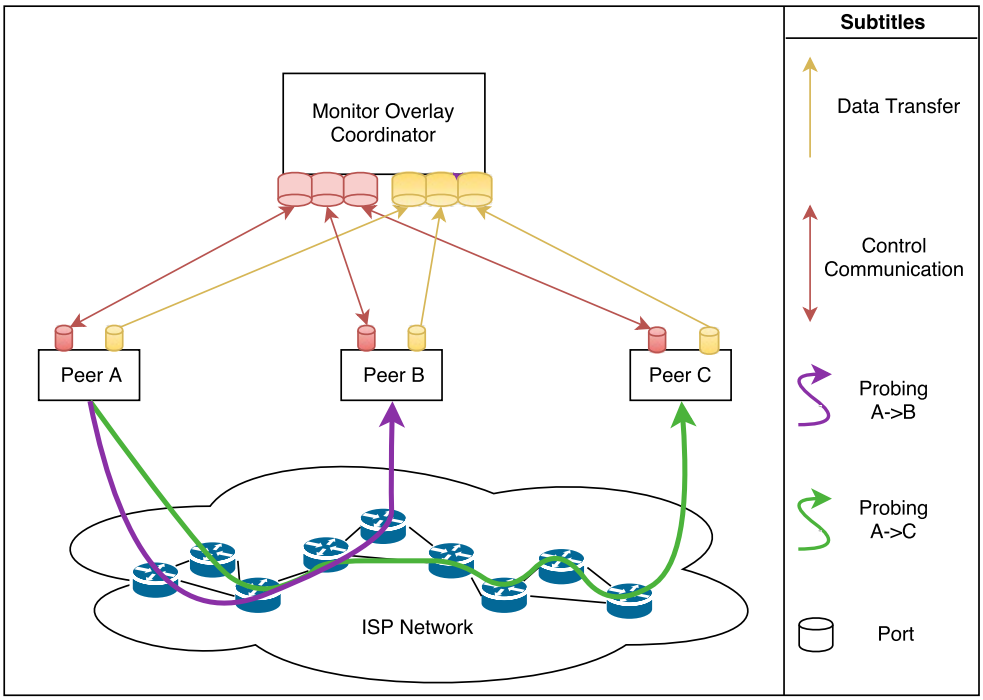


Figura 3.1: Arquitetura geral da rede overlay P2P.

guardadas numa base de dados de grafos. Esta entidade tem uma interface para os administradores enviarem comandos, receber notificações de alarmes e verificar o estado da rede ISP, tal como ilustrado na Figura 3.2.

3.1.2 *Peer*

A entidade *peer*, tem dois canais distintos, um para controlo que é preciso para receber pacotes TCP com comandos que vão correr no *peer*, e enviar pacotes de resposta de confirmação de volta ao coordenador, onde será necessária uma *queue* de controlo que funcionará nas duas direções. O outro canal é usado para transferência de dados através da comunicação constante com pacotes TCP até ao coordenador, para monitorização contínua da rede, um *buffer* de dados também irá ser necessário, para temporariamente guardar os dados enquanto se transfere a informação do *peer* ao coordenador. Uma lista de *peers* é necessária para permitir *probing* entre vários *peers*. Os *peers* têm um controlador que será preciso para cada *peer* saber que medições são necessárias ser feitas (indicadas previamente pelo coordenador), como ilustrado na Figura 3.3.

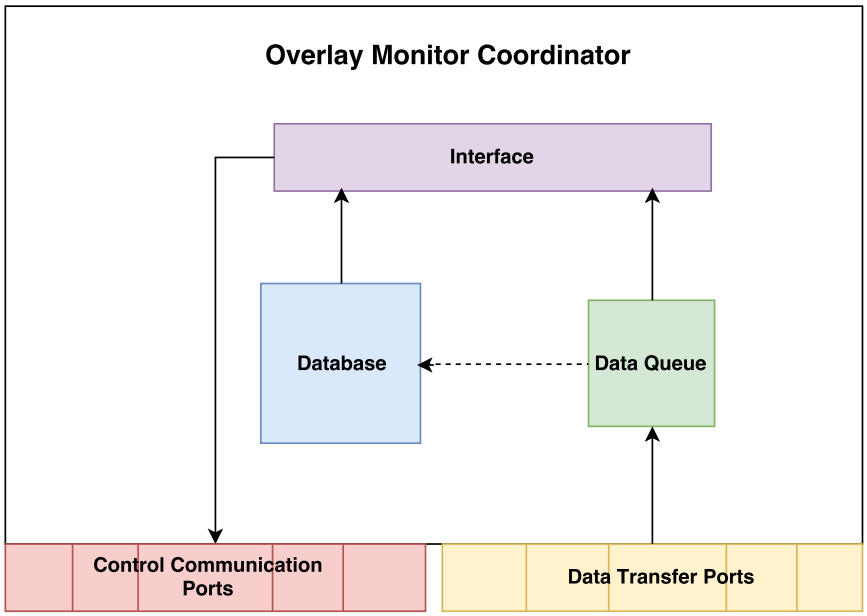


Figura 3.2: Coordenador da rede overlay P2P.

3.2 COMUNICAÇÃO ENTRE ENTIDADES

Nesta secção, será detalhada como é efetuada a comunicação entre as entidades, que protocolos de transporte são usados e explicada a razão destes serem escolhidos. Também serão apresentados os vários comandos que estão disponíveis para efetuar a comunicação do coordenador até *peer* e também como o *probing* P2P é realizado.

3.2.1 *Comunicação Coordenador e Peer*

Nesta secção, a comunicação entre as entidades coordenador e *peer* será detalhada. Existem dois tipos de comunicação, a de controlo e a de dados, na de controlo irão ser enviados comandos do coordenador aos *peers*, para os peers efetuarem as suas funções de monitorização. No tipo de comunicação de dados, serão transportados todos os dados recolhidos pelos *peers* até ao coordenador, informando o estado da rede.

O protocolo de transporte escolhido para a comunicação de controlo entre o coordenador e os *peers* foi o protocolo TCP. Este protocolo foi escolhido pois para correto funcionamento do sistema, é necessário entrega de pacotes confiável entre estas duas entidades, não sendo necessário preocupar com a ordenação dos pacotes de chegada ou erros. O número de portas usadas é o mesmo que o número de *peers* integrados na rede do ISP. Cada *peer* usa só uma porta para receber os comandos enviados pelo coordenador.

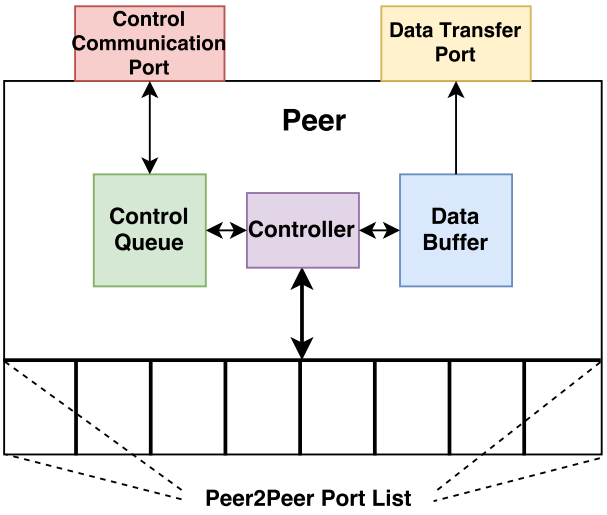


Figura 3.3: Peer da rede overlay P2P.

Para a comunicação dos dados também foi escolhido o protocolo TCP. Escolha efetuada pela robustez, versatilidade, garantia de entrega e facilidade de utilização que fornece. Tal como na comunicação de controlo, o número de portas usadas é o mesmo que o número de *peers* integrados na rede do ISP. Cada *peer* usa só uma porta para enviar os dados para o coordenador.

Como referido anteriormente irá ser possível enviar vários comandos do coordenador para os *peers*, estes comandos têm vários parâmetros opcionais para facilitar o administrador a escolher a melhor maneira de monitorizar a rede e definir os alarmes de diversas formas. Apresentam-se de seguida alguns dos comandos definidos:

- **PacketLoss** *Number of Packets, Size, Timeout, Loop, Feedback(Feedback Time) ou Statistic Options, Peer Y, Alarm(Alarm Condition):* Permite a medição da perda de pacotes entre o *route path* de dois *peers* diferentes (PeerX ->PeerY). Esta medição é calculada dividindo o número de pacotes perdidos pelo número total de pacotes enviados, sendo esta divisão posteriormente multiplicada por 100, sendo o resultado em percentagem;

- **Round Trip Time** *Number of Packets, Size, ICMP, Timeout, Loop, Feedback(Feedback Time) ou Statistic Options, Peer Y, Alarm(Alarm Condition):* Permite a medição do *round trip time* entre dois *peers* diferentes (PeerX ->PeerY). A ICMP *flag* está ativa por defeito (1), mas o administrador pode desativá-la mudando o seu valor para 0. Quando ativa, esta medição vai usar um *echo-request* para medir o *round trip time*. Se a ICMP *flag* está inativa (0), o *peer* vai usar uma implementação de *round trip time* puramente criada em *Java* usando UDP;

- **Jitter** *Number of Packets, Size, Loop, Timeout, ICMP, Feedback(Feedback Time) ou Statistic Options, Peer Y, Alarm(Alarm Condition):* Este comando mede a variação da latência entre pacotes de dados sucessivos, entre um caminho desde o *peer* de origem até ao *peer* de destino. Se a ICMP *flag* é ativada (1), que toma o valor de 1 por defeito, vai ser utilizado um *echo-request* para obter uma média do quão longe cada *ping* RTT está da média de RTT. Se a ICMP *flag* é desativada (0) o *one-way jitter* vai ser medido com uma implementação UDP no *peer* de destino, calculando a variação de atraso entre pacotes sucessivos;

- **RoutePath** *Timeout, Loop, Feedback(Feedback Time) ou Statistic Options, PeerY:* Este comando permite ao coordenador obter a *route path* do *PeerX* ao *PeerY*. Uma chamada ao sistema é usada para medir este processo. Isto faz com que seja possível verificar quando ocorrem mudanças em especificas *route paths* do ISP, e quanto tempo essas mudanças persistem, o que pode indicar uma possível anomalia na rede.

Além dos comandos anteriores de medição também existem comandos especiais, usados para outras utilidades:

- **Activate/Deactivate** *PeerX*: O comando *Activate* ativa um *peer*, fazendo com que comece à escuta de vários comandos ou retorne ao estado prévio de medições, se tiver tido algum antes. Por outro lado, o comando *Deactivate* para todas as comunicações e funcionalidades de *peers* que ocorrem no nó coordenador;

- **Cancel** *Probe ID, Timeout, PeerY:* Permite ao administrador cancelar indefinidamente um processo de medição que está a decorrer num *peer*. Depois de receber este comando e quando a informação está pronta a enviar, o *peer* vai enviá-la ao coordenador.

Para cada comando, além dos parâmetros necessários estão disponíveis alguns parâmetros opcionais com valores por defeito que podem ser usados, como ilustrado e explicado na Tabela 3.1.

3.2.2 *Comunicações P2P*

Os processos de comunicação entre os *peers* ativos na rede *overlay*, são consequência dos comandos recebidos pelo coordenador, sendo necessário realizar medições específicas na rede ISP subjacente. Dependendo dos comandos recebidos, são criados pacotes UDP distintos, com diferentes formatos e conteúdos que são gerados pelos *peers* que os enviam, desencadeando respostas de pacotes UDP, dos *peers* que os recebem. Baseado nos pacotes que foram trocados, e nos seus correspondentes conteúdos, vários dados de medição

Parâmetro	Descrição do Parâmetro
Number of Packets	Um valor numérico que indica o número de pacotes de dados que o administrador deseja enviar. O seu valor por defeito é 5.
Size	O tamanho do pacote que deve ser usado (apenas é útil para alguns comandos).
Loop	Um valor booleano que indica se a medição vai decorrer durante um tempo indefinido ou instantâneo. Por defeito este valor está a tempo instantâneo (0). Se a medição está definida para decorrer durante um tempo indefinido, o administrador pode escolher entre: *Time Feedback* (1), onde o administrador define o tempo entre as respostas dos *peers* ao coordenador; e *Statistical Feedback* (2), que permite ao administrador escolher um conjunto de opções de estatísticas para serem aplicadas quando o administrador cancela um comando de medição a ser feito *probing* pelos *peers*.
Timeout	Um valor numérico que indica quando um comando deve ser descartado, na presença de problemas. O seu valor por defeito é de 5 segundos.
Probe ID	O coordenador identifica o comando que quer cancelar. Apenas possível no comando *Cancel*.
Statistic Options	Com este parâmetro o administrador define os tipos de estatísticas que quer analisar: *Average*, *Mean*, *Maximum Value*, *Minimum Value*, *Mode*, *Median*. Este comando apenas é efetivo quando é utilizado o comando *Cancel* num comando com o *Loop flag* ativo (2), de outra forma o resultado será instantaneamente enviado ao administrador.
ICMP	Este valor de *flag* por defeito é ativo (1). Com esta *flag* ativa-se o protocolo ICMP para usar num comando específico. Se a *flag* for desativada pelo administrador, é usada uma implementação para executar o comando sem ICMP, puramente feita em Java.
Feedback	Se o administrador quiser ver os resultados de *probing* que são enviados pelos *peers*, pode definir o intervalo de tempo em que os *probings* são feitos e enviados para ele em milissegundos no campo do tempo de *Feedback*. Por defeito esta opção está ativa. Isto é possível apenas com a *flag* de *Time Feedback Loop* ativa, de outra forma o resultado será instantaneamente enviado ao Coordenador se o *Loop* for instantâneo. O resultado só será enviado se a sondagem for cancelada e a *Loop flag* estiver definida como *Statistical Feedback*.
Alarm Condition	Se a *Alarm flag* estiver ativa (1), o administrador precisa de escolher uma condição lógica (maior, menor, igual a valor) para disparar o alarme.

Tabela 3.1: Parâmetros dos comandos enviados pelo coordenador

são armazenados nos *peers* para transmissão subsequente para o coordenador da *overlay*, para serem armazenados na base de dados e serem apresentados os resultados dos *probings*. Quando um resultado de um *peer* é recebido, é verificado se este contém uma condição de alarme para verificação, se sim, é verificado o valor recebido com a condição definida pelo administrador. No caso das métricas *jitter*, média do RTT e *packet loss* podem ser aplicadas as condições $<$, $>$, $<=$ e $>=$, caso seja atingida, o alarme é disparado e o administrador é notificado na janela dos alarmes. Sempre que o coordenador recebe novos valores de *probings*, é também verificado se houve uma mudança de *path* e, se houver condição de alarme de mudança de *path*, é acionado um alarme caso a *path* nova for diferente da *path* anterior.

3.2.3 *Estrutura dos Pacotes*

Esta subsecção apresenta dois tipos de pacotes trocados entre o coordenador e os *peers:* 1) Os pacotes de controlo enviados pelo coordenador da *overlay* para os *peers*, contendo os comandos de *probing* que o coordenador pretende desencadear nos *peers* e 2) os pacotes de dados enviados pelos *peers* para o coordenador, contendo informação relacionada com a medição que vai ser guardada, analisada e apresentada ao administrador da rede ISP, usando as interfaces apropriadas. O formato do pacote do controlo de mensagens enviado pelo coordenador da *overlay* é apresentado na Figura 3.4. O pacote contém alguns campos que permitem a identificação do pacote e o processo de medida (campos de pacotes *pid* e *pckid*), seguido por uma identificação dos comandos descritos na secção anterior (campo de pacote *type* com valores "A"*Activate*, "D"*Deactivate*, "P"*Packet Loss*, "R"*Round Trip Time*, "r"*Route Path*, "C"*Cancel*, "J"*Jitter*, "I"*Packet Injection*, etc.). Após a identificação dos *peers* que participam no processo de *probing* (*peer* de origem e destino), os seguintes campos incluídos no formato do pacote da Figura 3.4 estão relacionados com os parâmetros de funcionamento descritos anteriormente, que variam dependendo do comando emitido pelo coordenador da *overlay*.

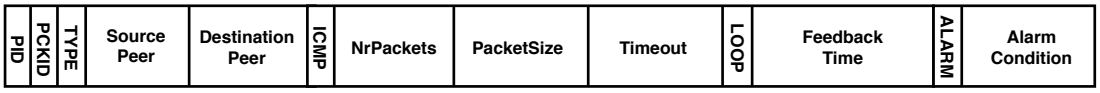


Figura 3.4: *Protocol Data Unit (PDU)* do link de controlo entre coordenador e peers.

De forma semelhante ao formato do pacote de controlo de comunicação, a Figura 3.5 ilustra o formato do pacote associado com os processos de transferência de dados entre os *peers* da *overlay* e o nó coordenador. Os resultados de monitorização (ou os dados estatísticos) são transmitidos no campo dos resultados/dados estatísticos, também se pode verificar na imagem que existe um campo para a mensagem de alarme, o que foi retirado na nova versão do sistema, pois o processamento dos alarmes passou a ser feito no coorde-

nador, permitindo a reutilização dos valores dos *probings* e redução do *payload* dos pacotes quando existe um alarme. Como explicado anteriormente, parte da informação transmitida nestes pacotes vai ser guardada numa base de dados interna, no nó coordenador.

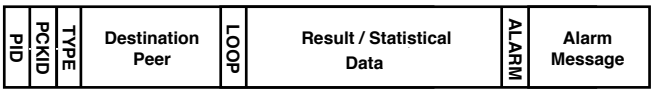


Figura 3.5: PDU do link de dados entre coordenador e peers.

3.3 METALINGUAGEM DE PROGRAMAÇÃO DO SISTEMA

Foi criada uma metalinguagem com o objetivo de aumentar a versatilidade e rapidez com que os administradores interagem com o sistema. Esta nova forma de comunicar com o sistema oferece a possibilidade de pré-programar a rede de monitorização através de instruções que podem ser construídas de várias formas, possibilitando a adaptação a diferentes cenários e tornar determinadas ações autónomas do sistema. Permite ações como iniciar ou cancelar probings numa data especifica, ou quando acontece algo na rede, tornando o sistema reativo.

3.3.1 *Definição da metalinguagem*

Esta metalinguagem tem o papel de fornecer novas funcionalidades ao sistema:

- Pré-programar a rede com instruções enviadas aos *peers* num tempo definido previamente, permitindo iniciar ou cancelar *probings* numa data especifica ou entre datas;
- Condicionalmente iniciar *probings* através de quantas vezes a condição de alarme de outros *probings* já existentes na rede foi disparada, permitindo a adaptação a diferentes cenários;
- Inserir uma lista de instruções através de um ficheiro ou numa área de texto que o sistema pode interpretar e corre-las uma a uma, rapidamente configurando o sistema;

Existem três tipos de instruções na metalinguagem definida, início de *probing*, cancelar *probing* e condicional, onde em cada tipo será possível fazer várias combinações e diferentes funcionalidades opcionais:

- No tipo de instrução início de *probing*, tem que ser indicado o IP de origem e o IP de destino, a métrica que irá ser indicada na condição de alarme, se vai utilizar ICMP e o tipo de *loop*. Existem também campos opcionais, o número de pacotes a serem

enviados, o tamanho desses pacotes, data de início e de fim, ou só data de início continuando o *probing* a correr.

- Na instrução de cancelar um *probing* é necessário indicar o id do pacote, existindo a opção de cancelar numa data especifica.
- Também existe a instrução condicional que, dando um id de um *probing* já a decorrer, caso esse *probing* dispare o alarme X vezes, é iniciado uma ou várias instruções novas.

Para ajudar a definir a gramática da metalinguagem foi utilizada a técnica de notação *Backus-Naur form (BNF)*, normalmente utilizada para descrever a sintaxe de linguagens usadas em computação [39], ilustrada na Figura 3.6.

Forma Backus–Naur

Instruction	::=	**ProbingFields** | **ProbingFields Instruction**
		| **CancelFields** | **CancelFields Instruction** | **Conditional**
ProbingFields	::=	Addresses Metric **AlarmFields** Number Number Icmp **LoopFields** **OpsFields**
CancelFields	::=	PacketId | PacketId Date
Conditional	::=	PacketId Condition Value **Instruction** | PacketId Condition Value Date Instruction
LoopFields	::=	LoopType FeedbackTime | LoopType
OpsFields	::=	"" | Date Date | Date | **Conditional** | Date Date **Conditional** | Date **Conditional**
AlarmFields	::=	Condition Value | Condition (caso routepath)
Addresses	::=	IPV4Address IPV4Address
IPV4Address	::=	Java Regex ^([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\. ([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\.([01]?\\d\\d?|2[0-4]\\d|25[0-5])
Metric	::=	"jitter" | "j" | "RTT" | "rtt" | "routepath" | "rp" | "packetloss" | "pl"
Number	::=	Java Integer | "" (valor por defeito)
Icmp	::=	"yes" | "no" | "y" | "n"
PacketId	::=	Java Integer
Date	::=	Java Date
LoopType	::=	"statistical" | "s" | "atomic" | "a" | "loop" | "l"
FeedbackTime	::=	Java Integer
Condition	::=	">" | "<" | ">=" | "<=" | "!=" (routepath)
Value	::=	Java Integer

Figura 3.6: *Backus-Naur form* da metalinguagem criada para o sistema *overlay*

3.3.2 *Exemplos de utilização*

De seguida serão apresentados e descritos alguns exemplos da utilização da linguagem:

- *Probing* normal, *10.0.2.1 10.0.5.2 j >= 50 56 20 yes loop 20*, neste exemplo, um *probing* de *jitter* com origem no IP *10.0.2.1* e destino no IP *10.0.5.2* é iniciado, onde o alarme será ativado quando o *jitter* for igual ou maior a 50, com o ICMP ativo, enviando 20 pacotes com o tamanho de 56 *bytes*, sendo os resultados enviados a cada 20 segundos em ciclo. Como descrito anteriormente, vários campos são opcionais;

- Cancelar *probing*, *1*, só é necessário o id do *probing* a cancelar;
- *Probing* com data de início e fim, *10.0.2.1 10.0.5.2 j >= 50 56 20 yes loop 20 2019-07-22-17-56-00 2019-07-22-17-57-00*, seguindo o exemplo de *probing* normal, para pré-programar um *probing* entre duas datas é necessário colocá-las depois da definição do *probing*, sendo a primeira a data do início do *probing* e a segunda data quando o *probing* deverá ser cancelado;
- Cancelar com data, *1 2019-07-22-17-57-00*, para cancelar um *probing* numa data especifica é necessário colocar a data à frente do id do *probing* a cancelar;
- Instrução condicional, *conditional 2 5 10.0.2.2 10.0.2.1 RTT > 100 yes loop 10*, neste exemplo, quando o alarme do probing com id 2 disparar pelo menos 5 vezes, será iniciado um novo *probing* de RTT. Este *probing*, terá origem no IP *10.0.2.2* e destino no IP *10.0.2.1*, onde o alarme será ativado quando o RTT for maior que 100, sendo os resultados enviados a cada 10 segundos em ciclo e com o ICMP ativo. Este tipo de instrução permite que a rede se torne reactiva, reagindo a possíveis acontecimentos futuros, podendo ser usada qualquer instrução em conjunção com a condicional.

3.4 OTIMIZAÇÃO DOS *probings*

Outro dos objetivos deste trabalho, é a redução do tráfego gerado pela rede de monitorização P2P através da composição dos resultados obtidos por diferentes *peers*.

Foi necessário a reestruturação de algumas partes do sistema, previamente, eram os *peers* que verificavam as condições de alarme de cada *probing* e enviavam se o alarme tinha sido disparado para o coordenador, agora, é o coordenador que faz a verificação, ao receber os resultados que os *peers* enviam.

Para otimizar o sistema de monitorização, foi necessário ter os valores do *jitter*, RTT e *packet loss* de cada *link* da *path* de cada *probing*, para os valores destas métricas poderem ser reaproveitados em novos *probings* que sejam criados pelo administrador. Por exemplo, se for iniciado um novo *probing* com links que já estão a ser monitorizados por outros *probings* existentes, o coordenador fica encarregue de reaproveitar os valores dos *probings* já existentes para o novo *probing*, sendo aplicadas as condições de alarme se tiverem sido definidas e sem ser necessário enviar comandos de início de *probings* a *peers*. A versão inicial do sistema utilizava a chamada de sistema *ping*, que não fornecia o valor das métricas referidas de cada *link* da *path* de um *probing*, sendo necessário, para obter todos esses valores, iniciar um *ping* por cada *link*. Para facilitar a obtenção desses valores, foi trocada a chamada de sistema *ping* pela ferramenta *My Traceroute (MTR)*, que fornece a funcionalidade das chamadas de sistema *ping* e *traceroute* [40]. Juntando a informação de ambas, a ferramenta MTR

retorna a rota inteira e respetivas métricas observadas em cada *link* que a compõe, entre dois determinados *peers* da rede de monitorização.

Sabendo os valores das métricas referidas de cada *link*, é então possível agregar os resultados de cada *probing*, não sendo necessário criar mais *probings* do que são precisos. Previamente era criado um novo *probing* sempre que era necessário saber qualquer uma das métricas, depois da otimização, como a ferramenta MTR retorna os valores de todas as métricas necessárias, é só preciso fazer um *probing* para retirar todos os valores precisos, quando anteriormente eram iniciados quatro *probings* diferentes.

Para otimizar os *probings* de RTT e de *packet loss*, foram estudados e implementados três casos diferentes onde são reaproveitados valores de *probings* já existentes na rede. Como ilustrado na Figura 3.7, um *probing* inicial representado a azul, está a ser efetuado entre o *peer* R1 até R5, de seguida é efetuado um outro *probing*, representado a vermelho, entre o *router* R2 e R4. Neste caso, não é necessário indicar a nenhum *peer* que é necessário iniciar outro *probing*, é o coordenador que irá iniciar verificações nos dados que o *probing* inicial lhe está a reportar, pois já tem informações de R2 a R4 e serão então aplicados os alarmes definidos pelo utilizador, otimizando a rede.

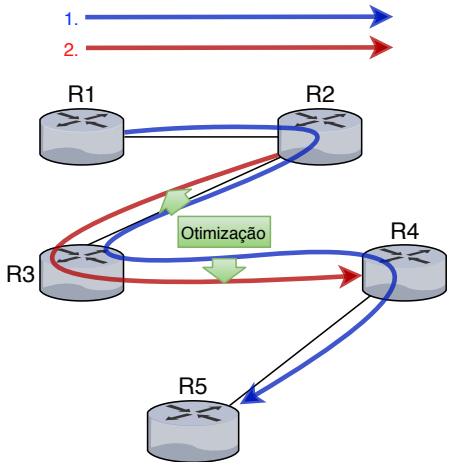


Figura 3.7: Caso 1 da otimização da rede, reaproveitar valores de um *probing* existente.

No segundo caso, serão também reaproveitados os valores de um *probing* existente mas será necessário enviar a um *peer* um novo início de *probing* pois o *probing* já existente não contém todos os dados necessários para efetuar o *probing* pedido pelo administrador do sistema. Como ilustrado na Figura 3.8, é efetuado um *probing* do *peer* R1 a R4, representado a azul e de seguida é efetuado um *probing* do *peer* R2 ao *peer* R5, representado a vermelho. Como o *probing* azul só contém informação do *peer* R1 a R4, o coordenador terá que iniciar um *probing* adicional para recolher a informação que falta do novo *probing* (R2 a R5), iniciando automaticamente um novo *probing* do *peer* R4 a R5. Para se obterem os valores finais das métricas RTT e *packet loss*, é necessário efetuar cálculos diferentes dependendo da

métrica em causa, estes cálculos serão especificados e explicados na secção 4.5 do capítulo seguinte. Caso o administrador tenha definido uma condição de alarme no *probing*, terá que ser feita a sua verificação no valor final resultante da otimização.

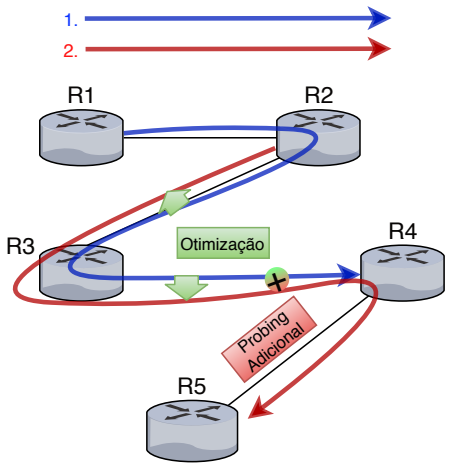


Figura 3.8: Caso 2 da otimização da rede, em que são reaproveitados valores de um *probing* existente e é necessário criar um *probing* adicional.

No último caso, ao contrário dos anteriores, em vez de serem reaproveitados os valores de *probings* já existentes, estes *probings* serão cancelados e é iniciado um com a *path* maior, que recolhe informações dos *probings* cancelados e outros caminhos. Como ilustrado na Figura 3.9, existe um primeiro *probing* representado a azul, que efetua medições do *peer* R1 ao *peer* R3, existe também outro *probing*, representado a verde, que efetua medições do *peer* R3 ao *peer* R5. O administrador posteriormente cria outro *probing*, representado a laranja, que tem início no *peer* R1 até ao *peer* R5. Como este *probing* tem uma rota com a *path* maior e inclui o primeiro e segundo *probing*, estes, já não serão necessários, sendo cancelados e passando a existir só o último *probing* efetuado. Desta forma, será necessário transportar as definições que o administrador tinha colocado previamente nos *probings* cancelados para os resultados do novo *probing*, para continuarem a funcionar como previamente e a reportar os devidos alarmes, otimizando a rede. Os três casos apresentados podem existir em conjunto, dependendo da combinação de *probings* que um administrador utilize para monitorizar uma rede.

3.5 Tolerância a Falhas do Sistema

Na versão inicial do sistema, a rede *overlay* não se ajustava caso houvesse, por exemplo, uma mudança de rota devido a alguma falha na rede. De igual forma, caso existissem falhas no coordenador, o sistema inteiro teria que ser restabelecido cancelando todos os *probings* a serem efetuados no momento da falha e refazendo-os posteriormente.

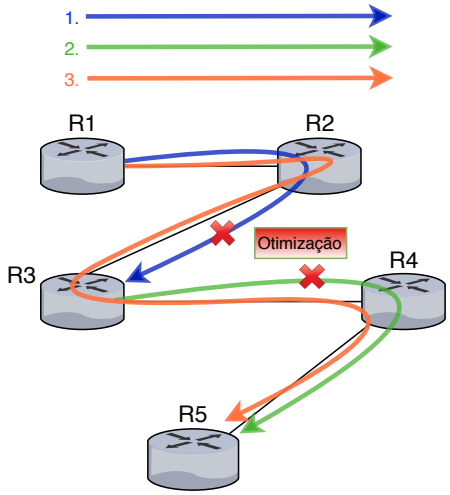


Figura 3.9: Caso 3 da otimização da rede, reaproveitar valores de um *probing* novo e cancelar *probings* existentes.

Para resolver estes dois casos anteriores foram feitas alterações ao sistema para o tornar tolerante a falhas que possam ocorrer na rede *overlay*.

Para permitir reajustes na interface para correta interpretação dos valores, o coordenador necessita de verificar constantemente se houve alguma mudança de rota nos *probings* que estão a ser efetuados na rede *overlay* e, se houver, efetuar as mudanças necessárias automaticamente, e informar o sistema da troca de rotas para posteriormente o administrador ter conhecimento.

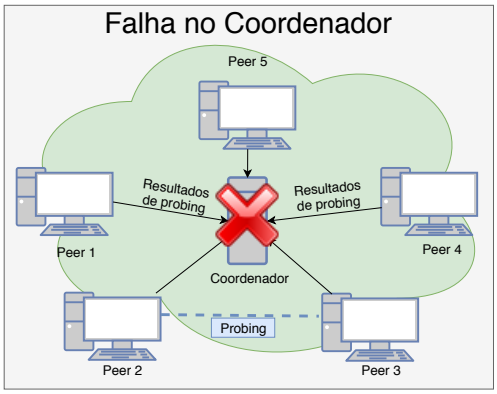


Figura 3.10: Falha do Coordenador no sistema

Se ocorrer uma falha no coordenador, é necessário haver um mecanismo que faça com que o estado anterior do coordenador volte a ser restabelecido, impedindo ter que refazer todos os *probings* de novo e evitar potenciais conflitos, com *probings* que estavam a decorrer

previamente, como ilustrado na Figura 3.10. Para isso, é necessário guardar os *probings* que são feitos, tal como todas as informações relacionadas com estes.

Se um ou vários *peers* tiverem falhas, como ilustrado na Figura 3.11 é necessário haver um mecanismo que faça com que o estado anterior de cada *peer* seja restabelecido, continuando a fazer os *probings* que estavam a fazer previamente, não necessitando de nenhum comando adicional do coordenador.

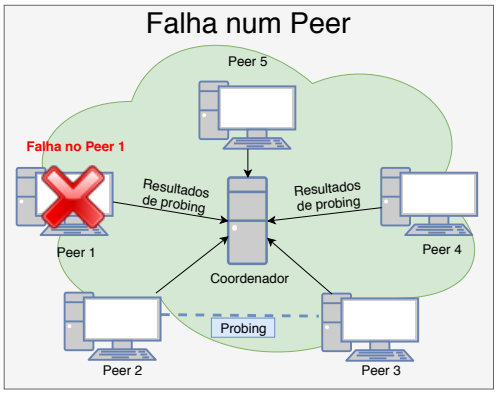


Figura 3.11: Falha de um *Peer* no sistema

3.6 SUMÁRIO

Neste capítulo, foram detalhadas as duas entidades principais do sistema, o coordenador e os *peers*, expondo o seu processo de comunicação e explicando comandos existentes ao administrador do sistema e a sua utilidade. Posteriormente foi especificado o papel da metalinguagem no sistema, permitindo pré-programar a rede com instruções enviadas aos *peers* através de uma lista de instruções dada pelo administrador. De seguida foi descrito o processo de otimização dos *probings* com o objetivo de reduzir o tráfego gerado pela rede P2P, através da composição dos resultados obtidos pelos diferentes *peers*. Por fim, foram explicados os benefícios de tornar o sistema tolerante a falhas bem como que processos têm que ser tomados em caso das falhas.

4

IMPLEMENTAÇÃO DO SISTEMA DESENVOLVIDO

Neste capítulo será exposto o processo de implementação das entidades da *overlay* e da sua comunicação na secção 4.1. É apresentado como foi feita a interface gráfica do sistema na secção 4.2 e como foi implementada a base de dados de grafos no sistema na secção 4.3. Também será descrito o processo de implementação da metalinguagem na secção 4.4, da otimização do sistema na secção 4.5 e da transformação do sistema para se tornar tolerante a falhas na secção 4.6, serão descritas que ferramentas foram utilizadas e a razão da sua escolha em cada uma das secções.

4.1 ENTIDADES DA *overlay* E PROCESSO DE COMUNICAÇÃO

Todas as entidades do sistema *overlay* foram implementadas em *Java*. Os processos de comunicação entre essas entidades foram implementados usando TCP/UDP *Java sockets*, sobre o qual todas as mensagens de controlo mencionadas, pacotes de transferência de dados e processos gerais relacionados ao *probing* ocorrem.

Para o sistema realizar todas as tarefas foi necessário criar threads nas duas entidades. No coordenador é necessário estar à escuta para receber todos os valores dos probings que os *peers* estão a efetuar. É precisa outra *thread* no coordenador para enviar os vários comandos aos *peers* quando necessário. O tratamento dos comandos recebidos é processado consoante os resultados dos PDUs recebidos, sendo feita a verificação dos alarmes sempre que for recebido um *probing* com uma condição de alarme definida.

Os *peers* também necessitam ter uma thread à escuta para receber os comandos do coordenador e outra para poderem ser efetuados *probings* sem ICMP ativo, utilizando pacotes UDP, criados com a classe de *Java DatagramPacket*. Na iniciação de *probings*, se a *flag* ICMP estiver ativa, são criados processos para fazer as chamadas de sistema com a ferramenta MTR.

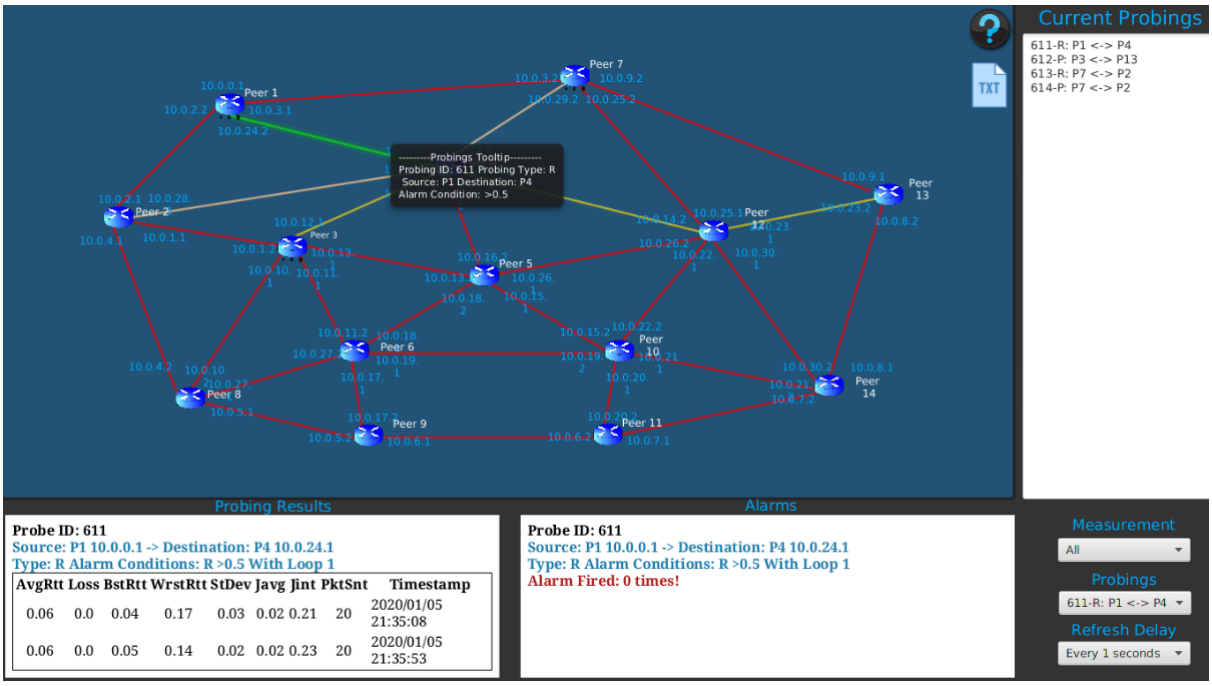


Figura 4.1: Interface desenvolvida para o sistema de monitorização *overlay* P2P

4.2 INTERFACE GRÁFICA DO SISTEMA

Foi desenvolvida uma interface gráfica para o administrador de ISP, usando a biblioteca GUI JavaFX [41] e a ferramenta *Scene Builder* [42], que permite desenvolver o *layout* de controles gráficos, possibilitando rapidamente fazer um protótipo de interfaces. Esta interface ajuda o administrador a monitorizar a rede e a configurar a rede *overlay* P2P intuitivamente, a Figura 4.1 apresenta a interface fornecida ao administrador da rede. Como observado, a interface integra uma visão da topologia da rede, sobre a qual a rede *overlay* opera, e os *links* e *routers* que a compõem. Na parte de baixo da interface estão disponíveis duas áreas para os relatórios dos *probings*, uma para os resultados e outra para as mensagens de alarme geradas. Para verificar os resultados das medições em tempo real, o administrador poderá escolher alguns filtros para reduzir a informação a ser apresentada e identificar o *probing* específico que quer, como escolher o id especifico de um *probing* ou mostrar todos os *probings* com alarmes de RTT por exemplo, o administrador também pode definir de quanto em quanto tempo os resultados são atualizados. Uma área de texto na parte direita da interface apresenta todas as medições a serem feitas no momento.

A Figura 4.1 ilustra também uma funcionalidade que o administrador tem disponível se passar o cursor por cima dos *links*, podendo ver os respectivos processos de *probing* a que estão a ser submetidos. Para além disso, a interface também destaca todos os *links*

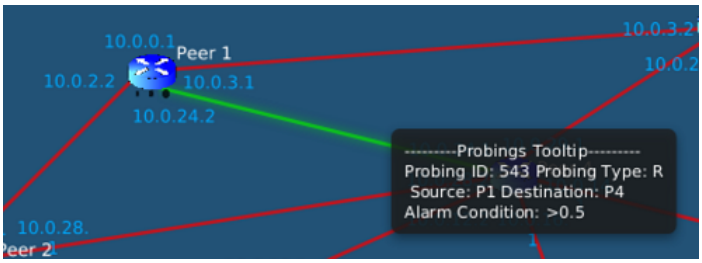


Figura 4.2: *Mouse over* num *link* na interface desenvolvida

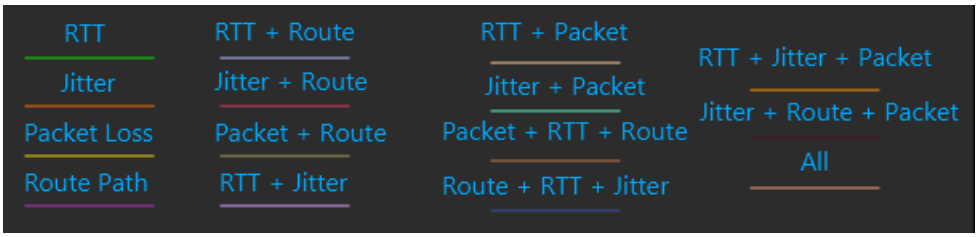


Figura 4.3: Ilustração das várias combinações das linhas que representam as interfaces da interface do sistema

associados a esses *probes* com um efeito *bloom* para ajudar a visualizar melhor o caminho do *probe*, como ilustrado na Figura 4.2.

Cada *link* pode ter várias cores para ajudar o administrador a associar cada *probe* ao seu tipo. Quando o botão do canto superior direito da interface, '?', é pressionado, a informação sobre todas as cores possíveis dependendo dos tipos de *probing* é apresentada ao administrador, como é ilustrado na Figura 4.3.

Usando a interface representada na Figura 4.1, para começar o processo de *probing* o administrador precisa de clicar com o botão esquerdo do rato num *peer* e depois selecionar a interface de destino, especificando, em seguida, outra informação (ex: o tipo de medição, alarmes, tipos de ciclo, *flag*, etc.) como observado na Figura 4.4. O administrador tem também a possibilidade de cancelar o *probing*, clicando com o botão esquerdo do rato num *peer*, escolhendo a opção "*cancel*"e selecionando o *probe* respetivo dos vários *probes* que estão a decorrer.

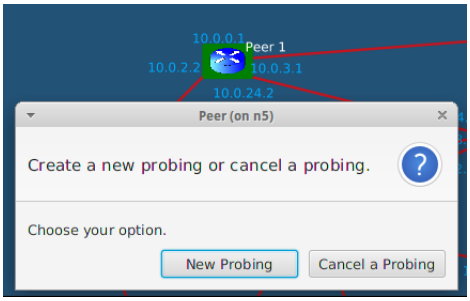


Figura 4.4: Interface de criação ou cancelamento de *probes*

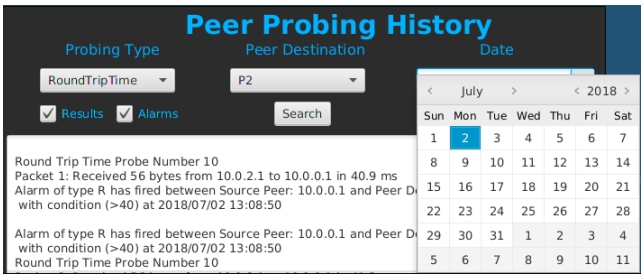


Figura 4.5: Interface do histórico de *probing* de um *peer*

A interface desenvolvida também permite ao administrador verificar o histórico de *probings* da rede ISP, como representado na Figura 4.5. Usando essa interface, o administrador consegue filtrar os resultados por tipo de *probing*, destino de *peer*, data em que o *probe* foi criado e apresentar apenas os resultados ou alarmes que foram acionados.

Foram feitas algumas alterações à interface gráfica do sistema inicial, os resultados de cada *probing* e dos alarmes ficaram mais visíveis e foram adaptados para corresponder melhor aos dados retornados pela chamada de sistema MTR, também foi adicionada uma nova opção que permite ao utilizador da aplicação escolher de quanto em quanto tempo as janelas de resultados de *probing* e de alarmes devem ser atualizadas como ilustrado na Figura 4.6.

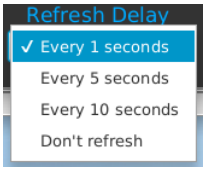


Figura 4.6: Caixa dos valores que a atualização da interface pode ser feita

A janela dos resultados dos *probings* informa ao administrador o ID do *probe*, a origem e destino, o tipo, as condições de alarme e de *loop*, também contém uma tabela com os resultados de cada *probing*, incluindo a média/pior/melhor valor do RTT, a percentagem da perda de pacotes, o desvio padrão, a média do *jitter* e o *interarrival jitter*, os pacotes enviados e o *timestamp* de cada *probing* como ilustrado na Figura 4.7.

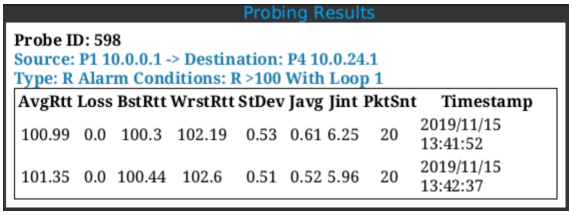


Figura 4.7: Janela dos resultados dos *probings*

A janela dos alarmes, indica o ID do probe, a origem e destino, o tipo, as condições de alarme e de loop, quantas vezes o alarme foi disparado, também é indicado o valor que disparou o alarme e em que timestamp é que ocorreu como ilustrado na Figura 4.8.

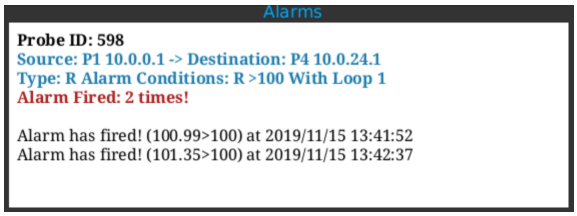


Figura 4.8: Janela dos resultados dos alarmes

Para criar as novas janelas de resultados foi utilizado o componente *WebView* do *javaFX*, permitindo utilizar *Cascading Style Sheets (CSS)* e *HyperText Markup Language (HTML)*, possibilitando facilmente alterar o tipo, cor, estilo, etc, da letra e criar tabelas para uma melhor apresentação dos resultados dos *probings*.

Foram utilizados *Executors* da API *Concurrency* de Java em conjunto com o método estático da classe *Platform* do javaFX *Platform.runLater()* para permitir atualizar as janelas de resultados no tempo que o utilizador do sistema definir com boa performance [43] [44].

Os *Executors* são capazes de correr *tasks* assíncronas e gerir *pools* de *threads* não sendo necessário criar *threads* novas manualmente, são utilizados como uma substituição de nível superior para trabalhar diretamente com *threads*.

O *JavaFX* não é *thread-safe*, não utiliza sincronização para evitar *race conditions* nas *threads* que tentam aceder a compontentes da GUI e as suas variáveis de estado. Desde que seja tudo feito na *thread* da aplicação, não há problemas. Podem ocorrer problemas quando outra *thread* tenta manipular componentes ou variáveis que também são utilizadas na *thread* da GUI.

Foi necessário ser utilizada a função *Platform.runLater* para atualizar a interface gráfica através de uma *thread* que não é de *javaFX*, permitindo colocar os novos resultados numa *queue* sendo estes tratados pela *thread* do *javaFX* quando for possível.

4.3 Base de Dados de Grafos do Sistema

O sistema de monitorização desenvolvido pode ser usado em redes ISP com um grande número de *routers* e *links*. Para além disso, é também possível configurar um grande número de processos de *probing*, recolhendo uma enorme quantidade de dados relacionados que estão armazenados no coordenador da rede *overlay*. Desta forma, a tecnologia de base de dados deve ser selecionada cuidadosamente. Neste contexto, o *Neo4j*, um sistema de gestão de base de dados de grafos, foi usado porque permite operações escaláveis

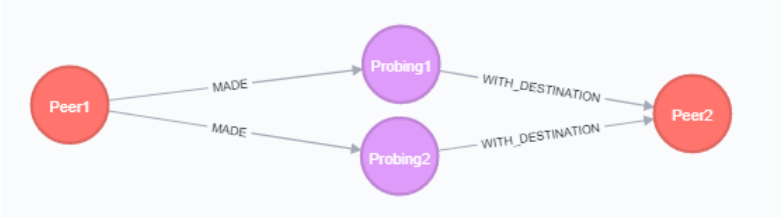


Figura 4.9: Exemplo de armazenamento de um *probing* no *Neo4j*

de leitura e escrita de alto desempenho e a realização de transações rápidas e fiáveis com uma taxa de transferência paralela alta mesmo que os dados cresçam. Também usa uma linguagem de consulta de grafos bastante poderosa e produtiva, *Cypher*. No *Neo4j*, é tudo armazenado na forma de *edge*, nó ou atributo. Cada nó e *edge* podem ter um qualquer número de atributos. Adicionalmente, nós e *edges* podem ser rotulados, permitindo reduzir as pesquisas. A Figura 4.9 ilustra um exemplo simples, que mostra o armazenamento de dois processos de *probing* (*probing* 1 e *probing* 2) feito do *peer1* até ao *peer2* de destino.

4.4 Implementação da Metaliguagem

Nesta secção será descrito o processo de implementação da metalinguagem, onde será justificada a escolha do gerador de parsers ANTLR e expostas as regras criadas.

4.4.1 *Gerador de parsers ANTLR*

Uma das ferramentas utilizadas na criação da metalinguagem foi o gerador de *parsers ANTLR*, uma ferramenta que ajuda na criação de *parsers* onde se pode ler, processar, executar ou traduzir texto estruturado ou ficheiros binários [45]. Como escrever *parsers* à mão é demorado e pode induzir em erros, investigadores já investiram décadas a estudar como criar *parsers* eficientes de gramáticas de alto nível [46].

No sistema desenvolvido nesta dissertação, o *input* do administrador é convertido numa *Abstract syntax tree (AST)*, sendo a representação lógica do *input*. Para obter esta AST é necessário definir um *lexer* e uma gramática de *parser*, o *ANLTR* gera ambos, permitindo a sua utilização em diversas linguagens, que neste caso, será em *Java*.

Existiam algumas alternativas na implementação da linguagem, como por exemplo utilizar expressões regulares, pois são bastante úteis quando se trata de identificar combinações de caracteres em *strings*, mas tem as suas limitações. Uma delas é a falta de recursividade, não se consegue encontrar uma expressão regular dentro de outra, só se para cada nível se programe especificamente, algo que rapidamente se tornava insustentável. Outra limitação

é que não é realmente escalável e é mais complicado de usar expressões regulares do que usar uma ferramenta como o *ANTLR* [47].

Também existia a opção de criar um *parser* de raiz, mas ao utilizar o *ANTLR* a produtividade é maior, pois o tempo ao implementar um *parser* pode ser aplicado a chegar ao objetivo mais rapidamente. Outra vantagem é que o *ANTLR* permite gerar múltiplos *parsers* em diferentes linguagens, podendo criar o *parser* em *Java* e funcionar em *JavaScript* para a mesma linguagem, possibilitando passar a linguagem para uma versão do sistema em *web* rapidamente. Criar um *parser* de raiz também teria as suas vantagens, poderia ter melhor desempenho e produzir melhores mensagens de erro, mas para o sistema em causa o *ANTLR* tem o desempenho suficiente, pois o número de instruções nunca será muito elevado.

Para analisar a nova linguagem é então necessário definir um *lexer* e uma gramática *parser* na qual, como descrito no capítulo anterior, foi usada a forma de BNF para a criar.

4.4.2 *Regras da Metalinguagem*

Para descrever as regras da metalinguagem foi utilizada a ferramenta *rrd-antlr4*, que dando como *input* as regras criadas, gera diagramas *railroad* para cada regra [48]. Serão então descritas as regras, explicando a função e a interligação entre elas, começando pelas regras de *parser* e posteriormente as regras do *lexer*.

A linguagem é constituída por uma ou mais instruções e é obrigatório iniciar por uma instrução, posteriormente vindo o final do ficheiro, como ilustrado na Figura 4.10.

Figura 4.10: Diagrama do início do *parse*.

Cada instrução pode ser de três tipos, de *probing*, para iniciar os *probings*, de *cancel*, para cancelar os *probings* e condicional, para permitir ao administrador indicar ao sistema para iniciar *probings* caso aconteça algo na rede, como ilustrado na Figura 4.11. O campo *skips* significa nova linha, sendo opcional.

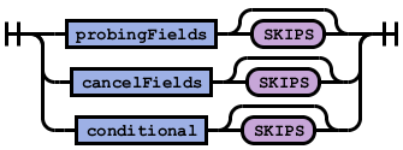


Figura 4.11: Diagrama de uma instrução.

Uma das instruções é a de *probing*, é sempre necessário especificar dois endereços, a métrica, se usa ICMP ou não e o tipo de *loop* (parâmetros explicados na Tabela 3.1). Os campos opcionais são: as indicações do alarme, pois o administrador pode não querer um alarme no *probing*, indicação de se o *probing* é condicional, definição dos pacotes, onde se pode colocar o número de pacotes e o seu tamanho e por fim, *"opsFields"*, o que irá permitir fazer o *probing* entre datas ou iniciá-lo condicionalmente, dependendo dos valores de outro *probing* já existente, como ilustrado na Figura 4.12.

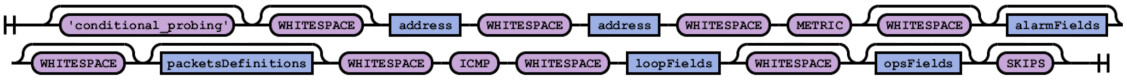


Figura 4.12: Diagrama dos campos para iniciar um *probing*.

Outra instrução possível é a de cancelar *probings* já existentes, podendo também ser condicional e indicada uma data para o cancelamento ocorrer, sendo sempre necessário indicar o identificador do *probing*, como ilustrado na Figura 4.13.

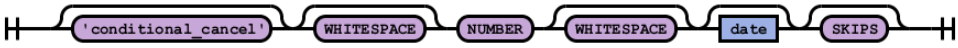


Figura 4.13: Diagrama dos campos para iniciar um cancelamento de *probing*.

A última instrução possível de fazer é a condicional, iniciando-se com a palavra *"conditional"*, e especificando dois números, um que corresponde ao identificador do *probing* e outro que indica quantas vezes um alarme definido previamente no *probing* indicado foi disparado, ao atingir esse valor, uma ou várias instruções novas podem ser iniciadas, podendo esta instrução condicional também ser acionada numa data especifica, como ilustrado na Figura 4.14. Nos campos de *loop* é necessário indicar o tipo de *loop* e pode vir acompanhado com um número, que indica de quanto em quanto tempo os *peers* devem reportar ao coordenador, como ilustrado na Figura 4.15.

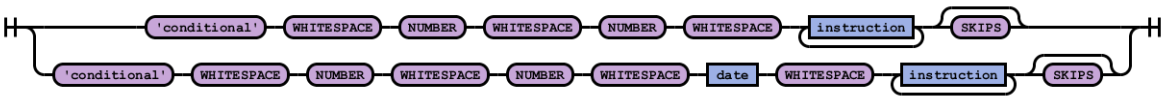


Figura 4.14: Diagrama dos campos para iniciar uma instrução condicional.

Os campos opcionais existentes na instrução de início de *probing*, pode ser de três formas, se foram indicadas duas datas, o *probing* irá ter início na primeira data e posteriormente será feito o seu cancelamento na data seguinte indicada. Se o input conter só uma data, o *probing* é iniciado nessa data. Uma instrução de *probing* também pode conter outra instrução

Figura 4.15: Diagrama dos campos de *loop*.

condicional onde não será necessário indicar o identificador do *probing*, pois a condição será logo aplicada no *probing* que está a ser criado no momento, como ilustrado na Figura 4.16.

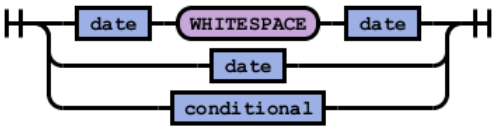


Figura 4.16: Diagrama dos campos opcionais.

Para definir o alarme, o *input* é constituído obrigatoriamente por uma condição ('>', '<', '≥', '≤' e '!=') e opcionalmente por um número para indicar o valor que as métricas têm que alcançar para ativar o alarme. Devido ao tipo de *probing* de *route path*, o número é opcional, pois ao contrário das outras métricas, esta só ativa o alarme se a *path* for alterada e não se um valor for maior, menor ou igual, como ilustrado na Figura 4.17.

Figura 4.17: Diagrama dos campos de alarme.

Como exemplo de geração de uma AST, pelo *ANTLR*, é criado um ficheiro *DOT* usando a funcionalidade *DOTGenerator* do *ANTLR*, que é um ficheiro que contém uma linguagem que descreve grafos, posteriormente é utilizado um visualizador de grafos, *Graphviz* [49], e, dado o *input* de exemplo *10.0.24.2 10.0.26.1 rtt ≥ 150 56 20 yes loop 20 22-07-2019-15-30-00 25-07-2019-15-30-00*, é criado o grafo ilustrado na Figura 4.18. Este *input* inicia um *probing* entre os endereços *10.0.24.2* a *10.0.26.1* da métrica RTT, com campos de alarme de condição '≥ *150*', com pacotes de tamanho 56 kb e 20 pacotes enviados, usando ICMP, tipo de *loop Time Feedback* com tempo de resposta a cada 20 segundos, e são indicadas 2 datas, que informam o coordenador quando o *probing* deve ser iniciado e quando deverá ser cancelado.

Um ficheiro de gramática .g4, que contém as regras do *lexer* e *parser*, é utilizado pelo *ATNLR*, que gera, posteriormente, 3 classes Java necessárias para a interpretação da linguagem criada, um *Lexer*, um *Parser* e um *Listener* encarregue de métodos que são automati-

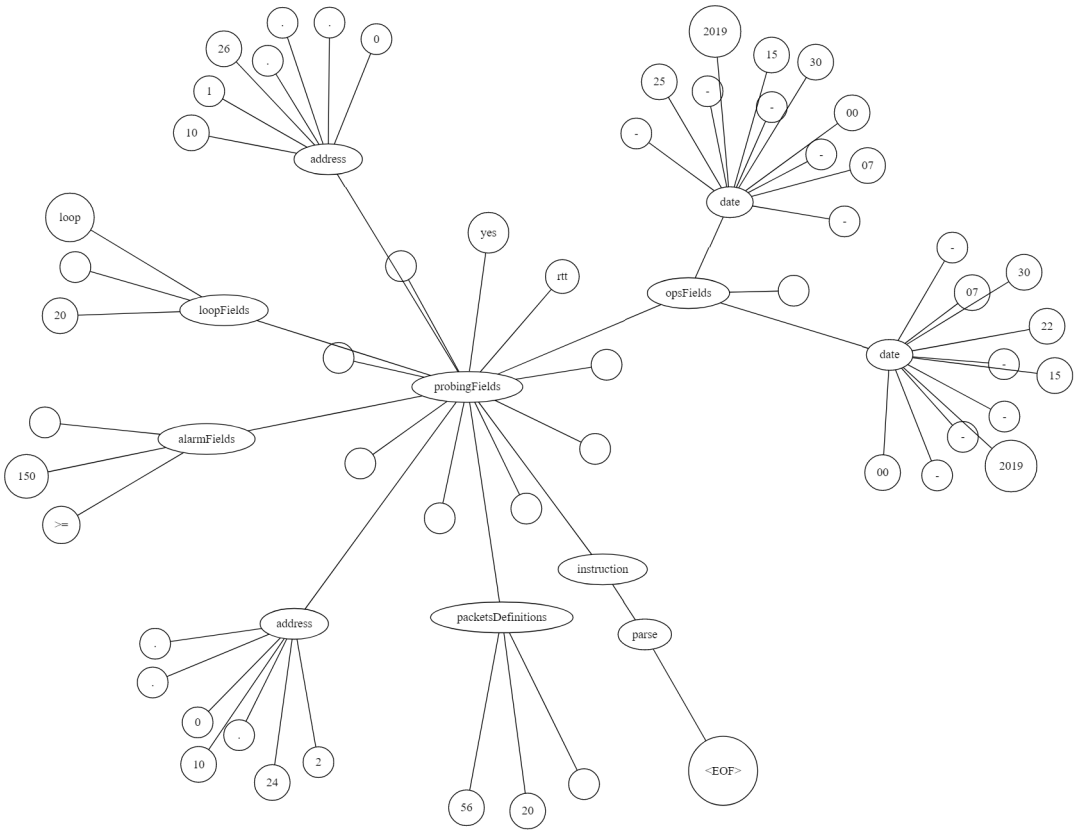


Figura 4.18: Grafo da linguagem criada dado um *input* de exemplo.

camente chamados pelo *ANTLR* se for dado um objeto *walker* para os percorrer, assim, na árvore a ser percorrida pode-se retirar todos os valores que um administrador escreverá na janela de *input* e serem interpretados para correr as instruções.

Foi implementada uma janela nova na interface para os administradores poderem colocar o *input* desejado para programar o sistema, podendo opcionalmente colocar um ficheiro de várias instruções. Esta janela de *input* contém um analisador de erros de sintaxe, indicando a linha e posição de possíveis erros e indicando que campo estaria à espera na respetiva posição.

4.5 Implementação da Otimização de probings do Sistema

Como referido anteriormente, foram feitas otimizações no sistema com o objetivo de reduzir o tráfego gerado pelo sistema. Para otimizar os *probings* de RTT, é feito o somatório do valor do RTT de todos os *links* contidos na *path* de um *probing*, podendo ser reaproveitados os valores de *probings* já existentes no coordenador. Por exemplo, se for iniciado um *probing* de RTT que precisa dos valores de três links que já estão a ser monitorizados, em que o

valor desses links é de 13ms, 5ms, e 20ms, o valor final do novo *probing* será 13+5+20 = 38ms, podendo ser aplicadas condições de alarme sobre este valor caso sejam definidas pelo administrador.

Em relação à otimização dos valores de *packet loss* é necessário aplicar a Equação 1. Onde a variável l_i é o valor de *packet loss* de cada interface, sendo feita a permutação até ao número total de interfaces de resultado da chamada de sistema do MTR.

$$1-(\prod_{i=1}^{interfaces}(1-l_i)) = 1-((1-l_1)\cdot(1-l_2)\cdots(1-l_{númeroDeInterfaces-1})\cdot(1-l_{númeroDeInterfaces})) \quad (1)$$

Como exemplo, dadas duas interfaces a 1 *hop* de distância de cada uma, com valores de 15% e 20% de *packet loss* respetivamente, o valor do *packet loss* do caminho de uma interface até à outra será obtido fazendo (1 - ((1 - 0.15) * (1 - 0.2))) * 100 resultando em 32%.

Foi necessário ter algum cuidado para interpretar os dados resultantes do MTR. Como ilustrado na Figura 4.19, é possível verificar que na segunda linha, na interface *10.0.27.2* existe *packet loss* e RTT significante. Para poder retirar o valor do RTT da interface *10.0.19.2* é necessário retirar o valor do *link* anterior, neste caso, 82.4 - 21, resultando em 61.4 ms. Para o *packet loss* será diferente, para obter a percentagem de *packet loss* da interface *10.0.19.2* é necessário aplicar a Equação 2, onde a 'interface' será o valor do *packet loss* da interface *10.0.19.2* (0,3) e a 'interfaceAnterior' será o valor da interface *10.0.27.2* (0,2). Utilizando a Equação 2, o *packet loss* da interface *10.0.19.2* será então 0.125 * 100 = 12,5%.

Start: 2019-09-25T21:06:42+0100

HOST: n1	Loss%	Snt	Last	Avg	Best	Wrst	StDev
1.|-- 10.0.4.2	0.0%	10	0.2	0.1	0.0	0.2	0.0
2.|-- 10.0.27.2	20.0%	10	21.0	23.0	21.0	33.1	4.1
3.|-- 10.0.19.2	30.0%	10	82.4	82.1	81.6	82.4	0.3
4.|-- 10.0.21.2	50.0%	10	82.8	82.7	81.9	83.6	0.6

Figura 4.19: Exemplo de uma chamada de sistema mtr.

$$1-\frac{(1-interface)}{(1-interfaceAnterior)} \quad (2)$$

Cada *probing* de RTT na nova versão do sistema irá conter o melhor, pior e último valor de cada medição em adição ao já existente, valor médio do RTT.

Para efetuar as devidas otimizações foram criados algoritmos com base nos três casos descritos no capítulo anterior. Para o caso 1, onde são reaproveitados valores de um *probing* existente e para o caso 2, onde também são reaproveitados os valores de um *probing* existente mas é necessário criar um adicional, é utilizado o Algoritmo 1. Para o caso 3,

Algoritmo 1: Caso 1 e 2 da otimização de novo *probing* onde existe um probing no coordenador que contém valores que o novo probing precisa sendo preciso ou não de criar mais probings na rede

```
1  OtimizaçãoDeProbing1e2
   Input      : Novo probing np_i
   Output     : Endereço IP ip
2  listaDeInterfacesCorrespondentes = []
3  interfacesAOtimizar = []
4  foreach probingExistente pe_i ∈Mapa De Probings do
5      numInterfC = interfCorrespondentes(np_i.caminho, pe_i.caminho)
6      if numInterfC > 0&pe_i.length >= np_i.length then
7          foreach novaInterface ∈np_i.interfaces do
8              if pe_i.caminho.contains(novaInterface) then
9                  listaDeInterfacesCorrespondentes.append(novaInterface)
10                 if !((i + 1) >= np_i.length) then
11                     ip = novaInterface     // Será preciso inicar mais probings para
                        efetuar o novo probing pois faltam valores de mais
                        interfaces
12                 else
13                     ip = "none"     // Já existem probings a ser efetuados com os
                        valores necessários para o novo probing
14         if listaDeInterfacesCorrespondentes.lenght == np_i.length then
15             // O novo probing criado só precisa de valores que já estão a ser
               recebidos por outros probings já existentes, não sendo necessário
               iniciar outro probing na rede
16             interfacesAOtimizar.put(np_i.id, listaDeInterfacesCorrespondentes)
17             ip = "none
18         else if !ip.equals(ipDoInputDoUtilizador) then
19             // O probing criado ainda precisa de mais valores para ser feito,
               sendo só alguns valores de probing já existentes reutilizados
20             interfacesAOtimizar.put(pe_i.id, listaDeInterfacesCorrespondentes)
21         if ip == "none" then
22             pe_i.colocarInterfacesAOtimizar(interfacesAOtimizar)
23         else
24             np_i.colocarInterfacesAOtimizar(interfacesAOtimizar)
25 return ip
26
```

onde são cancelados *probings* existentes e criado um com uma *path* maior, é utilizado o Algoritmo 2. Quando o coordenador recebe os valores dos *probings* efetuados, é verificado se é necessário reutilizar estes valores para outros *probings* existentes que possam estar a necessitar de juntar valores de outros *probings*, verificando posteriormente os alarmes, ver Algoritmo 3.

Nestes três algoritmos são expostas várias verificações necessárias fazer sobre todos os *probings* já existentes no coordenador, a comparação do tamanho da *path* de *probings* já existentes com o tamanho da *path* de novos *probings*, será dos fatores mais importantes na determinação da acção que deverá ser tomada para otimizar o sistema.

Algoritmo 2: Caso 3 da otimização de novo *probing*

```
1  OtimizaçãoDeProbing3
   Input     : Novo probing np_i
   Output    : Endereço IP ip
2  listaDeInterfacesCorrespondentes = []
3  interfacesAOtimizar = []
4  foreach probingExistente pe_i ∈Mapa De Probings do
5      numInterfC = interfCorrespondentes(np_i.caminho, pe_i.caminho)
6      if (pe.length < np.lengthnp_i.caminho.contains(pe_i.caminho)) then
7          foreach interfaceExistente ∈pe_i.interfaces do
8              if (np.caminho.contains(interfaceExistente)) then
9                  listaDeInterfacesCorrespondentes.append(interfaceExistente)
10                 ip = ipDoInputDoUtilizador
11         if listaDeInterfacesCorrespondentes.lenght == pe_i.length then
12             interfacesAOtimizar.put(np_i.id, listaDeInterfacesCorrespondentes)
13             ip = ipDoInputDoUtilizador
14         cancelarProbingsExistentes(pe_i.id, np_i.id)
           interfacesAOtimizar.put(pe_i.id, listaDeInterfacesCorrespondentes)
           np_i.colocarInterfacesAOtimizar(interfacesAOtimizar)
15 return ip
16
```

4.6 Implementação da tolerância a falhas da rede *overlay*

Para tornar o sistema tolerante a falhas, foi necessário utilizar algoritmos que permitem o coordenador ou os *peers* recuperarem o seu estado anterior caso haja uma falha. O Algoritmo 4 trata da recuperação do coordenador em caso de falha, sendo executado quando o coordenador é novamente iniciado, é necessário alertar os *peers* que o seu estado foi restabelecido para voltarem a enviar os dados dos *probings* que estavam a fazer. É também necessário verificar se algum *probing* deverá ser cancelado ou iniciado enquanto o coordenador estava desligado, pois pode ter sido corrida uma instrução com data definida quando o coordenador estava desligado. O coordenador armazena todos os *probings* que estão ativos na rede *overlay* num ficheiro para no caso de falha poder utilizá-lo para repor o seu estado anterior.

Tal como o coordenador, os *peers* também terão de ter um ficheiro onde são armazenados todos os *probings* que cada *peer* tem ativos. É necessário alertar o coordenador quando um

peer volta a restabelecer o seu estado. O Algoritmo 5 trata da recuperação dos *Peers* em caso de falha.

Caso haja uma mudança de rota no sistema e exista um *probing* a correr que utilize os *links* dessa rota, é necessário ter as devidas precauções e atualizar todos os valores dos mapas existentes no coordenador e todas as linhas visuais, o Algoritmo 6, é utilizado pelo coordenador no caso de troca ou falha de rota. Este algoritmo é utilizado sempre que é recebido um valor novo de *probing*, sendo verificado se houve alteração de *path*.

4.7 SUMÁRIO

Como referido neste capítulo, as entidades do sistema *overlay* foram implementadas em *Java* e comunicam usando TCP/UDP *java sockets*. A interface gráfica para o administrador foi desenvolvida usando a libraria *JavaFX* onde possibilita o administrador ter um grande controlo sobre o sistema. Foi utilizada uma base de dados de grafos, *Neo4j*, para possibilitar ver o histórico de *probings* e seus resultados no sistema.

Foi feita uma comparação dos vários caminhos possíveis para a criação da metalinguagem para o sistema, explicando razões de ter sido escolhido o gerador de *parsers* ANTLR. As várias regras da metalinguagem tal como o seu funcionamento e exemplos de utilização foram expostos.

A ferramenta MTR foi utilizada para otimizar o sistema, e a implementação dos vários casos foi descrita através de algoritmos, onde foram especificados vários pormenores necessários resolver para alcançar a otimização pretendida.

Por fim, foram descritos os algoritmos utilizados para permitir o sistema ser tolerante a falhas.

Algoritmo 3: Otimização na recepção dos valores de *probings*

```
1  OtimizaçãoDeProbingRecepcao
   Input      : Id e resultadoMtr
2  probing.setLastResult(mtrResult);
3  //Devido à chamada de sistema MTR é necessário retirar o valor do anterior RTT para
     saber o valor da interface, à excepção do primeiro e último valor
4  mtrResult.AvgRtt = mtrResult.AvgRtt − lastRtt;
5  //É utilizada a equação 2 para obter o packetloss de links específicos
6  mtrResult.Loss = (1 − ((1 − (mtrResult.Loss/100))/(1 − (lastLoss/100)))) ∗ 100;
7  //É necessário saber o valor de cada interface caso seja necessário reutilizar os valores
8  mapaDosValoresDeCadaInterface.put(mtrResult.Interface, mtrResult);
9  //Se os valores recebidos não necessitarem ser otimizados pode-se passar para a
     verificação dos alarmes e colocar os valores finais
10 if !probing.optimized then
11 |  if checkAlarmCondition(mtrResult, probing.condition) then
12 |  |  raiseAlarm(probing.id)
13 |  probing.appendResult(mtrResult)
14 |  probing.setInterfaceValues(mapaDosValoresDeCadaInterface)
15 if probing.hasInterfacesToOptimize() then
16 |  //Para cada probing associado para reaproveitar os valores
17 |  foreach probeInterfaces ∈probing.getInterfaceToAdd do
18 |  |  probingComValoresNecessarios =
   |  |   mapaDeProbings.get(probeInterfaces.getKey())
19 |  |  foreach interface ∈probeInterfaces do
20 |  |  |  if !probing.optimized then
21 |  |  |  |  foreach interfaceValueMap ∈probing.getInterfaceValues() do
22 |  |  |  |  |  if interface.contains(interfaceValueMap.getKey())) then
23 |  |  |  |  |  |  valueToAdd[i] = interfaceValueMap.getValue()
24 |  |  |  |  |  |  i++
25 |  |  |  if probing.optimized then
26 |  |  |  |  foreach interfaceValueMap
   |  |  |  |   ∈probingComValoresNecessarios.getInterfaceValues() do
27 |  |  |  |  |  if interface.contains(interfaceValueMap.getKey())) then
28 |  |  |  |  |  |  valueToAdd[i] = interfaceValueMap.getValue()
29 |  |  |  |  |  |  i++
30 |  |  //É aplicada a equação 1 para reutilizar os valores de packet loss e reaproveitados os
   |  |   valores para as restantes métricas
31 |  |  probingFinalValue =
   |  |   calcularValoresFinais(valueToAdd, mapaDosValoresDeCadaInterface)
32 |  |  if checkAlarmCondition(probingFinalValue, probing.condition) then
33 |  |  |  raiseAlarm(probing.id)
34 |  |  probing.appendResult(probingFinalValue)
35 |  |  probing.setInterfaceValues(mapaDosValoresDeCadaInterface)
36 |  |
```

Algoritmo 4: Recuperação do estado do Coordenador

```
1  mapaDeProbingsAtivosPreviamente = ficheiroGetProbingsAtivos()
2  foreach probing ∈mapaDeProbingsAtivosPreviamente do
3      //É necessário verificar se algum probing deverá ser cancelado enquanto o coordenador
         estava desligado
4      if probing.notEnded() then
5          if probing.Started() then
6              //É preciso atualizar todos os mapas que utilizam as informações dos probings com
                 os probings que estavam a ser corridos antes da falha
7              atualizarMapas(probing)
8              mudarLinhasNaInterface(probing)
9          else
10             //Podem haver probings que estavam programados a serem iniciados numa data em
                 que o coordenador estava desligado, sendo necessário inicia-los
11             verificarDatasProbing(probing)
12     else
13         cancelarProbing(probing)
14     alertarPeersQueEstadoFoiRestablecido()
15
```

Algoritmo 5: Recuperação do estado do *Peer*

```
1  mapaDeProbingsAtivosPreviamente = ficheiroGetProbingsAtivos()
2  //É necessário verificar se o Coordenador enviou algum probing enquanto o Peer
     estava desligado
3  probingsRecuperados = checkIfCoordHasProbings()
4  foreach probingR ∈probingsRecuperados do
5      correrChamadaDeSistema(probingR)
6  if !mapaDeProbingsAtivosPreviamente.isEmpty() then
7      foreach probing ∈mapaDeProbingsAtivos do
8          correrChamadaDeSistema(probingR)
```

Algoritmo 6: Reajuste das linhas da interface face uma mudança de rota

```
   Input: caminhoExistente, probeId
1  probe = probeMap.get(probeId)
2  caminhoAnterior = probe.getRoutePath()
3  if !caminhoAnterior.equals(caminhoExistente) then
4  |  //Path sofreu alteração
5  |  probeWithNewPath = probe;
6  |  probeWithNewPath.resetRoutePath(currentRoutePath)
7  |  probeMap.replace(probeId, probe, probeWithNewPath)
8  |  //Retirar linhas existentes do caminho anterior
9  |  changeLinesCancel(LinesToCancel(probeId), probeId)
10 |  //Colocar as novas linhas
11 |  changeLine(probeWithNewPath.getdestIP(), probeId)
12 |  //Atualizar todos os mapas do sistema que utilizavam a rota que foi alterada,
   |    necessário pois as otimizações dependem do valor que está a ser feito probing.
   |    Sendo também preciso verificar se é necessário iniciar novos probings.
13 |  refreshMapValuesAndCheckOptimizations(caminhoAnterior, currentRoutePath)
```

5

ANÁLISE DE RESULTADOS

Para assegurar o correto funcionamento do sistema, serão apresentados casos de teste para comprovar o funcionamento dos *probings* na secção 5.1. Em relação à metalinguagem criada, será verificado se o sistema corre todas as instruções corretamente e, quando houver um erro, se indica corretamente o problema ao administrador, na secção 5.2. Para a otimização do sistema, serão feitas comparações de antes e depois, sendo feita a análise com gráficos comparativos, que se poderão verificar na secção 5.3. Por fim, na secção 5.4, são apresentados os testes para verificar se o sistema se tornou tolerante a falhas.

5.1 TESTES DE *probings*

Os testes do sistema desenvolvido foram feitos utilizando o emulador de rede *Common Open Research Emulator (CORE)*, onde é possível criar topologias de redes ISP e executar aplicações. Usando este emulador, aplicações reais podem ser executadas em nós de rede e *hosts*, que são sistemas baseados em *Linux*. Para todos os casos de testes que serão apresentados, depois de criar uma rede ISP no emulador, foi iniciada a versão do sistema implementada para o coordenador e para vários *peers*. O lado esquerdo da Figura 5.1 mostra uma topologia de rede ISP ilustrativa com *routers* e *links* criados no emulador CORE, o lado direito da Figura 5.1 retrata a interface, anteriormente explicada, onde o administrador visualiza a rede ISP e é capaz de interagir com a rede de monitorização *overlay* P2P.

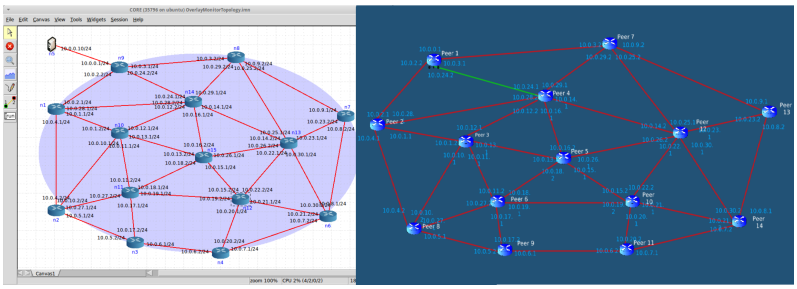


Figura 5.1: Topologia de rede emulada em CORE (lado esquerdo) e a interface gráfica do sistema *overlay* desenvolvido.

Nas subsecções seguintes, serão apresentados três diferentes casos de teste, usando *loop flags* (parâmetro explicado na Tabela 3.1) e tipos de medição de *probes* distintos (RTT, *jitter* e *route path*). Estes comandos foram ativados pelo administrador usando a interface da aplicação mencionada anteriormente como ilustrado na Figura 5.2.

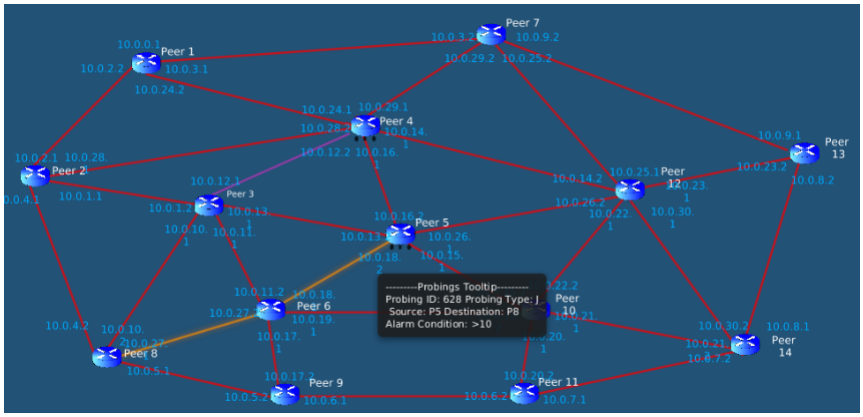


Figura 5.2: Ativação de todos os casos de teste na interface de administrador

5.1.1 *Caso de teste 1 - Probing Instantâneo*

Este caso de teste mostra um *probing* instantâneo de RTT pelo administrador entre o *peer1* e o *peer2* e com a condição de alarme de ">40"ms. Na Figura 5.3 pode ser observado o *output* mostrado na caixa de texto de resultados de *probing* da interface, denotando o RTT medido e o alarme acionado quando o RTT ultrapassa os 40 milissegundos. Com o objetivo de ativar o alarme para teste, foi colocado *delay* de 40 milissegundos no *link* entre o *peer1* e *peer2*, resultando no alarme ser disparado com uma média de RTT de 81.17 ms.

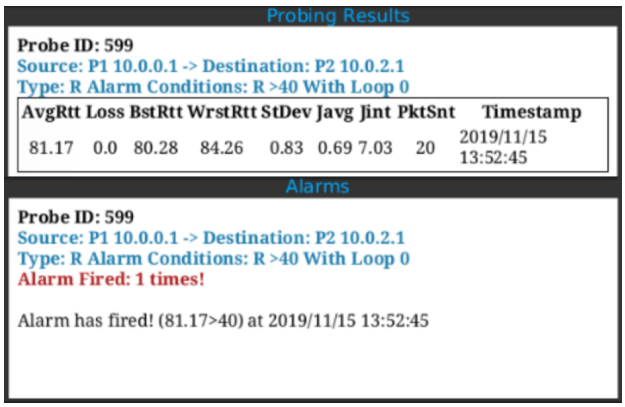


Figura 5.3: *Probing* instantâneo de RTT entre o *peer1* e o *peer2*

5.1.2 *Caso de teste 2 - Feedback Loop Probing*

No segundo caso de teste é apresentado um caso de um *loop probe* com *feedback* constante (tempo de *feedback* de 5 segundos) e com o tipo de medição *route path*. Este *probing* foi configurado no *link* entre o *peer3* e o *peer4* e com o alarme configurado para ser acionado quando uma mudança de *route path* ocorrer. Para forçar este cenário, no emulador CORE, o *link* entre os *peers* vai ser desligado, forçando assim uma nova rota entre os *peers*. A Figura 5.4 mostra claramente que a mudança de *route path* foi detetada pelos alarmes acionados, mostrando a alteração de caminho gerado pelo *link* que foi desligado na topologia de rede.

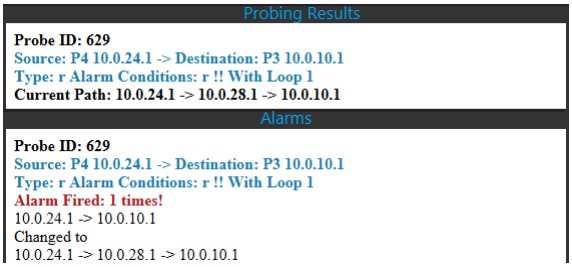


Figura 5.4: *Feedback loop probing* de *route path* entre o *peer3* e o *peer4*

5.1.3 *Caso de teste 3 - Statistical Loop Probing*

Neste terceiro e último caso de teste, é apresentado o *statistical loop probing* com a métrica *jitter*. Como mencionado previamente, este tipo de *loop probing* não fornece resultados constantes ao administrador e se este quiser realmente ver as estatísticas do *probing* a ser feito, ele precisa de acabar o processo de *probing* com um comando *cancel*. Este *probing* é executado entre o *peer5* e o *peer8* e o alarme foi ajustado para ser acionado com a condição ">10", sendo injetados *delays* aleatórios nos *links* no emulador de forma a aumentar o *jitter*. A Figura 5.5 mostra os *outputs* gerados para este caso de teste.

5.2 Testes da Metalinguagem

Foram realizados testes de utilização da metalinguagem, verificando se as instruções eram efetivamente executadas devidamente e se existirem erros verificar se são corretamente indicados.

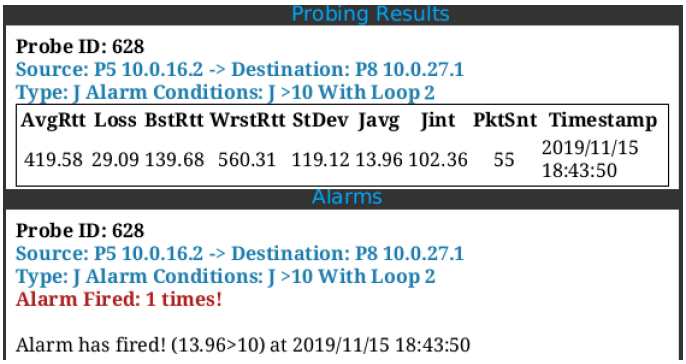


Figura 5.5: *Statistical loop probing* de *jitter* entre o *peer5* e o *peer6*

5.2.1 *Caso de teste 1 - Várias instruções de probing*

A metalinguagem permite programar o sistema de várias formas, como descrito anteriormente. Dando um ficheiro de instruções ou escrevendo na janela de *input* da interface como ilustrado na Figura 5.6, é possível colocar o sistema criado a fazer vários *probings* diferentes rapidamente, para testes, o exemplo contém várias instruções em *loop Time Feedback* com as métricas *jitter* e *packet loss*, e com vários parâmetros de configuração de alarme e *probing* diferentes.

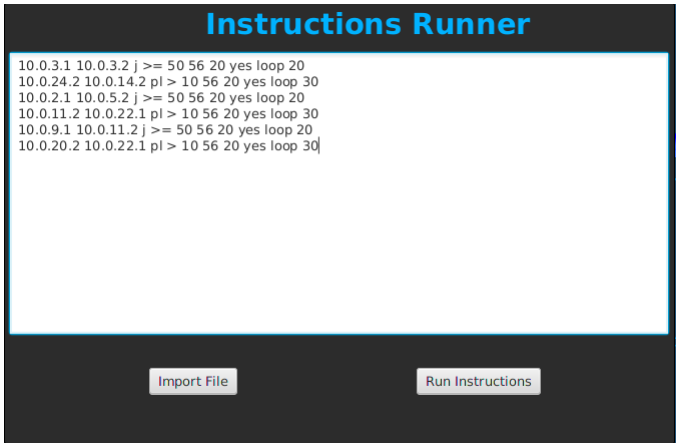


Figura 5.6: Várias instruções para o sistema.

A Figura 5.7, ilustra o estado do sistema depois de serem executadas todas as instruções, recebendo o administrador imediatamente qualquer problema que o sistema tenha detetado na rede, consoante os alarmes definidos pelo administrador.

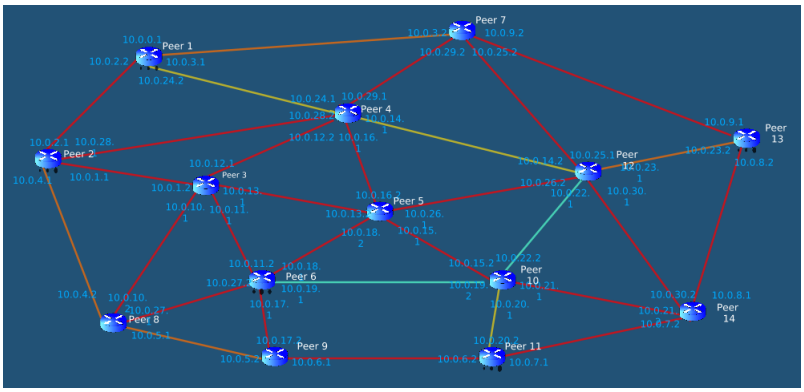


Figura 5.7: Estado do sistema depois de serem executadas várias instruções de iniciar *probings*.

5.2.2 *Caso de teste 2 - Várias instruções de probing canceladas*

Foi feito um caso de teste onde são cancelados todos os *probings* iniciados pelas instruções da Figura 5.6. Especificando o ID de cada *probing*, é então enviado o comando de cancelar os *probings* para cada *peer* respectivo, como ilustrado na Figura 5.8.

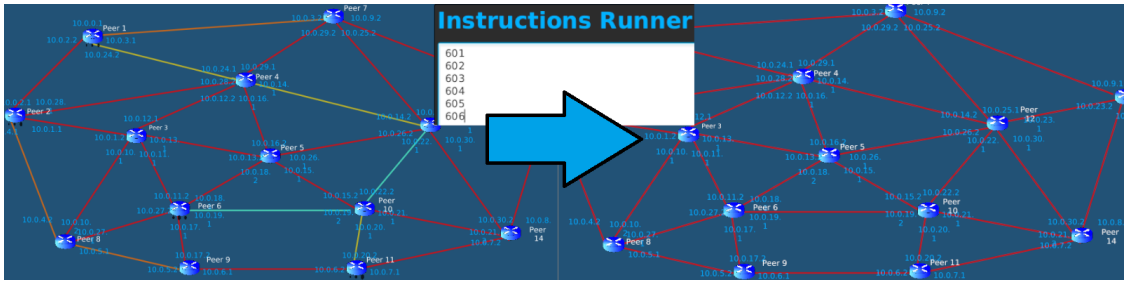


Figura 5.8: Estado do sistema depois de serem executadas várias instruções de cancelar *probings*.

5.2.3 *Caso de teste 3 - Instruções de probing condicional*

Como descrito anteriormente, também é possível criar instruções condicionais, para este caso de teste, foi criada a instrução *conditional 619 5 10.0.0.1 10.0.4.2 rtt > 100 yes loop 20*, que irá iniciar um *probing* de RTT de origem o IP *10.0.0.1* e destino *10.0.4.2* (entre outros parâmetros) quando o *probe* com o ID 619 disparar o alarme 5 ou mais vezes como ilustrado na Figura 5.9.

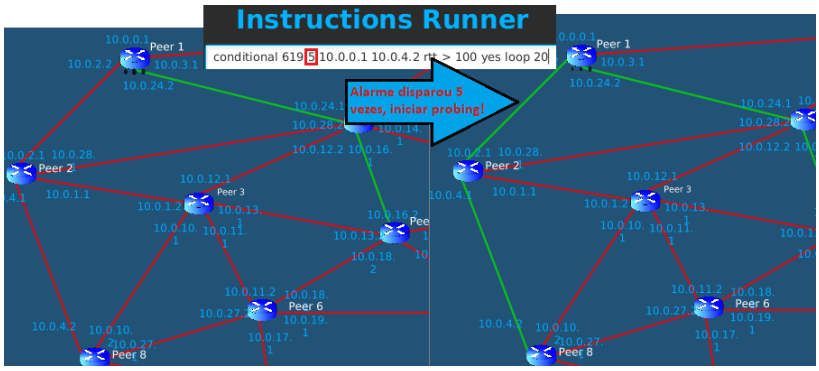


Figura 5.9: Transformação do sistema ao ser executada uma instrução condicional e a condição de alarme ser alcançada

5.2.4 *Caso de teste 4 - Inputs errados do administrador*

Como ilustrado na Figura 5.10, é efetuado um *probing* do IP origem *10.0.11.2* até ao IP destino *10.0.22.1*, com a métrica 'p', neste caso esta métrica não é reconhecida, teria que ser 'pl' ou '*packetloss*', resultando no erro apresentado na figura. O erro indica o tipo de erro, sendo neste caso de sintaxe, a linha, posição e também o tipo de valor que falta, "*METRIC*".

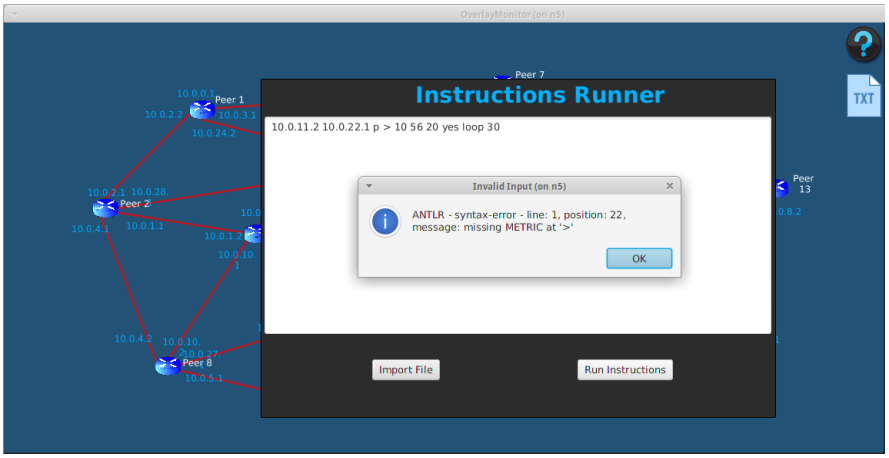


Figura 5.10: Exemplo de um erro ao tentar programar a rede com uma instrução.

5.3 Testes da Otimização do Sistema

Nesta secção, são realizados testes de utilização da otimização do sistema, pois existem *popups* que são apresentados ao utilizador do sistema que apresentam que ação deve ser

tomada. Também foram feitos testes comparativos entre o antes e depois dos processos de otimização.

5.3.1 *Testes de Utilização*

Para ajudar o administrador a perceber o que está a acontecer no sistema, foram criados vários *popups* que o informam sobre o que acontece e lhe permite tomar certas decisões consoante os casos de otimização que podem vir a aparecer.

Um dos casos acontece quando o administrador inicia um novo *probing* e já existem *probings* no sistema a recolher informações relativas a este novo *probing*, sendo usados os valores dos *probings* existentes e criado um novo se existirem valores que o sistema ainda não contenha. Para este caso é mostrado um *popup* indicando o id dos *probings* que contêm os valores do novo *probing*, a rota que o novo *probing* vai tomar e as interfaces que coincidem como ilustrado na Figura 5.11. O administrador tem a possibilidade de escolher se será feita ou não a otimização, pois há casos onde os valores dos parâmetros dos *probings* são diferentes (*feedbacktime*, ICMP, número de pacotes, etc) podendo o administrador não querer realizar a otimização.

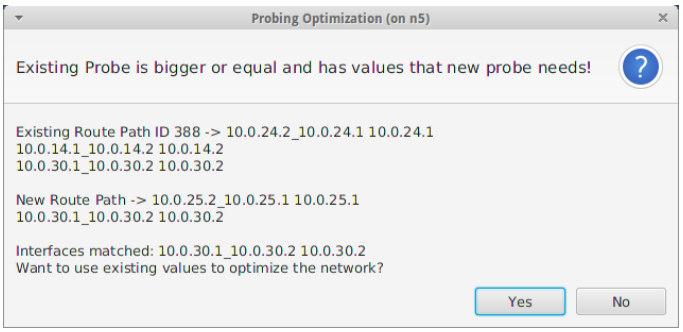


Figura 5.11: *Popup* de aviso ao administrador a indicar que existem *probings* que contêm valores do novo *probing*.

Outro caso que pode ocorrer é quando um novo *probing* tem uma rota que contém interfaces que o sistema já está a fazer *probing* e tem a *path* maior, sendo cancelados *probings* já existentes e criado um novo, reutilizando os valores retirados do *probing* novo para os já existentes. É apresentado um *popup* para o administrador que indica as informações das rotas e que interfaces têm em comum, dando a opção ao administrador de cancelar os *probings* existentes para otimizar a rede, como ilustrado na Figura 5.12.

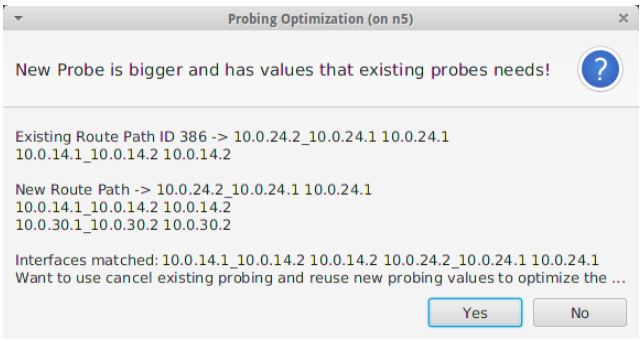


Figura 5.12: *Popup* de aviso ao administrador a indicar que o novo *probing* tem a *path* maior e tem valores que *probings* existentes necessitam.

5.3.2 *Testes Comparativos*

Foram feitos testes comparativos, onde foi comparado o sistema anterior com o sistema depois de serem feitas várias otimizações. A ferramenta de medição de tráfego de redes *vnStat* foi utilizada para este teste, onde é possível escolher que interfaces serão monitorizadas, dando informações sobre cada interface como o envio e receção de *bytes*, máximo, mínimo e média de envio por segundo, também permite a medição de pacotes por segundo [50]. O *vnStat* não faz *sniffing* de nenhum tráfego e assegura uma utilização leve, sendo um perfeito candidato para utilização no CORE.

Foram executados 12 *probings* utilizando a metalinguagem, uma vez com a otimização outra sem a otimização (versão anterior do sistema), como ilustrado na Figura 5.13. Cada *probing* será de RTT ou *packet loss*, focando nestas duas métricas em particular pois foram as que foram mais otimizadas. Para gerar algum tráfego na rede os pacotes foram enviados com tamanho de 908 *bytes*, enviando 20 pacotes a cada 20 segundos, para estar continuadamente a gerar tráfego na rede.

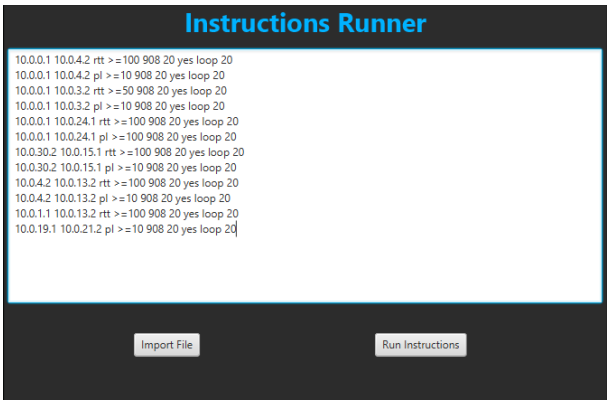


Figura 5.13: Instruções executadas para análise de testes da otimização da rede

Todas as instruções executadas no sistema com o objetivo de comparar a otimização do sistema anterior com o otimizado, resultaram nos vários *probings* da Figura 5.14, onde se pode verificar que existem 6 *probings* de RTT e 6 *probings* de *packet loss*. Foram escolhidas estas duas métricas pois foram as mais otimizadas, devido à não possibilidade de reaproveitar os valores do *jitter* nem *routpath* para outros *probings* diferentes.

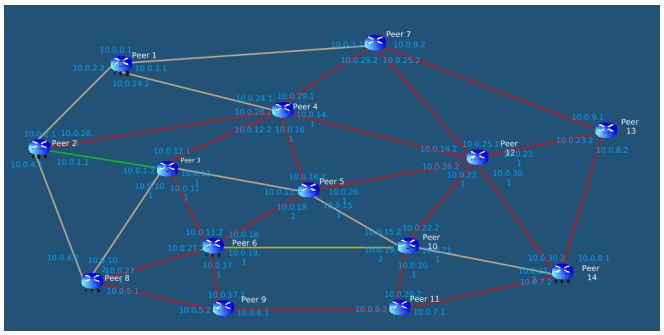


Figura 5.14: Instruções executadas para análise de testes da otimização da rede

Foi executada a ferramenta *vnStat* na interface *etho* (*10.0.0.1*) do coordenador e, posteriormente, nas interfaces *etho* (*10.0.2.2*), *eth1* (*10.0.3.1*) e *eth2* (*10.0.24.2*) de um *peer* (n9), como ilustrado na Figura 5.15. Foram comparados vários tipos de valores, tais como: a velocidade de transferência, os *bytes* transferidos, o número de pacotes por segundos e o número de pacotes enviados entre a versão otimizada e não otimizada, durante um período de 10 minutos, resultando em vários dados que permitiram criar diferentes gráficos comparativos. Estes gráficos serão apresentados de seguida com a legenda:

- rx(NO), valor de *download* dos *links* em causa no sistema anterior, sem otimização
- rx(O), valor de *download* dos *links* em causa no sistema otimizado
- tx(NO), valor de *upload* dos *links* em causa no sistema não otimizado
- tx(O), valor de *upload* dos *links* em causa no sistema otimizado

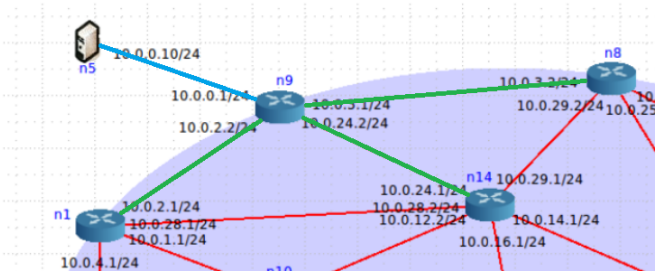


Figura 5.15: Imagem da rede emulada no CORE para comparação de otimizações

No gráfico da Figura 5.16, é comparado o tráfego que percorre a interface do coordenador com otimização e sem otimização. Em média, na versão do sistema sem otimização, o valor de *download* é de 19,73 kbit/s e o valor de *upload* de 11 kbit/s, na versão do sistema otimizado o valor de *download* é de 2,13 kbit/s e o valor de *upload* 1,16 kbit/s. Observa-se uma diminuição de 89% do tráfego de *download* e 89% do tráfego de *upload* na interface do coordenador, esta otimização elevada no *link* do coordenador deve-se principalmente pelo facto de que previamente era enviado o resultado sempre que um *ping* era feito, com a nova implementação feita com a chamada de sistema MTR só é enviado o resultado depois de serem enviados todos os pacotes definidos pelo administrador entre os *peers*.

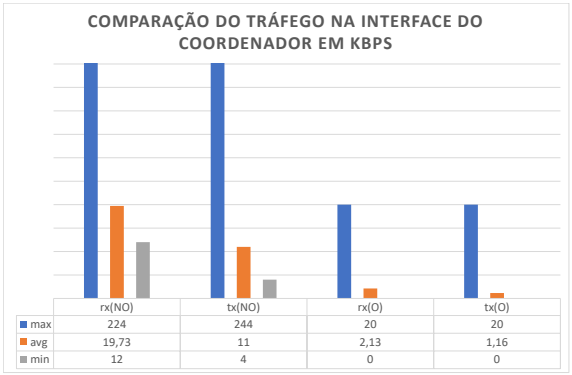


Figura 5.16: Comparação do tráfego na Interface do coordenador.

No gráfico da Figura 5.17, são comparados os *bytes* transferidos na interface do coordenador com otimização e sem otimização. Na versão do sistema sem otimização, o valor de *bytes* totais recebidos é de 1.45 MiB e o valor *bytes* totais enviados de 825 KiB, o mesmo valor na versão do sistema otimizado é de 231 KiB *bytes* totais recebidos e 145 *bytes* totais enviados. Observa-se uma diminuição de 89% de *bytes* recebidos e de 89% de *bytes* enviados na interface do coordenador.

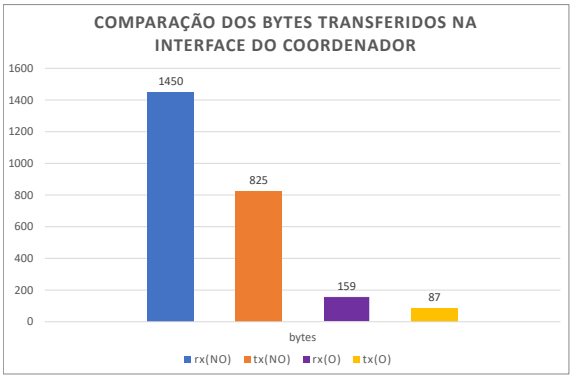


Figura 5.17: Comparação dos *bytes* transferidos na Interface do coordenador.

No gráfico da Figura 5.18, é comparado o número de pacotes por segundo na interface do coordenador com otimização e sem otimização. Em média, na versão do sistema sem otimização, o valor de pacotes recebidos é de 24 p/s e o valor de envio de 20 p/s, na versão do sistema otimizado o valor em média de pacotes recebidos é de 2 p/s e o valor de envio de 1 p/s. Observa-se uma diminuição de 92% no número de pacotes recebidos por segundo e 95% no número de pacotes enviados na interface do coordenador.

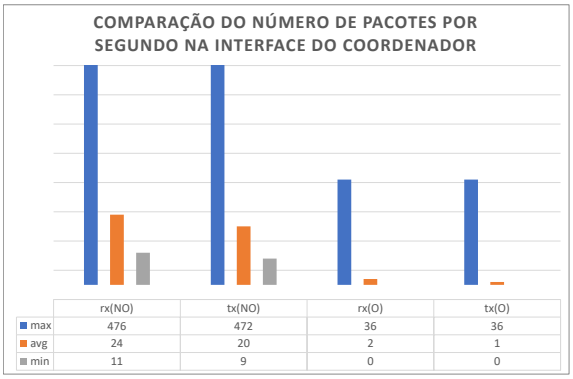


Figura 5.18: Comparação do número de pacotes por segundo na Interface do coordenador.

No gráfico da Figura 5.19, é comparado o número de pacotes totais enviados na interface do coordenador com otimização e sem otimização. Na versão do sistema sem otimização, foram recebidos 14495 pacotes e enviados 12445, na versão do sistema otimizado foram recebidos 1268 pacotes e enviados 1067. Observa-se uma diminuição de 91% no número de pacotes recebidos por segundo e 91% no número de pacotes enviados na interface do coordenador.

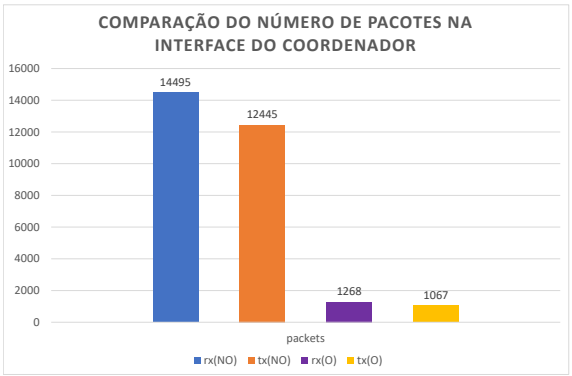


Figura 5.19: Comparação do número de pacotes na Interface do coordenador.

No gráfico da Figura 5.20, é comparado o tráfego que percorre as 3 interfaces do *peer* n9 com otimização e sem otimização. Em média, na versão do sistema sem otimização, o valor de *download* é de 33,28 kbit/s e o valor de *upload* de 28,99 kbit/s, na versão do sistema

otimizado o valor de *download* é de 14,03 kbit/s e o valor de *upload* 14,76 kbit/s. Observa-se uma diminuição de 58% do tráfego de *download* e 49% do tráfego de *upload* nas interfaces do *peer* n9. Nos *links* dos *peers* a otimização não é tão elevada como no *link* do coordenador porque, neste caso, maior parte da otimização deriva da reutilização dos valores de *probings* já existentes e não devido à redução de pacotes enviados ao coordenador como nos casos anteriores.

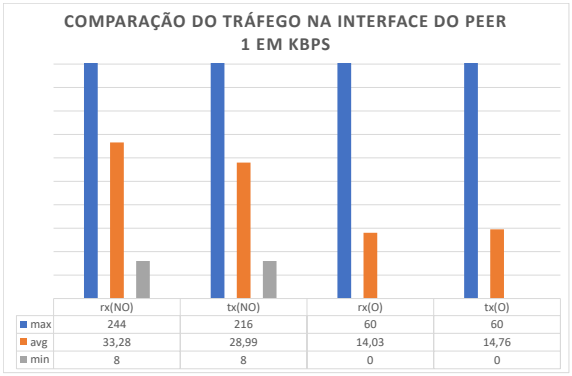


Figura 5.20: Comparação do tráfego na Interface do *Peer* 1.

No gráfico da Figura 5.21, são comparados os *bytes* transferidos nas interfaces do *peer* n9 com otimização e sem otimização. Na versão do sistema sem otimização, o valor de *bytes* totais recebidos é de 2478 MiB e o valor *bytes* totais enviados de 2177 KiB, o mesmo valor na versão do sistema otimizado é de 1050 KiB *bytes* totais recebidos e 1104 *bytes* totais enviados. Observa-se uma diminuição de 58% de *bytes* recebidos e de 49% de *bytes* enviados nas interfaces do *peer* n9.

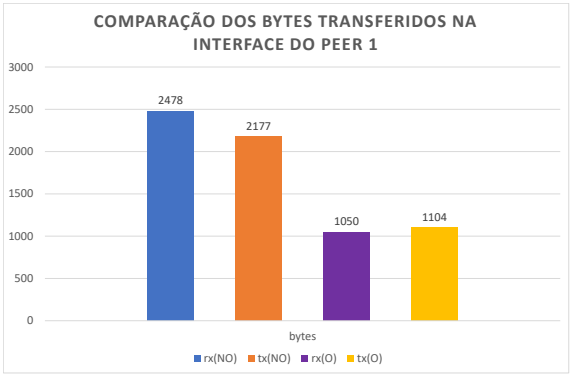


Figura 5.21: Comparação dos *bytes* transferidos na Interface do *Peer* 1.

No gráfico da Figura 5.22, é comparado o número de pacotes por segundo nas interfaces do *peer* n9 com otimização e sem otimização. Em média, na versão do sistema sem otimização, o valor de pacotes recebidos é de 15 p/s e o valor de envio de 14 p/s, na versão

do sistema otimizado o valor em média de pacotes recebidos é de 3 p/s e o valor de envio de 3 p/s. Observa-se uma diminuição de 80% no número de pacotes recebidos por segundo e 79% no número de pacotes enviados nas interfaces do *peer* n9.

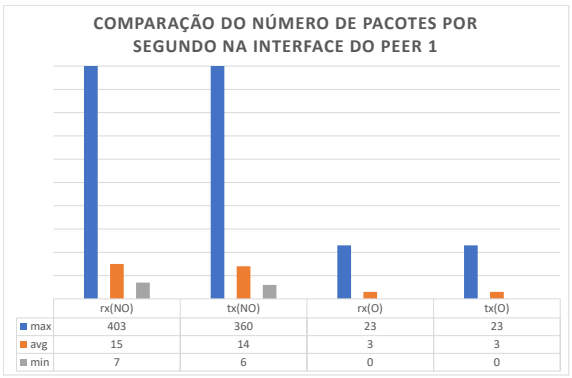


Figura 5.22: Comparação do número de pacotes por segundo na Interface do *Peer* 1.

No gráfico da Figura 5.23, é comparado o número de pacotes totais enviados na interface do coordenador com otimização e sem otimização. Na versão do sistema sem otimização, foram recebidos 9638 pacotes e enviados 8674, na versão do sistema otimizado foram recebidos 2408 pacotes e enviados 2365. Observa-se uma diminuição de 75% no número de pacotes recebidos por segundo e 73% no número de pacotes enviados nas interfaces do *peer* n9.

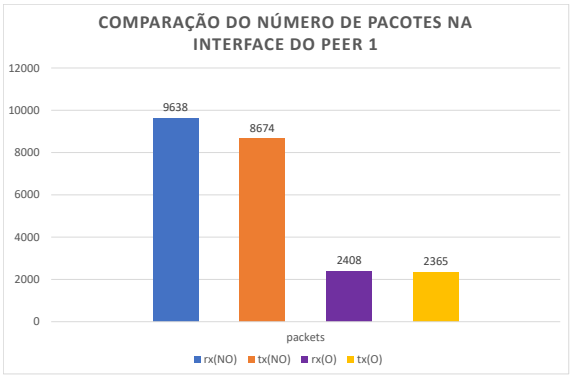


Figura 5.23: Comparação do número de pacotes na Interface do *Peer* 1.

Como se pode verificar, a otimização foi bem sucedida, reduzindo bastante o tráfego da rede e obtendo os mesmos resultados finais de cada *probing*. A discrepância dos valores deve-se:

- Na versão inicial do sistema era enviado o resultado ao coordenador sempre que um *ping* era feito, resultando em tráfego desnecessário a atravessar a rede. Na versão

otimizada do sistema, com a ferramenta *mtr*, só é enviado o resultado de cada *probing* depois de enviar os pacotes definidos pelo administrador entre o *peer* origem e *peer* destino;

- Previamente, sempre que era criado um *probing*, não eram reutilizados nenhuns valores de outros *probings* já existentes, na versão otimizada só é necessário correr um *probing* para obter resultados sobre as métricas todas de uma *path*, onde anteriormente para obter os resultados de todas as métricas dessa mesma *path* seria necessário correr quatro *probings* diferentes. Para casos de *probings* de RTT e *packet loss*, no sistema otimizado, são também reutilizados todos os valores dos *links* que já estão a ser monitorizados, sendo aplicado um dos três casos de composição de métricas referidos nos capítulos anteriores, assim, *probings* de RTT e *packet loss*, são otimizados mesmo se não existirem *probings* com exatamente a mesma *path*, basta existirem os valores dos *links* que compõe a *path* dos novos *probings*;

- O *payload* da versão inicial do sistema era maior quando ocorria um alarme pois este ia no resultado dos *probings*. Na versão otimizada os *peers* já não precisam de enviar o alarme nos resultados dos *probings* pois é o coordenador que verifica os alarmes ao receber os valores dos *peers*.

5.4 Testes da Tolerância a Falhas do Sistema

Para testar se o sistema se tornou tolerante a falhas, foi colocado o sistema em funcionamento com vários probings iniciais, sendo posteriormente injetadas falhas no coordenador e em *peers* com o objetivo de verificar se estes retornam ao seu estado anterior quando forem novamente iniciados.

Ao reiniciar o coordenador, desligando abruptamente o processo que corria o sistema e iniciando-o de novo, é apresentado um *popup* indicando algumas informações do seu estado anterior, como ilustrado na Figura 5.24, onde são apresentados quantos *probings* estava a realizar. De seguida, foram reiniciados vários *peers* de forma a testar se voltavam ao estado anterior. É possível verificar que o estado do coordenador e *peers* pré falha foi retomado, estando todos os *probings* a correr devidamente.

Para testar se o sistema se adaptava corretamente a uma falha num *link*, foi iniciado um *probing* de RTT, do *Peer* 1 até ao *Peer* 12 e posteriormente introduzida uma falha no *link* que interliga o *Peer* 4 e o *Peer* 12. Como ilustrado na Figura 5.25, o sistema deteta a falha e automaticamente é atualizada a GUI do sistema para refletir a mudança de *path*.

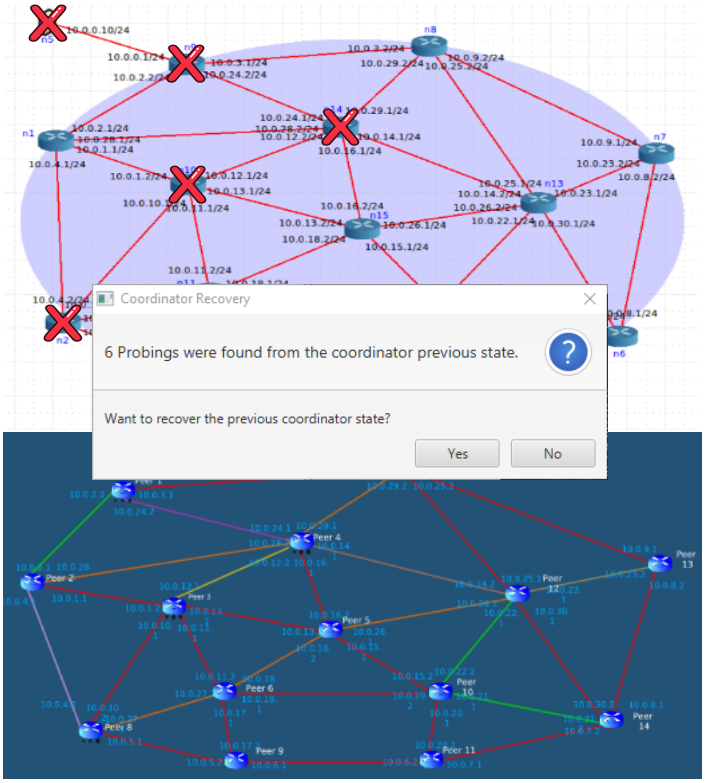


Figura 5.24: Ilustração da recuperação do estado anterior do sistema depois de serem criadas falhas.

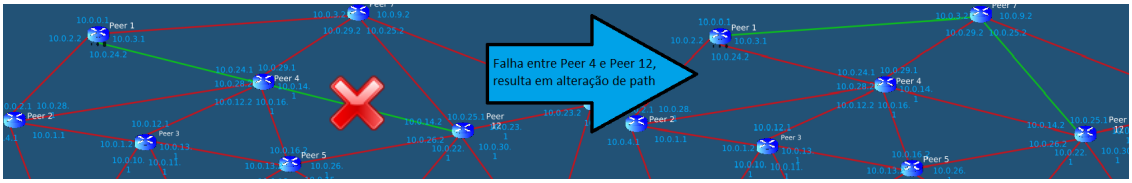


Figura 5.25: Ilustração da alteração de *path* na interface do sistema.

5.5 Sumário

Foram realizados vários testes de *probing* diferentes a fim de testar as funcionalidades de monitorização base do sistema. Também foram realizados testes de utilização da meta-linguagem, verificando se os vários tipos de instruções criados funcionam corretamente, e, no caso de o utilizador se enganar numa instrução, este, ser devidamente informado do ocorrido. Posteriormente foram feitos vários testes comparativos da otimização do sistema, comparando o sistema antes e depois de ser otimizado, resultando em uma redução de tráfego bastante significativa, também foram feitos testes de utilização da GUI para o administrador ter a possibilidade de otimizar ou não os *probings* no momento da criação.

Por fim, foram realizados testes para verificar se o sistema se tornou tolerante a falhas, introduzindo falhas nos *peers* e coordenador e verificando que efetivamente retornavam ao estado pré falha.

6

CONCLUSÕES

Neste capítulo serão referidas as conclusões obtidas resultantes do trabalho desenvolvido durante esta dissertação, como também algumas dificuldades encontradas, na secção 6.1. Serão referidas as principais contribuições na secção 6.2 e apresentado possível trabalho futuro para melhorar o sistema na secção 6.3.

6.1 RESUMO DO TRABALHO DESENVOLVIDO

Neste trabalho, foi necessária uma base de conhecimento para o desenvolvimento das novas funcionalidades do sistema tal como o seu melhoramento. Foram estudadas várias tecnologias de monitorização existentes, *packet probing*, *packet sniffing*, SNMP, e monitorização por *flows*, sendo apresentadas várias vantagens e desvantagens de cada um e para que objetivo se equadravam, sendo o *packet probing* o utilizado neste projeto. Os requisitos de sistemas de monitorização de redes também foram estudados, onde foram descritas várias métricas diferentes e as diferentes formas de as medir, sendo as principais deste projeto: RTT, *jitter*, *packet loss* e *routepath*. Também foram estudadas várias funcionalidades existentes nos sistemas de monitorização de redes. Como o projeto é baseado num sistema P2P, foi fundamental enriquecer a base de conhecimento com as diferentes arquiteturas existentes P2P, foi referida a sua aplicabilidade nos sistemas de monitorização, sendo a arquitetura centralizada a utilizada neste projeto. Foram apresentados quatro projetos relacionados com o tema e objetivos deste projeto, onde também foi feita a sua comparação.

A arquitetura do sistema foi detalhada, especificando os vários componentes existentes nas entidades *peer* e coordenador, e sendo exposta como é feita a comunicação entre elas, sendo referidas todas as possibilidades que um administrador de ISP tem para interagir com o sistema. Foram descritos os passos para a criação de uma metalinguagem que permite um administrador pré-programar, onde foram inicialmente descritos os três tipos de instruções principais criadas para interagir com o sistema e foi utilizada a técnica de notação BNF para ajudar a definir a gramática. Foram também apresentados e descritos vários exemplos da utilização da metalinguagem, descrevendo como iniciar ou cancelar

um *probing* com vários parâmetros distintos. O processo de redução do tráfego gerado pela rede P2P foi exposto, onde foram apresentados três casos diferentes que foram necessários resolver para atingir este objetivo, sendo este uma das maior dificuldades deste projeto, pois existiram várias particularidades a alcançar que foram referidas anteriormente. Foram também explicados os vários benefícios de tornar um sistema tolerante a falhas, bem como que processos tiveram que ser tomados caso o coordenador ou um dos *peers* falhe.

O processo de implementação da arquitetura do sistema e mecanismos desenvolvidos foi exposto, onde foram apresentadas ferramentas utilizadas e a razão da sua escolha. O gerador de *parsers ANTLR* utilizado na criação da metalinguagem e o MTR para a otimização do sistema foram as principais ferramentas utilizadas. Para aprofundar como foi feita a implementação destes mecanismos, foram apresentados vários algoritmos juntamente com a sua explicação.

Com o objetivo de assegurar o correto funcionamento do sistema e de todos os mecanismos implementados, foi utilizado o emulador CORE. Foram feitos casos de testes para três tipos de *probings*, sendo injetado *delay* e *packet loss* em certos *links* de modo a testar os alarmes do sistema. Vários testes de utilização da metalinguagem terão sido feitos, verificando se os comandos eram efetivamente executados devidamente e, se houvesse algum erro em alguma instrução, verificar se os erros eram corretamente indicados. Para comparar o tráfego gerado pela versão inicial do sistema e o sistema otimizado foram feitos vários testes. Verificou-se uma redução bastante significativa do tráfego gerado, devido à versão otimizada reutilizar os valores de *probings* já existentes e porque na versão do sistema inicial era enviado o resultado ao coordenador pelos *peers* sempre que um *ping* era feito, na versão do sistema otimizado só é enviado o resultado de cada *probing* depois de enviar os pacotes definidos pelo administrador entre o *peer* origem e *peer* destino. Para testar se o sistema se tornou tolerante a falhas, foram injetadas falhas no coordenador e em *peers* com o objetivo de verificar se estes retornavam ao seu estado anterior quando fossem novamente iniciados, o que foi confirmado.

6.2 Principais Contribuições

Como principais contribuições desta dissertação está a implementação e demonstração de um sistema P2P de monitorização de redes que é versátil, otimizado e com uma interface intuitiva e simples de utilizar por administradores de redes ISP. Apesar de existirem alguns projetos com algumas particularidades relacionadas com este, não existe muita informação relacionada com o desenvolvimento e implementação de sistemas P2P para monitorização de redes, sendo este projeto uma boa contribuição para quem queira fazer algo relacionado com este tipo de sistemas.

6.3 TRABALHO FUTURO

Como possível trabalho futuro, podem ser adicionados mais tipos de *probings*. Também seria interessante estudar a expansão e testar o sistema desenvolvido em cenários envolvendo ambientes multi-ISP, onde vários administradores podem cooperar na gestão das operações das redes e na deteção das anomalias. A interface do sistema poderia ser enriquecida com uma melhor visualização dos alarmes e com novas funcionalidades como adicionar novos *peers* facilmente. Também poderia ser feita uma análise dos níveis de alerta gerados e dos tempos de recuperação da rede mediante os diferentes protocolos de *routing* utilizados pelo ISP (*Routing Information Protocol (RIP)*, *Open Shortest Path First (OSPF)*, etc.).

BIBLIOGRAFIA

[1] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, and Aiko Pras. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014.

[2] *Flowmon Network Performance Monitoring*, . Disponível em https://www.flowmon.com/en/solutions/use-case/netflow-ipfix/network-performance-monitoring.

[3] Vidar Faltinsen and Gro-Anita Vindheim. Framework conditions and requirements for network monitoring in campus networks. *Technical Report GN3-NA3-T4-UFS128*, pages 4–9, 2011.

[4] *Network Management Costs Overshoot User Needs*. Disponível em http://solarwinds-marketing.s3.amazonaws.com/solarwinds/whitepapers/IDG_NETMAN_QuickPulse_Survey.pdf.

[5] Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. Architecture of peer-to-peer systems. In *Peer-to-peer Computing*, pages 11–37. Springer, 2010.

[6] Jörg Eberspächer, Rüdiger Schollmeier, Stefan Zöls, and Gerald Kunzmann. Structured p2p networks in mobile and fixed environments. *proceedings of HET-NETs*, 4, 2004.

[7] *perfSONAR*. Disponível em https://www.perfsonar.net/.

[8] Rafiullah Khan and Sarmad Ullah Khan. Design and implementation of an automated network monitoring and reporting back system. *Journal of Industrial Information Integration*, 9:24–34, 2018.

[9] Sihyung Lee, Kyriaki Levanti, and Hyong S Kim. Network monitoring: Present and future. *Computer Networks*, 65:84–98, 2014.

[10] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[11] Maitreya Natu and Adarshpal S Sethi. Active probing approach for fault localization in computer networks. In *End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on*, pages 25–33. IEEE, 2006.

[12] Matthew J Luckie, Anthony J McGregor, and Hans-Werner Braun. Towards improving packet probing techniques. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 145–150. ACM, 2001.

[13] Van Jacobson and S Deering. Traceroute tool, 1989.

[14] Ajay Tirumala Les Cottrell Tom Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Presented at Passive and Active Monitoring Workshop (PAM 2003)*, 2003.

[15] Pallavi Asrodia and Vishal Sharma. Network monitoring and analysis by packet sniffing method. *International Journal of Engineering Trends and Technology (IJETT)*, 4(5), 2013.

[16] Venkat Mohan, YR Janardhan Reddy, and K Kalpana. Active and passive network measurements: a survey. *International Journal of Computer Science and Information Technologies*, 2(4):1372–1385, 2011.

[17] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[18] *PRTG*. Disponível em https://www.paessler.com/.

[19] Paul Moceri. Snmp and beyond: A survey of network performance monitoring tools, 2006. Disponível em https://www.cse.wustl.edu/~jain/cse567-06/net_traffic_monitors2.htm.

[20] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[21] Douglas Mauro and Kevin Schmidt. *Essential SNMP: Help for System and Network Administrators*. "O'Reilly Media, Inc.", 2005.

[22] Chavee Issariyapat, Panita Pongpaibool, Sophon Mongkolluksame, and Koonlachat Meesublak. Using nagios as a groundwork for developing a better network monitoring system. In *Technology Management for Emerging Technologies (PICMET), 2012 Proceedings of PICMET'12:*, pages 2771–2777. IEEE, 2012.

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

[24] Luca Deri and Stefano Suin. Ntop: Beyond ping and traceroute. In *International Workshop on Distributed Systems: Operations and Management*, pages 271–283. Springer, 1999.

[25] *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*, . Disponível em https://www.ietf.org/rfc/rfc7011.txt.

[26] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[27] Peter Phaal, Sonia Panchen, and Neil McKee. Rfc3176: Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks, 2001.

[28] *Neo4j*. Disponível em https://neo4j.com/.

[29] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

[30] Stefan Saroiu, Krishna P Gummadi, and Steven D Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia systems*, 9(2):170–184, 2003.

[31] David P Anderson. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[32] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*, pages 46–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44702-3. doi: 10.1007/3-540-44702-4_4.

[33] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings first international conference on peer-to-peer computing*, pages 99–100. IEEE, 2001.

[34] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.

[35] *RIPE Atlas*. Disponível em https://atlas.ripe.net/.

[36] *Sam Knows*. Disponível em https://www.samknows.com/.

[37] *NLNOG RING*. Disponível em https://ring.nlnog.net/.

[38] Miguel Silva, Ricardo Mendonça, and Pedro Sousa. A p2p overlay system for network anomaly detection in isp infrastructures. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2019.

[39] A. F. Kurgaev and S. N. Grigoriev. Metalanguage of normal forms of knowledge. *Cybernetics and Systems Analysis*, 52(6):839–848, Nov 2016. ISSN 1573-8337. doi: 10.1007/s10559-016-9885-3.

[40] *MTR*. Disponível em http://www.bitwizard.nl/mtr/.

[41] *JavaFX*. Disponível em https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm.

[42] *Scene Builder*. Disponível em https://gluonhq.com/products/scene-builder/.

[43] *Java Executors*. Disponível em https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/.

[44] *Programming with Threads on JavaFX*. Disponível em http://math.hws.edu/javanotes/c12/s2.html.

[45] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[46] Terence Parr and Kathleen S Fisher. Ll (*): The foundation of the antlr parser generator draft. *PLDI*, 2011.

[47] *Tutorial ANTLR*. Disponível em https://tomassetti.me/antlr-mega-tutorial/.

[48] *Diagramas Railroad para ANTLR*. Disponível em https://github.com/bkiers/rrd-antlr4.

[49] *Vizualizador DOT Graphviz*. Disponível em https://dreampuf.github.io/GraphvizOnline.

[50] *vnStat*. Disponível em https://humdi.net/vnstat/.