TÉCNICO LISBOA

Agente SNMP para SR Linux

Guilherme André Soares Cardoso

Dissertação para obtenção do Grau de Mestre em

Engenharia de Telecomunicações e Informática

Orientadores: Prof. Rui Jorge Morais Tomaz Valadas
Eng. Luís Miguel Leal Simões

Júri

Presidente: Prof. Ricardo Jorge Fernandes Chaves
Orientador: Prof. Rui Jorge Morais Tomaz Valadas
Vogal: Prof. Paulo Rogério Barreiros D'Almeida Pereira

Outubro 2022

Agradecimentos

Em primeiro lugar quero agradecer à minha Mãe e ao meu Pai por acompanharem-me desde sempre, por estarem sempre presentes e por serem o meu exemplo.

À minha avó por tudo o que me ensinou e pelo seu exemplo de vida.

Agradeço a todos os meus colegas e amigos por todos os momentos de amizade e interajuda partilhados.

À equipa da Nokia, nomeadamente ao Eng. Miguel Simões e ao Eng. Sérgio Santos pelo apoio e suporte na concretização desta dissertação.

E finalmente um agradecimento muito especial ao professor Rui Valadas pela sua orientação, pela sua disponibilidade, apoio total, pelas suas críticas e colaboração.

Abstract

In recent years computer networks have evolved by leaps and bounds, progressing based on factors such as increasing complexity, size, security, virtualization, and use in data centers. Due to this sudden expansion, technologies developed for managing and monitoring network devices have become increasingly unadapted to the current context of computer networks. Faced with this mismatch, the industry is looking for alternatives that allow automation and functional monitoring of networks. Nokia's SR Linux operating system is an open system dedicated to data centers networks that use modern management interfaces such as gNMI or JSON-RPC.

This master's dissertation describes the development of an SNMP agent for the SR Linux system. The agent aims to send SNMP Traps when an element of the system that is being monitored change state according to certain conditions. This agent uses recent technologies such as gNMI, for the internal subscription of the elements to be monitored, and Yang for easy integration into the SR Linux system. The implementation of the agent was written in python and specific libraries were used, as is the case of *pygnmi*. The implementation of the agent was tested in a virtual environment, using different scenarios, which allowed us to evaluate its correct operation.

This dissertation was supported by Nokia and Instituto de Telecomunicações.

Keywords

Computer Networking; Telemetry; SNMP; gNMI; YANG; SR Linux;

Resumo

Nos últimos anos, as redes de computadores têm evoluído a passos largos, progredindo com base em fatores como o aumento da complexidade, dimensão, segurança, virtualização e utilização em centros de dados. Devido a esta expansão repentina, as tecnologias desenvolvidas, para gestão e monitorização dos dispositivos de rede, tornaram-se cada vez mais desadaptadas do contexto atual das redes de computadores. Face a este desajustamento, a indústria procura alternativas que permitam a automatização e monitorização funcional das redes. O sistema operativo SR Linux da Nokia é um sistema aberto vocacionado para redes de centros de dados, que usa interfaces de gestão modernas tais como gNMI ou JSON-RPC.

Esta Dissertação de Mestrado descreve o desenvolvimento de um agente SNMP para o sistema SR Linux. O agente tem como objetivo o envio de SNMP traps quando um elemento do sistema que esteja a ser monitorizado mudar de estado segundo certas condições. Este agente usa tecnologias recentes como gNMI, para subscrição interna dos elementos que se pretende monitorizar, e YANG para uma fácil integração no sistema SR Linux. A implementação do agente foi escrita em Python e utilizaram-se bibliotecas específicas, como é o caso do *pygnmi*. A implementação do agente foi testada em ambiente virtual, recorrendo a diferentes cenários, o que nos permitiu avaliar o seu correto funcionamento.

Esta dissertação foi apoiada pela Nokia e pelo Instituto de Telecomunicações.

Palavras Chave

Gestão de Redes de Computadores; Telemetria; SNMP; gNMI; YANG; SR Linux;

Conteúdo

Lista de Figuras

Lista de Tabelas

Blocos de Código

Acrónimos

API	*Aplication Programming Interface*
ASN.1	*Abstract Syntax Notation One*
BER	*Basic Encoding Rules*
CLI	*Command Line Interface*
CRUD	*Create, Read, Update, Delete*
FCAPS	*Fault, Configuration, Accounting, Performance* e *Security*
gNMI	*gRPC Network Management Interface*
gRPC	*Google Protocol RPC*
HTTP	*Hypertext Transfer Protocol*
IAB	*Internet Architecture Board*
IDB	*Impart Database*
IDL	*Interface Description Language*
IETF	*Internet Engineering Task Force*
ISO	*Internation Organization for Standardization*
ITIL	*Information Technology Infrastructure Library*
ITU-T	*International Telecommunications Union*
JSON	*JavaScript Object Notation*
MIB	*Management Information Base*
MTU	*Maximum Transfer Unit*
NDK	*NetOps Development Kit*
NETCONF	*Network Configuration Protocol*
NMS	*Network Management System*

OID *Object Identifier*

OSI *Open Systems Interconnection*

RESTCONF *REST Configuration*

RPC *Remote Procedure Calls*

RTT round trip time

SDK *Software Development Kit*

SLA *Service Level Agreement*

SMI *Structure of Management Information*

SNMP *Simple Network Management Protocol*

SOAP *Simple Object Access Protocol*

SSH *Secure Shell*

TCP *Transmission Control Protocol*

UDP *User Datagram Protocol*

USM *User-Based Security Model*

VRF *Virtual Routing and Forwarding*

XML *Extensible Markup Language*

YANG *Yet Another Next Generation*

1

Introdução

Conteúdo

1.1 Motivação e Enquadramento

Nos últimos anos, os centros de dados de larga escala têm vindo a tornar-se o epicentro da evolução das redes de computadores, tanto em dimensão como complexidade entre os diferentes componentes existentes neste tipo de infraestruturas, permitindo, por exemplo, o rápido *deployment* e escalabilidade de aplicações baseadas na nuvem, a virtualização de sistemas e o armazenamento de dados. Dada esta evolução, os sistemas tradicionais de gestão e monitorização dos dispositivos nos centros de dados têm vindo a tornar-se desadaptados desta realidade. É comum nos centros de dados haver uma grande heterogeneidade de dispositivos, podendo estes estar ligados entre si através de conexões virtuais ou físicas.

Deste modo, é crucial que todos os sistemas e dispositivos existentes nos centros de dados sejam monitorizados em tempo real, lançando alertas para os técnicos e administradores destes sistemas, quer haja alguma falha momentânea, quer prevendo e/ou antecipando essas mesmas falhas, lançando avisos, para serem tomadas as medidas adequadas, de modo a mitigar essas falhas.

O sistema Service Router Linux, mais conhecido como SR Linux, é a mais recente solução desenvolvida pela empresa de telecomunicações Nokia para os routers e switchs dos centros de dados. Este sistema é baseado no *kernel* do Linux, o que permite melhorar a fiabilidade, a portabilidade e a criação de aplicações baseadas em Linux. As aplicações executadas no SR Linux são denominadas agentes.

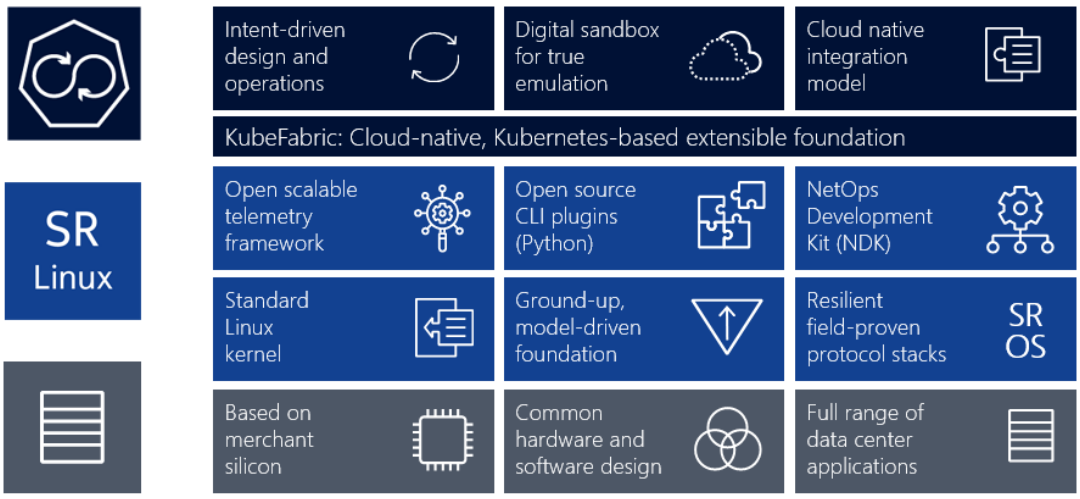

Figura 1.1: Solução *Nokia Data Center Fabric Solution* [1]

Este novo sistema operativo foi desenhado para fazer parte da solução *Nokia Data Center Fabric Solution* da Nokia. Observamos a arquitetura desta solução na Figura 1.1. Esta solução, que tem como base a utilização do sistema operativo SR Linux, visa apresentar uma solução escalável, um aumento da automação entre os diferentes componentes, um mecanismo de DevOps, simplificando o desenvolvimento e integração de aplicações nos vários componentes da solução, uma gestão de trabalho entre os componentes de uma forma simples e eficaz, bem como melhorar a monitorização de cada componente da solução.

O sistema SR Linux é baseado no modelo de gestão *model-driven CLI*, isto é, os comandos do CLI seguem e operam segundo os modelos de dados escritos em YANG [15]. A utilização de modelos de dados YANG, na forma de observar o sistema, aumenta a consistência entre as diferentes tecnologias de gestão e monitorização das redes, nomeadamente, gNMI/gRPC e JSON-RPC. Estas tecnologias permitem a gestão e a recolha de dados de uma forma eficaz utilizando novos mecanismos ao nível da camada de transporte e conteúdo, face a tecnologias mais antigas, como é o exemplo do SNMP.

1.2 Objetivos

Assim, pretende-se com esta dissertação:

- Estudar as tecnologias mais recentes na área da gestão e monitorização de redes, incluindo o conceito de *Network Telemetry*, o protocolo gNMI/gRPC, a linguagem de modelação de dados YANG e o sistema operativo SR Linux.
- Desenvolver um agente SNMP para o SR Linux.

1.3 Contribuições

O primeiro contributo desta dissertação foi ilustrar que é possível adicionar novas funcionalidades a um equipamento de rede de nova geração de uma forma simples, através de uma plataforma aberta para o desenvolvimento de aplicações. Assim, um utilizador que necessite de uma funcionalidade específica, pode ele mesmo desenvolvê-la, sem estar dependente do fabricante do equipamento.

O segundo contributo foi o desenvolvimento de um agente SNMP para o SR Linux. Apesar de o SR Linux ter protocolos de gestão de nova geração (gNMI, JSON-RPC), um cliente pode usar SNMP

nos seus sistemas de gestão e há necessidade do SR Linux suportar esta tecnologia, que não existe nativamente no equipamento e, por isso, o desenvolvimento desta funcionalidade é necessária. O código-fonte do agente desenvolvido está disponível no GitHub *GascPT/snmp_agent* [16].

1.4 Organização do Documento

Este documento de dissertação está organizado em capítulos estruturados da seguinte forma:

Capítulo 2 - Tecnologias de Gestão e Monitorização de Redes

Neste capítulo é feito o enquadramento teórico. É detalhado o conceito de gestão e monitorização das redes apresentando diversas tecnologias e mecanismos existentes nesta área.

Capítulo 3 - Telemetria

Neste capítulo é feito o enquadramento do conceito *Network Telemetry*. Um conceito relativamente recente que vem mudar a perspetiva sobre como as informações de telemetria da rede devem ser processadas.

Capítulo 4 - Requisitos e Implementação

Neste capítulo são descritos os requisitos do problema e apresentada a solução idealizada para o problema.

Capítulo 5 - Testes e Resultados

Neste capítulo é feita a avaliação sobre o trabalho desenvolvido, apresentando cenários de simulação que permitiram a verificação do correto funcionamento do sistema.

Capítulo 6 - Conclusão e Trabalho Futuro

Neste capítulo são apresentadas as conclusões finais sobre o trabalho realizado.

2

Tecnologias de Gestão e Monitorização de Redes

Conteúdo

Este capítulo tem como principal objetivo apresentar as várias tecnologias de gestão e monitorização das redes. Pretende-se comparar estas tecnologias em relação às capacidades e limitações de cada uma, com especial foco nas capacidades de monitorização de cada uma das tecnologias.

Em primeiro lugar, serão descritos alguns conceitos básicos na área de gestão e monitorização das redes, apresentando dois modelos de gestão FCAPS e ITIL e será dada uma breve comparação entre os mesmos.

De seguida, serão apresentadas diversas tecnologias de gestão e monitorização das redes, desde as tecnologias mais antigas como o SNMP, Syslog e o CLI até às tecnologias mais recentes como NETCONF, RESTCONF, gNMI/gRPC e JSON-RPC. Para cada uma destas tecnologias, será apresentada a arquitetura, o funcionamento e a evolução temporal. No final do capítulo, apresenta-se uma análise comparativa destas tecnologias.

2.1 Conceitos básicos

Atualmente, a monitorização das redes e de sistemas é um ponto fulcral para o normal funcionamento de qualquer empresa ou instituição. O responsável pela administração da rede deve ter acesso, em tempo real, ao estado dos sistemas existentes na rede, desde os sistemas considerados críticos até aos sistemas considerados não críticos. Por exemplo, em servidores, o administrador deve ter acesso ao espaço em disco, à utilização do CPU e da memória, à quantidade de tráfego na rede e a largura de banda utilizada em cada conexão, entre outros.

Com este intuito, foram desenvolvidas diversas aplicações que possibilitam a monitorização automática das redes e sistemas. A principal função destas aplicações é a recolha de informações úteis provenientes de diversas partes da rede. Como os dispositivos de rede podem estar dispersos numa grande área geográfica, estes podem não ter uma ligação direta ao nó que executa este tipo de aplicações e, por isso, foram desenvolvidas tecnologias de monitorização para permitir o acesso destas aplicações de monitorização aos dispositivos de rede [17].

Com o aumento da complexidade e expansão das redes, as aplicações de monitorização apresentam-se cada vez mais como uma peça fundamental para o bom funcionamento das redes, daí a importância de perceber os objetivos que têm de ser alcançados, para uma monitorização eficiente e prática das redes.

Existem três principais objetivos no que concerne à monitorização das redes, segundo *William Stallings* [5].

- ***Performance Monitoring***.
- ***Fault Monitoring***.
- ***Account Monitoring***.

Estes três objetivos fazem parte dos cinco objetivos definidos para a gestão das redes, propostos pelo grupo de trabalho *Open Systems Interconnection* (OSI). Os dois objetivos que não fazem parte deste grupo são *configuration management* e *security management* [17].

O primeiro objetivo da monitorização, *performance monitoring*, lida com a medição e monitorização do desempenho da rede. Neste objetivo, existem três pontos importantes associados:

- A recolha de informação de desempenho para planeamento e identificação de problemas na expansão da própria rede.
- O tempo de monitorização deve ser grande o suficiente para permitir criar um modelo comportamental da rede fidedigno, permitindo retirar medições de desempenho próximas aos valores reais.
- O que medir numa rede? Existem diversas informações que podem ser medidas na rede, mas têm de ser identificadas quais são as informações com mais-valia avaliar.

O segundo objetivo, *fault monitoring*, lida com problemas operacionais dos dispositivos, nas diferentes camadas que o compõem e, dependendo das características da rede, definir o que é uma falha ou não. Por exemplo, erros de transmissão de dados, só será considerado erro de transmissão, quando for ultrapassado um certo valor no modelo comportamental da rede.

O terceiro objetivo, *account monitoring*, trata de como os utilizadores podem utilizar a rede. Por exemplo, mantém-se um registo de quais os dispositivos da rede o utilizador utiliza e com que frequência utiliza esses dispositivos [17].

2.1.1 Modelos de Gestão das Redes

De forma a obter uma gestão e monitorização eficiente das redes, é preciso existir normas e padrões que possam ser utilizados pelos administradores e por todos os utilizadores envolvidos no uso dessas tecnologias. Assim, surgiram os modelos de gestão de redes. Estes modelos explicitam a organização das diferentes tarefas e funcionalidades, permitindo executar com mais facilidade a monitorização dos sistemas e operações que compõem a infraestrutura da rede. De seguida, serão apresentados os dois modelos mais utilizados no âmbito da gestão das redes: FCAPS e ITIL.

2.1.1.1 FCAPS

O *Internation Organization for Standardization* (ISO) em conjunto com o *International Telecommunications Union* (ITU-T), definiu um modelo de gestão de redes conhecido por FCAPS. O nome do modelo de gestão é o acrónimo de *Fault, Configuration, Accounting, Performance* e *Security* (FCAPS) [18]. Este modelo é utilizado por sistemas de gestão para gerir infraestruturas escaláveis e minimizar as falhas na rede.

Este modelo define cinco áreas para a gestão de redes de modo que cada área possui um âmbito e um conjunto de componentes bem definido. As suas áreas são as seguintes:

- **Gestão de Falhas** (*Fault Management*);
- **Gestão de Configuração** (*Configuration Management*);
- **Gestão de Contas** (*Accounting Management*);
- **Gestão de Desempenho** (*Performance Management*);
- **Gestão de Segurança** (*Security Management*);

A Figura 2.1 ilustra cada área da norma FCAPS e como cada área interage com as outras. É possível verificar que a área *Security Management* interage com todas as áreas, dada a importância da segurança nas infraestruturas nos dias de hoje, enquanto a área *Configuration Management*, que é a base das outras áreas pois fornece informações importantes sobre cada componente, não interage diretamente com a área de *Accounting Management*.

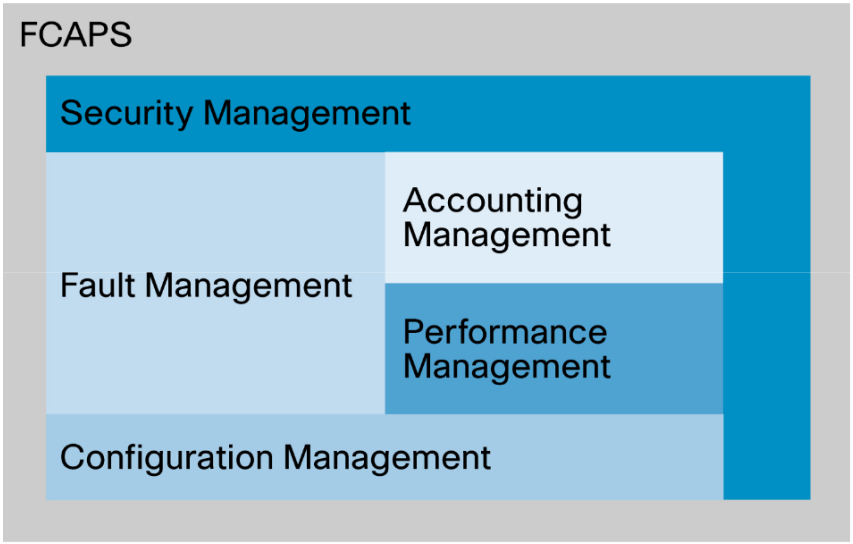

Figura 2.1: Funções do modelo de gestão FCAPS [2]

A seguir é detalhado cada área do modelo FCAPS [19] [20] :

Gestão de Falhas

É uma das principais áreas funcionais deste modelo de gestão, destaca-se pela importância na gestão de redes de comunicação. O objetivo desta área é manter uma rede operacional e sem falhas. Para atingir este objetivo a categoria está dividida em três fases distintas: monitorização, diagnóstico e envio de alertas.

A fase de monitorização consiste na deteção de erros, mediante a ocorrência de alertas produzidos pelos dispositivos de rede. Para isso, deverá existir um meio de visualização de informação relativa a todos os alertas enviados pelos dispositivos de rede. Por exemplo, através de representações gráficas do estado dos componentes, ao longo do tempo, que permitam uma fácil interpretação da informação.

Por norma, quando acontece uma falha são acionados diversos alarmes. Por isso, é imperativo a existência de um diagnóstico dos erros gerados, para determinar a causa, de modo a auxiliar a tomada de decisão por parte do sistema ou do gestor de rede.

Gestão de Configuração

É uma área que abrange um vasto conjunto de funções em que se destaca a função de recolha, monitorização e a alteração de informações de configuração dos dispositivos de rede.

Ao nível dos serviços, esta área permite modificar um serviço de rede, modificando os seus parâmetros-/configurações, ou desativando o mesmo.

Nos sistemas de gestão de redes mais sofisticados, é possível que, a partir da informação de cada dispositivo de rede, se chegue, de forma automática, ter uma visão topológica dessa infraestrutura.

Gestão de Contabilização

A função de contabilização apresenta uma maior importância em redes comerciais, em que é preciso contabilizar a utilização dos serviços da rede pelos utilizadores.

No entanto, em redes não comerciais, esta função também pode ser considerada importante, pois através dela é possível identificar padrões de utilização dos serviços, o que poderá auxiliar o administrador de rede na tomada de decisões.

Apesar de não haver custos associados à utilização de recursos pelos utilizadores em redes não comerciais, os equipamentos e meios de comunicação têm um custo, nomeadamente na aquisição de equipamentos e na manutenção dos mesmos. Assim, é importante ter noção sobre os recursos utilizados na rede, por parte dos utilizadores, para ser possível justificar os custos de manutenção e justificar eventual escalabilidade da rede.

Gestão de Desempenho

A Gestão de Desempenho tem como principal objetivo a recolha e análise do estado dos elementos de rede, ao nível físico e lógico, com intuito de monitorizar e corrigir o comportamento e eficácia da rede. A análise dos elementos e serviços da rede permite identificar se a qualidade do serviço é a que se encontra acordada no *Service Level Agreement* (SLA).

Sistemas mais elaborados, através da análise de desempenho, podem estabelecer modelos comportamentais da rede e assim diagnosticar problemas na rede e prever o seu desempenho futuro.

Gestão de Segurança

É uma área que abrange uma série de funções de forma a garantir mecanismos de segurança numa infraestrutura de rede. Algumas destas funções encontram-se enumeradas na Tabela 2.1.

Tabela 2.1: Funções de Segurança no modelo FCAPS

Acesso seletivo a recursos	Registo em logs	Privacidade dos dados
Controlo de acesso por utilizadores	Alarmes de Segurança	*Firewalls*

Estas funções permitem garantir que apenas utilizadores autorizados consigam aceder aos equipamentos e efetuar alterações a configurações da rede. Para além das medidas internas de segurança, devem ser tomadas também medidas que permitam garantir segurança em acessos externos. Um exemplo deste tipo de medidas é a limitação de acesso a portos e endereços IP acessíveis de fora da rede.

2.1.1.2 ITIL

A framework *Information Technology Infrastructure Library* (ITIL) é um conjunto de boas práticas para tecnologias e serviços IT. A framework ITIL teve a sua génese no ano de 1980 em Inglaterra, na *Central Computer and Telecommunication Agency*. A versão atual é a ITILv4 que foi publicada em 2019, mas por motivos de simplificação no contexto desta dissertação será detalhada a versão ITILv3.

A versão ITILv3 surgiu em abril de 2011 e conforme se pode observar pela Figura 2.2, compreende 5 áreas distintas.

Figura 2.2: Estrutura ITIL [3]

Antes de detalhar cada área, é importante definir o conceito de serviço, segundo Jack Probst: “Um serviço é a forma de fornecer valor acrescentado aos clientes, facilitando ou ajudando-os a alcançar os resultados desejados, sem que tenham de assumir os custos e os riscos específicos” [21]. Por outras palavras, o cliente de um sistema IT deve concentrar-se no seu negócio e não tem que se preocupar com questões técnicas associadas à execução do serviço. Um exemplo de serviço é o serviço de email, quando o utilizador envia ou recebe um email, este não se preocupa qual o servidor ou software que utiliza para enviar ou receber a mensagem.

Service Strategy

É a área que contempla um conjunto de diretrizes que ajudam as organizações a definir os serviços que pretendem executar/desenvolver, segundo a estratégia da própria organização, e de modo que os objetivos do negócio sejam atingidos.

Service Design

É uma das áreas mais importantes do ITIL. Nesta fase, são desenhados os serviços e processos definidos na área de Service Strategy. No desenho de cada serviço são definidos os métodos e estratégias a utilizar, para atingir os objetivos pretendidos. Define-se ainda o nível de serviço que é executado, a capacidade e a disponibilidade do serviço, entre outros parâmetros.

Service Transition

É a fase que põe os serviços em produção. São oferecidos ao utilizador final, todos os serviços, tendo a garantia que qualquer problema que ocorra, em qualquer dos serviços disponibilizados, esteja previsto a sua resolução, minimizando o impacto da falha do serviço, de forma a gerar valor para a organização. Fazem parte, desta fase, os seguintes processos: a gestão de configuração, a gestão de mudança, validação do serviço, entre outros.

Service Operation

É a fase que contempla um conjunto de diretrizes de forma a fornecer instruções para uma entrega eficiente de cada serviço ao utilizador. Nesta fase, há um conjunto de processos de forma a garantir a fluidez da operação: gestão de problemas, gestão de incidentes, gestão de eventos, gestão de acessos, entre outros. Como se pode observar pela Figura 2.3.

Continual Service Improvement

Esta é a última fase do ciclo ITIL, conforme ilustrado na Figura 2.2. É nesta fase que se verifica se os objetivos definidos inicialmente, na fase de Service Strategy, foram atingidos com sucesso, e o que se pode fazer para melhorar cada serviço. Faz parte desta fase um processo que permite identificar e melhorar cada serviço denominado “The Seven-Step Improvement Process” [22].

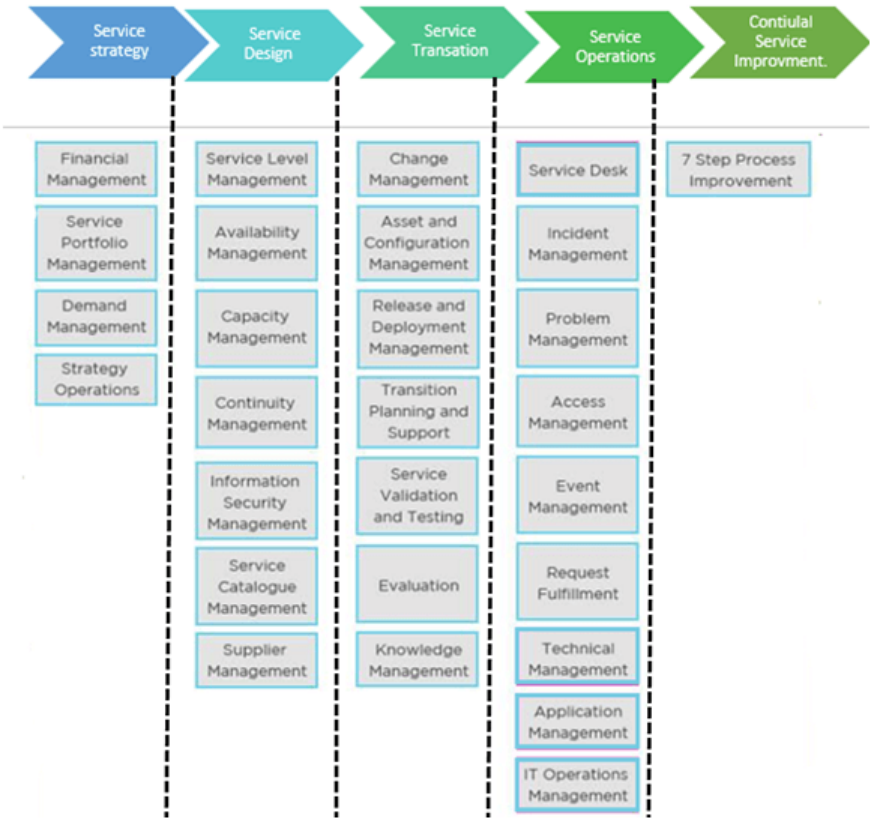

Figura 2.3: Processos ITILv3 [4]

2.1.1.3 FCAPS vs ITIL

A norma FCAPS e norma ITIL, apesar de terem o mesmo objetivo, a gestão de redes e serviços, diferem principalmente na forma como se focam para atingir este objetivo. A norma ITIL foca-se na vertente de negócio enquanto a norma FCAPS foca-se na tecnologia. Na Tabela 2.2, podemos observar o mapeamento das categorias FCAPS, já enumeradas, com alguns dos processos ITIL da Figura 2.3.

Tabela 2.2: Mapeamento de funções ITIL e FCAPS [11]

	Funções FCAPS				
Funções ITIL	Falhas	Configuração	Contabilização	Desempenho	Segurança
Incident Management	✓				
Problem Management	✓				
Configuration Management		✓			
Change Management		✓			
Release Management		✓			
Accounting			✓		
Capacity Management				✓	
Availability Management				✓	
Service Reporting				✓	
Information Security Management					✓

2.2 SNMP

O protocolo *Simple Network Management Protocol* (SNMP) é um protocolo da camada da aplicação, desenvolvido pelo *Internet Architecture Board* (IAB) e definido no RFC 1157 [23]. Este protocolo foi desenvolvido no final dos anos 80 e permite a gestão e configuração de dispositivos de rede a partir de um *Network Management System* (NMS), que designaremos por *manager*.

2.2.1 Arquitectura

O protocolo utiliza uma arquitetura *manager-agent* [24] onde os vários *agents* comunicam com o *manager*, conforme ilustrado na Figura 2.4.

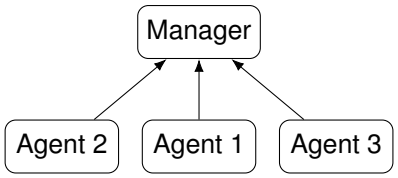

Figura 2.4: Arquitectura *manager-agent*

O protocolo SNMP assenta sobre três componentes básicos [25]:

- *Agents* — Consiste num *software* que está instalado nos dispositivos de rede que se pretende gerir. Uma das responsabilidades do *agent* é comunicar informações que constem na *Management Information Base* (MIB) ao *manager* quando este o solicitar. Pode, quando devidamente configurado, enviar mensagens do tipo *SNMP Trap* quando algum elemento da MIB mudar de estado.

- *Managers* — Enviam mensagens para os *agents* para recolher informações sobre as suas MIBs.

- MIB — É uma base de dados composta por dois formatos de objetos: escalar ou tabelas, que existe em cada *agent* e que representa os modelos de dados do dispositivo.

2.2.2 Funcionamento

O protocolo SNMP utiliza por omissão o protocolo *User Datagram Protocol* (UDP) na troca das mensagens, apesar de ser possível utilizar o protocolo *Transmission Control Protocol* (TCP). A opção pela utilização de UDP deve-se ao facto da comunicação por UDP ser mais rápida que por TCP, beneficiando o modo de atuação do protocolo SNMP, ao permitir o rápido envio de mensagens entre o *manager* e o *agent* [26].

Este protocolo utiliza a notação *Structure of Management Information* (SMI) [27] para definir as suas MIB [28]. A representação dos dados existe de duas formas: escalar ou por tabelas. A forma escalar representa uma instância de um objeto, enquanto, a forma por tabelas permite representar e correlacionar múltiplas instâncias de objetos, que são agrupados nas tabelas MIB. As duas formas são identificadas por *Object Identifier* (OID). Estes OID têm uma estrutura que representa um caminho ao longo de uma árvore de dados normalizada denominada *ISO Object Identifier Tree*. Na Figura 2.5, está ilustrada esta árvore de dados e o caminho até à MIB-II. A MIB-II é a segunda versão de uma MIB para gestão de protocolos de redes baseados em TCP/IP [29].

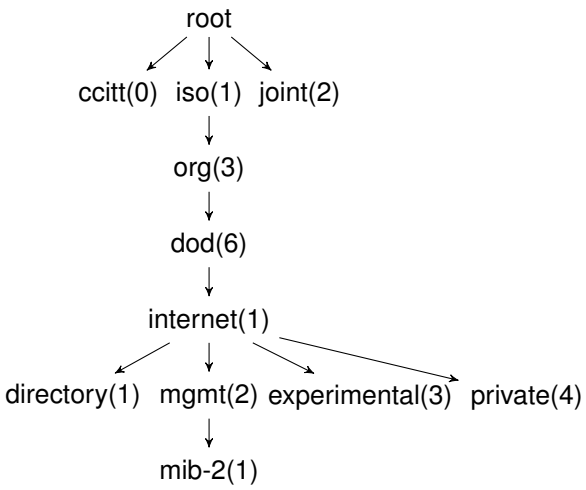

Figura 2.5: *ISO Object Identifier Tree*

Qualquer equipamento de rede que suporte SNMP tem de implementar também a subárvore MIB-II [29] [30], que tem como caminho **iso.org.dod.internet.mgmt.mib-2**, ou numericamente **1.3.6.1.2.1**, como se pode observar pela Figura 2.5. Os objetos dentro da MIB-II estão divididos nos dez grupos apresentados na Tabela 2.3.

Tabela 2.3: Subárvore MIB-II

Grupo	Descrição
system (1)	Informações sobre o sistema
interfaces (2)	Interfaces de rede
at (3)	Tradução de endereços
ip (4)	Protocolo IP
icmp (5)	Protocolo ICMP
tcp (6)	Protocolo TCP
udp (7)	Protocolo UDP
egp (8)	Protocolo EGP
transmission (9)	Meios de transmissão
SNMP (11)	Protocolo SNMP

O *manager*, quando quer obter informações sobre algum elemento de um *agent*, envia o OID desse elemento numa mensagem para o *agent* com uma operação do tipo *Get*. Por exemplo, para se obter o nome do dispositivo de rede (*sysName*), o *manager* envia o OID **1.3.6.1.2.1.1.5**, que corresponde a **mib-2.system.sysName**, para conseguir-se obter a informação pretendida. Apesar deste protocolo ser amplamente aceite e utilizado na comunidade, a utilização de uma sequência de números extensa, para identificar os objetos, por vezes, dificulta a compreensão humana do protocolo e, por consequência, a sua utilização em larga escala.

O protocolo SNMP apresenta dois modos de atuação, como ilustrado na Figura 2.6.

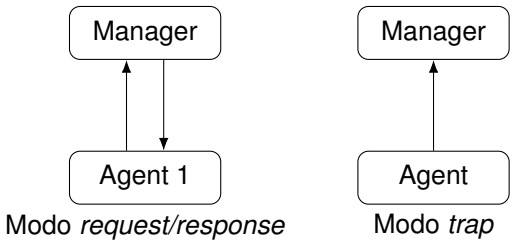

Figura 2.6: Modos de atuação do protocolo SNMP

No modo *request/response*, o *manager* inicia a comunicação com o *agent*, enviando um *request* para o *agent*, enquanto que, no modo *trap*, é o *agent* que inicia a comunicação com o *manager*.

2.2.2.1 Mensagem SNMP

Cada mensagem SNMP trocada entre o *manager* e o *agent* contém três campos: versão SNMP, a *community string* e um *Protocol Data Unit* (PDU). Existem diferentes tipos de PDUs para cada operação que o protocolo SNMP pode realizar. Estes PDUs estão divididos em 5 classes distintas, conforme consta na Tabela 2.4 [31].

Tabela 2.4: Classes de PDUs.

Classe	PDUs
Read Class	GetRequest-PDU, GetNextRequest-PDU e GetBulkRequest-PDU
Write Class	SetRequest-PDU
Response Class	Response-PDU e Report-PDU
Notification Class	TRAPv2-PDU e InformRequest-PDU
Internal Class	Report-PDU

Na Figura 2.7, observam-se os diferentes formatos de um pacote SNMP nas diferentes operações que suporta.

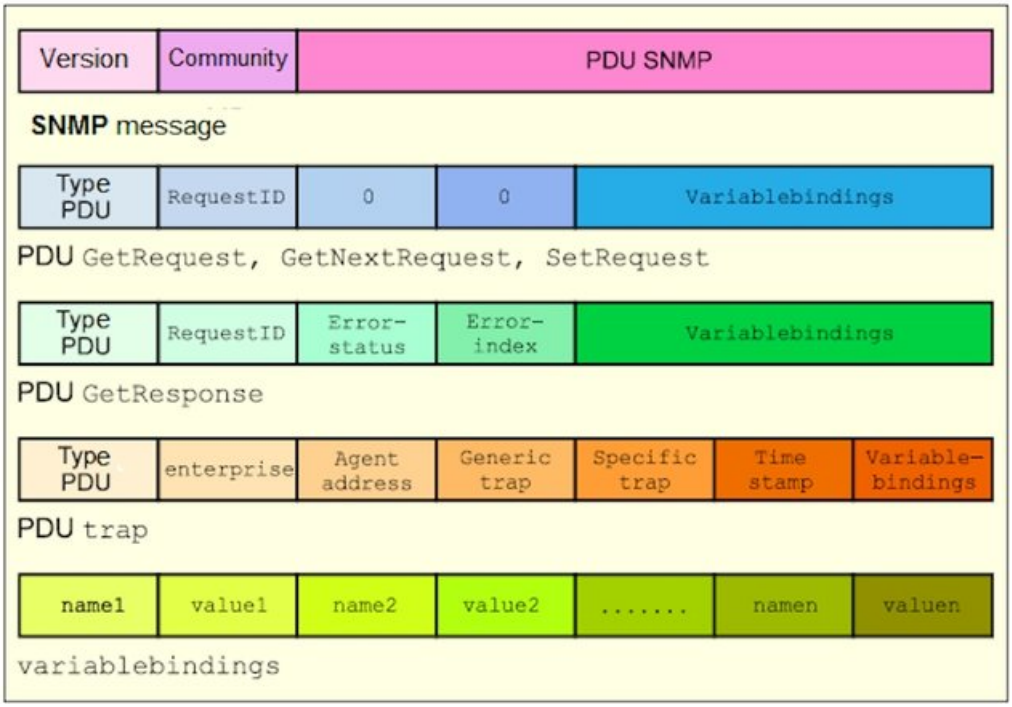

Figura 2.7: Formatos das mensagens SNMP [5]

Os principais campos de uma mensagem PDU SNMP são:

- ***Type PDU*** — Este campo pode assumir 5 valores distintos de modo a distinguir o tipo de operação que a mensagem transporta.

Tabela 2.5: Campo *Type PDU*

Tipo PDU	Operação
0	Get-Request
1	Get-Next-Request
2	Get-Response
3	Set-Request
4	Trap

- ***Request ID*** — Identificador do tipo inteiro que permite identificar uma mensagem SNMP na troca de mensagens do tipo *request/response*. A resposta a um pedido SNMP contém o mesmo *Request ID* que o pedido de modo a identificar o par pedido/resposta.

- ***Error Status*** — Campo que identifica o tipo de erro ocorrido. No caso de ser uma mensagem do tipo *Request* o campo assume sempre o valor "0", apenas as respostas é que transportam o erro.

- ***Error Index*** — Caso exista um erro, este campo guarda o ponteiro para o objeto que causou o erro.
- ***Variable Bindings*** — Lista com os objetos pedidos, cada objeto possui dois campos, o OID e o Value.

Os campos descritos pertencem às operações *Get, GetNext, Set e Response*, as Traps PDU apresentam campos diferentes:

- ***Enterprise*** — identifica o sistema que gerou o trap, é o valor do *SysObjectID* do sistema.
- ***Agent Address*** — contém o endereço IP do agente que gerou a trap.
- ***Generic Trap*** — Identificador que identifica o tipo de Trap entre seis tipos pré-definidos, conforme consta na Tabela 2.6.

Tabela 2.6: Tipos de Trap Genérica

Valor	Trap
0	coldStart
1	warmStart
2	linkDown
3	linkUp
4	authenticationFailure
5	egpNeighborLoss
6	enterpriseSpecific

- ***Specific Trap*** — Campo que permite identificar uma Trap específica, fora dos tipos pré-definidos.
- ***Timestamp*** — Tempo decorrido desde a última inicialização do agente.

2.2.3 Evolução do SNMP

A primeira versão do protocolo, o SNMPv1 [23], apenas especificava as operações *GetRequest*, *GetNextRequest*, *SetRequest* e a operação *Trap*. Esta versão apresentava vários problemas, quer ao nível da segurança, quer ao nível operacional. Por exemplo, a operação *GetRequest* apenas podia ler uma entrada da MIB em cada pedido. Assim, para ler várias entradas teriam de ser trocadas diversas mensagens *GetRequest* o que consumia uma grande largura de banda na rede. Da necessidade de colmatar este problema, foi desenvolvida a versão SNMPv2 [32] do protocolo que introduziu duas novas operações *GetBulkRequest*, que permitia ler várias entradas de uma MIB, e *Inform* para transferência síncrona de notificações de eventos.

Apesar deste desenvolvimento ao nível operacional, ao nível da segurança não houve o mesmo desenvolvimento. Estas duas versões utilizavam a noção de *communities* para estabelecer uma relação

de confiança entre o *manager* e os *agents*. Um *agent* era configurado com três tipos diferentes de nomes de *communities*: i) *read-only*; ii) *read-write*; iii) *trap*. Nestas versões, o nome das *communities* funcionava essencialmente como uma *password* para acesso aos dados guardados nas MIBs, isto é, se um *manager* enviasse uma mensagem *GetRequest* com o nome da *community* igual a "X" e os dados no *agent* tivessem guardados na *community* com o nome "Y", o *agent* iria descartar esta mensagem, pois o *manager* não enviou o nome da *community* certa. A maioria dos fabricantes utilizava, por omissão, o nome *public* para a comunidade *read-only* e *private* para a comunidade *read-write*.

Como o nome das *communities* sugere, cada *community* desempenha funções diferentes. A *community read-only* permite apenas que sejam feitas leituras de estado/valor (*Read Class*) sobre um OID, a *community read-write* permite que se faça a leitura e modificação de dados no *agent* (*Write Class*) e a *community Trap* permite que o *manager* receba as *traps* geradas pelo *agent*.

Estas *community strings* eram transmitidas entre o *manager* e o *agent* em formato de texto e podiam ser facilmente visualizadas na rede através da inspeção do tráfego. Devido a esta fraca segurança, o SNMP era muitas vezes limitado às suas funções de monitorização, dado que os administradores de rede não arriscavam habilitar operações *SetRequest* na rede, pois a *community string* passava em claro na rede e podia potenciar um possível vetor de ataque aos elementos da rede.

Posteriormente, foi desenvolvida a versão SNMPv3. Esta versão adicionou suporte à autenticação dos *agents* e dos *managers*, cifra nas comunicações e um melhor controlo de acessos às MIBs. O grupo de trabalho, responsável pelo desenvolvimento desta versão do protocolo, definiu um modelo de segurança baseado em nomes de utilizadores, denominado *User-Based Security Model* (USM) [33].

O SNMP foi, durante vários anos, o principal protocolo de monitorização de redes [34] mas, atualmente, devido a vários fatores operacionais do protocolo, já não responde às necessidades atuais das redes. Isto deve-se principalmente a fatores como: a falta de flexibilidade do SMI, para descrever dados complexos; falta de mecanismos de *lock*, para gestão distribuída de vários *managers*, de modo que fosse garantida a consistência dos dados quando, um *agent* fosse gerido em simultâneo por mais que um *manager*; a não diferenciação entre os dados de estado e de configuração; a complexidade de leitura/escrita de configurações completas num *agent* ou a utilização de UDP na camada de transporte, sendo um protocolo que não garante entrega. Ainda existiu uma proposta para utilizar TCP como protocolo de transporte [35] mas esta nunca passou da fase experimental. A utilização de UDP também apresentava outra grande desvantagem, onde as mensagens UDP são limitadas pelo *Maximum Transfer Unit* (MTU), o que causa problemas quando se usa a operação *GetBulkRequest* sobre uma grande quantidade de dados, pois esse limite de dados, por mensagem, era facilmente ultrapassado [36]. O protocolo SNMP, apesar das várias versões de desenvolvimento e tentativas de melhoramento do protocolo, apresentou sempre diversas lacunas e a sua utilização sempre ficou reduzida às funções de monitorização [34] [36] [24].

2.3 Syslog

O protocolo Syslog é amplamente utilizado, para registo de eventos, desde a década de 80. Surgiu, pela primeira vez, num serviço de email [12] mas foi rapidamente adaptado a outras aplicações e, assim, tornou-se um standard de *logging* de aplicações em sistemas Unix. Originalmente, não houve nenhuma normalização do protocolo e, por isso, existem diversas implementações do protocolo incompatíveis entre si. Só em março de 2009, com o RFC 5424 [37], é que o protocolo foi normalizado.

2.3.1 Arquitetura e Funcionamento

O protocolo utiliza uma arquitetura de três camadas, em que cada camada é responsável por uma série de funcionalidades distintas, conforme se pode observar na Figura 2.8 [12].

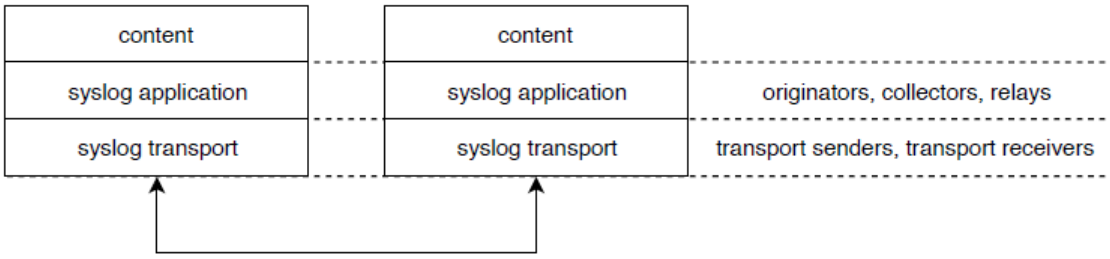

Figura 2.8: Camadas Syslog

- **Content** — Camada responsável pela informação a ser transmitida,
- **Syslog aplication** — Nesta camada, é processada a mensagem Syslog, desde a sua síntese, processamento e armazenamento.
  - *originator* — Entidade que cria a informação a ser transmitida.
  - *collector* — Entidade que coleta a informação.
  - *relay* — Entidade que encaminha mensagens syslog entre o *originator* e o *collector*.
- **Syslog transport** — Camada que insere a mensagem syslog num protocolo de transporte.

Diversos cenários podem ser implementados, utilizando um ou mais *relays* para encaminhar as mensagens Syslog entre o *originator* e o *collector*, mas também é possível que o *originator* e o *collector* residam na mesma máquina física.

2.3.1.1 Mensagem Syslog

As mensagens Syslog trocadas entre o *originator* e o *collector*, são normalmente transportadas sobre UDP. A escolha por UDP deve-se ao facto deste protocolo consumir menos largura de banda, o que

em redes altamente congestionadas é uma vantagem, apesar do protocolo não garantir a entrega.

Uma mensagem Syslog consiste nos seguintes campos [12]:

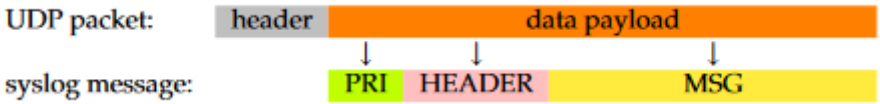

Figura 2.9: Mensagem Syslog [6]

Como se pode observar pela Figura 2.9, a mensagem Syslog está dividida em três campos:

- **PRI** — Campo que permite verificar a prioridade da mensagem, é composto por três a cinco caracteres sendo que o primeiro é o carácter "<" e o último carácter é ">", entre estes símbolos está o valor de prioridade que é calculado utilizando a seguinte fórmula:

$$\mathrm{P}_{riority} = 8 \cdot \mathrm{F}_{acility} + \mathrm{S}_{everity} \tag{2.1}$$

Os valores de FACILITY e SEVERITY estão descritos na Tabela 2.7 e na Tabela 2.8 respetivamente.

Tabela 2.7: Códigos Facility [12]

Código	Facility
0	kernel messages
1	user-level messages
2	mail system
3	system daemons
4	security/authorization messages (note 1)
5	messages generated internally by syslogd
6	line printer subsystem
7	network news subsystem
8	UUCP subsystem
9	clock daemon (note 2)
10	security/authorization messages (note 1)
11	FTP daemon
12	NTP subsystem
13	log audit (note 1)
14	log alert (note 1)
15	clock daemon (note 2)
16	local use 0 (local0)
17	local use 1 (local1)
18	local use 2 (local2)
19	local use 3 (local3)
20	local use 4 (local4)
21	local use 5 (local5)
22	local use 6 (local6)
23	local use 7 (local7)

Tabela 2.8: Códigos Severity [12]

Código	Severity
0	Emergency: system is unusable
1	Alert: action must be taken immediately
2	Critical: critical conditions
3	Error: error conditions
4	Warning: warning conditions
5	Notice: normal but significant condition
6	Informational: informational messages
7	Debug: debug-level messages

- **HEADER** — Este campo contém um *timestamp* e um identificador do *originator*, que pode ser o *hostname* ou o endereço IP.
- **MSG** — Contém a informação útil do evento ocorrido, é composto por dois campos *TAG* e *CONTENT*.

2.3.2 Evolução do Syslog

O protocolo Syslog ganhou uma grande adesão como protocolo para registo de eventos. Apesar desta adesão o protocolo apresenta grandes desvantagens em contextos distribuídos onde o *originator* tem de enviar mensagens para o *colletor* sobre a rede. Algumas destas desvantagens são [6]:

- **Transporte**

  Devido ao facto do protocolo utilizar, por omissão, UDP para enviar as mensagens sobre a rede, este não consegue garantir a entrega das mensagens, além de não garantir a entrega também não garante a correta ordem de entrega das mensagens. Em outras palavras, as mensagens podem chegar ao *collector* com a ordem diferente da enviada pelo *originator*, o que pode causar uma má interpretação dos eventos ocorridos.

- **Segurança**

  O recetor da mensagem Syslog não consegue garantir que a mensagem tenha sido enviada por um *originator* legítimo. Assim, um atacante pode fazer *spoofing* de mensagens Syslog para um *collector*, fazendo-se passar por um *originator* legítimo, gerando uma grande quantidade de eventos no sistema, podendo distrair os administradores do sistema de outros ataques a decorrer.

- **Uniformização**

  Cada aplicação e sistema são escritos independentemente uns dos outros, como o RFC original do protocolo é vago na forma como utilizar o Syslog com uma estrutura de dados para descrever os eventos, também a forma como cada aplicação e sistema operativo reflete os seus eventos é independente, e por isso, existe uma grande falta de uniformização dos dados.

2.4 CLI

A maioria dos equipamento de rede apresenta uma forma *ad-hoc* de ser configurado. Esta forma *ad-hoc* denomina-se por *Command Line Interface* (CLI). O CLI é, normalmente, visualizado através de um terminal e permite que os utilizadores executem comandos de modo a visualizar informações do equipamento ou configurar o equipamento com diferentes parâmetros. Nos primórdios dos equipamentos de rede, cada dispositivo era configurado fisicamente através do CLI, mas rapidamente se percebeu que esta forma de gerir e configurar os equipamentos de rede não era escalável para topologias de média ou grande dimensão.

Posteriormente, surgiram tecnologias que permitiam o acesso remoto ao CLI de um equipamento, como o *Telnet* ou *Secure Shell* (SSH). Através do uso destas tecnologias, era possível visualizar os dados dos equipamentos remotamente, o que não era possível até à altura. Com isto, surgiu uma técnica denominada *Screen Scraping* [38]. Esta técnica consiste na execução de um *script* para leitura do *output* de um programa/terminal para utilização num outro sítio, por exemplo, para recolha e armazenamento dos dados.

Esta técnica apresentava algumas falhas no seu funcionamento. Principalmente, dependia bastante do formato do *output*. Por exemplo, normalmente um *script* acede ao CLI de um equipamento, executa um comando *show* e recolhe a informação que lhe interessa, através de padrões de texto pré-definidos para o *output* daquele comando, caso esse padrão de texto mude já não é possível recolher a informação pretendida e ter-se-ia de mudar o *script* para o padrão voltar a corresponder com a informação pretendida.

Hoje em dia, os equipamentos de rede oferecem *Aplication Programming Interface* (API) que, por sua vez, oferecem a mesma capacidade de visualização e retorno de dados que o CLI mas de forma programática e estruturada e que não dependem do formato do *output* do CLI. Assim, a necessidade desta técnica tem decrescido, ao longo dos últimos anos.

2.5 NETCONF

O protocolo *Network Configuration Protocol* (NETCONF) é um protocolo de configuração dos dispositivos de rede, publicado pelo *IETF Working Group*, no final do ano de 2006, e definido no RFC 4741 [39]. Este protocolo apresenta um conjunto de operações que permitem obter e configurar dados nos dispositivos de rede através de *Remote Procedure Calls* (RPC).

2.5.1 Arquitetura

O protocolo utiliza uma arquitetura cliente/servidor, onde o servidor é o dispositivo de rede e o cliente o sistema que opera o dispositivo de rede através das operações disponibilizadas pelo NETCONF. Na Figura 2.10, podemos observar a divisão do protocolo NETCONF em camadas.

Conteúdo	Dados de Configuração	Dados de Notificação
Operações	<get><get-config><edit-config>...	
Mensagens	<rpc>/ <rpc-reply>	<notification>
Transporte	SSH/TLS/...	

Figura 2.10: Camadas do protocolo NETCONF [7]

- Conteúdo: Na camada conteúdo, podemos ter dois tipos de dados, a ser transportados, dados de configuração ou dados de notificação. O primeiro tipo de dados diz respeito aos dados que o cliente pretende obter ou gerir. O segundo tipo de dados é gerado exclusivamente pelo servidor (dispositivo de rede), com informação de algum evento ocorrido, semelhante à operação Trap do protocolo SNMP. A comunicação parte exclusivamente do dispositivo de rede para o cliente.

- Operações: O NETCONF disponibiliza uma série de operações que fazem com que seja um protocolo mais completo, para gestão e monitorização de dispositivos de rede. Entre as operações que dispõe vale a pena referir as seguintes:

  - <*get*> — Obtém dados de configuração ou dados de estado do servidor.
  - <*edit-config*> — Permite editar ou criar, caso não exista, uma configuração.
  - <*lock*> — O cliente bloqueia o acesso a dados de configuração no servidor.

- Mensagens: — As mensagens em NETCONF são estruturadas no formato *Extensible Markup Language* (XML). Se os dados forem provenientes de operações de configuração são encapsuladas com a *tag* <rpc>, se os dados forem provenientes de uma notificação a mensagem é encapsulada com a *tag* <notification>.

- Transporte: No desenvolvimento do protocolo, pretendeu-se que este fosse *"connection-oriented"*, isto é, houvesse garantia de sessão e garantia na ordem de entrega das mensagens. Além deste requisito também a segurança foi tida em conta, para garantir autenticação, integridade e confidencialidade da ligação [39]. Assim, estão previstas quatro opções para utilização na camada transporte: o SSH [40], o BEEP [41], o SOAP [42] e o TLS [43].

2.5.2 Funcionamento

Quanto ao modo de operação, quando um cliente pretende configurar um dispositivo de rede, o cliente conecta-se ao dispositivo de rede por SSH ou por *Simple Object Access Protocol* (SOAP). De seguida, os intervenientes autenticam-se mutuamente e trocam informação sobre o que se pode configurar (*capabilities*). A partir deste momento, quando um cliente quer realizar uma operação no dispositivo de rede irá estruturar uma mensagem em XML com a operação e informação que pretenda realizar e/ou obter.

O protocolo NETCONF introduz o conceito de *datastores* ou repositório de configurações. Estes repositórios são conjuntos de configurações que permitem um dispositivo mudar de estado. Foram definidos três tipos de configurações. A Figura 2.11 apresenta as interações entre as diferentes *datastores*:

- *running* — Configuração ativa no dispositivo.
- *startup* — Configurações que por omissão são carregadas ao iniciar o dispositivo.
- *candidate* — Configuração que permite mudar a configuração de um dispositivo sem alterar o estado atual do mesmo.

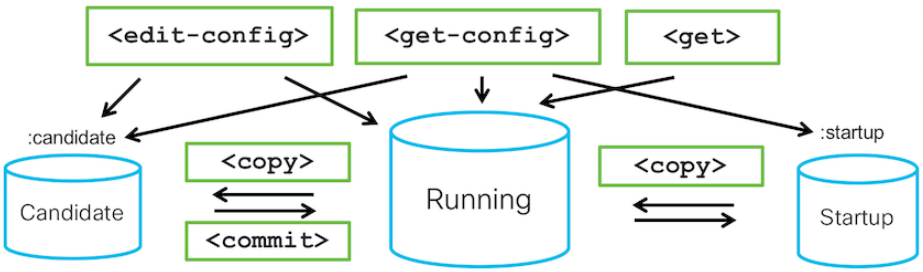

Figura 2.11: Interações entre *datastores* e operações NETCONF [8]

2.5.3 Evolução do NETCONF

O protocolo NETCONF surgiu da necessidade de resolver os problemas que o protocolo SNMP apresentava na gestão de dispositivos de rede. A segunda revisão do protocolo, o RFC 6241 [7], introduziu a utilização da linguagem de modelação de dados YANG, que será explicada na secção 2.9, como recomendação para modelar os dados e operações do protocolo, cobrindo assim a camada Conteúdo e Operações, representadas na Figura 2.10.

2.6 RESTCONF

Semelhante ao NETCONF, o protocolo *REST Configuration* (RESTCONF) é um protocolo de configuração dos dispositivos de rede definido no RFC 8040 [44]. O protocolo foi construído sobre o protocolo *Hypertext Transfer Protocol* (HTTP) e utiliza métodos HTTP para providenciar operações *Create, Read, Update, Delete* (CRUD) sobre o dispositivo de rede. A camada conteúdo do protocolo utiliza modelos de dados YANG, que será explicado na secção 2.9, como forma de ver os dados no dispositivo de rede.

2.6.1 Arquitetura e Funcionamento

Como já foi referido, o protocolo RESTCONF utiliza mensagens HTTP na comunicação entre os intervenientes da solução, semelhante ao NETCONF, cliente e servidor, onde o servidor é o dispositivo de rede que se planeia gerir. O protocolo utiliza a mesma noção de *datastores* que o NETCONF para gerir as configurações de um dispositivo. Nas Figuras 2.12 e 2.13 observamos um pedido e uma resposta em RESTCONF.

```
GET /restconf/data/ HTTP/1.1
Host: example.com
Accept: application/yang.data+json
```

Figura 2.12: Exemplo de um pedido em RESTCONF [9]

```
HTTP/1.1 200 OK
Date: Tue, 14 Apr 2015 17:01:00 GMT
Server: restconf-impl-server
Content-Type: application/yang.data+json

{
  "data": {
    "system" : {
      // contents taken out for display purposes
    }
  }
}
```

Figura 2.13: Exemplo de uma resposta em RESTCONF [9]

2.7 gNMI/gRPC

A *framework gRPC Network Management Interface* (gNMI) é uma especificação do protocolo *Google Protocol RPC* (gRPC) desenvolvido pela Google. Esta *framework* visa oferecer aos administradores das redes uma ferramenta capaz de configurar e monitorizar os dispositivos de rede remotamente [45]. Devido ao facto desta *framework* ser ainda bastante recente, não existem RFCs sobre este tópico, e o que existe ainda está em versões *draft*, à data deste relatório [46].

O protocolo gRPC, por si só, é um protocolo de elevada confiabilidade e performance. Utiliza *Protocol Buffers* e HTTP/2 para atingir estes objetivos, o que faz com que este protocolo seja bastante utilizado por diversas empresas que operam em larga escala, como, por exemplo, Netflix, Cisco, Juniper, etc. [47].

Protocol Buffers é uma linguagem de codificação de estruturas de dados, diferente da linguagem YANG. Esta linguagem define formatos de codificação e armazenamento dos dados, *Google Protocol Buffers self-describing* e *Google Protocol Buffers compact*, ou *GPB compact* e *GPB self-describing*. A linguagem permite definir como é que as operações entre um cliente e um servidor devem ser executadas e com que parâmetros.

2.7.1 Arquitetura

A *framework* gNMI utiliza a linguagem gRPC para definir o serviço que disponibiliza, bem como as mensagens que devem ser trocadas. A especificação define quatro métodos: *Get*, *Set*, *Capabilities* e *Subscribe*. Estes métodos são utilizados no cliente, para invocar funções com certos parâmetros e retornos predefinidos no serviço.

No Bloco de Código 2.1, podemos observar a especificação do serviço no ficheiro *.proto* retirado do repositório Github do gNMI [48].

```
1 service gNMI {
2     rpc Capabilities(CapabilityRequest) returns (CapabilityResponse);
3     rpc Get(GetRequest) returns (GetResponse);
4     rpc Set(SetRequest) returns (SetResponse);
5     rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
6 }
```

Bloco de Código 2.1: Serviço *gNMI*

No serviço gNMI, que podemos observar no Bloco de Código 2.1, observamos a especificação de quatro métodos do tipo RPC, cada um deles com o nome do método, o tipo de parâmetros de entrada e os tipos de retorno. Neste serviço temos dois tipos de métodos: métodos unários, que são os métodos *Capabilities, Get, Set*, e um método com *streaming* bidirecional, o método *Subscribe*.

Estes métodos utilizam a notação *gNMI Path Convention* para identificar os elementos do servidor. Esta notação percorre a árvore de dados do servidor para compor uma *string* capaz de identificar um elemento. Por exemplo, se no servidor temos o *container* A que contém o *container* B, que por sua vez contém a *leaf* C, a *string* resultante da identificação do elemento C é A/B/C. Apesar de se poder usar modelos de dados escritos em YANG para descrever a árvore de dados do servidor, sendo que os autores referem que qualquer *Interface Description Language* (IDL) pode ser usada.

2.7.2 Funcionamento

Os métodos do gNMI têm as seguintes funções:

- *Capabilities* — Permite ao cliente entender que modelos de dados o servidor suporta, bem como *encodings* e a versão gNMI que o servidor utiliza.

- *Get* — Obtém o estado/valor de um determinado elemento do sistema identificado por um *path*, segundo a *gNMI Path Convention*.

- *Set* — Modifica o estado/valor de um determinado elemento do sistema identificado por um *path*, segundo a *gNMI Path Convention*. Esta função permite três modos de operação diferentes: *Update*, *Replace* e *Delete*.

- *Subscribe* — Permite subscrever um elemento do sistema para que quando este modifique o seu estado/valor, o cliente receba uma notificação sobre a alteração do estado. Esta função permite três modos de operação diferentes: *Stream*, *Poll* e *Once*.

A *framework* utiliza HTTP/2 na camada de transporte. A utilização de HTTP/2 permite que sejam feitos múltiplos pedidos HTTP sobre a mesma conexão TCP, o que aumenta a eficiência e utilização dos recursos da rede [49]. Na Figura 2.14, temos uma comparação entre o protocolo HTTP 1.1, à esquerda,

e o protocolo HTTP/2, à direita. Como se pode observar, o facto de o protocolo HTTP/2 fazer os pedidos em paralelo diminui o tempo de espera pelos mesmos, isto se não houver congestionamentos nas ligações.

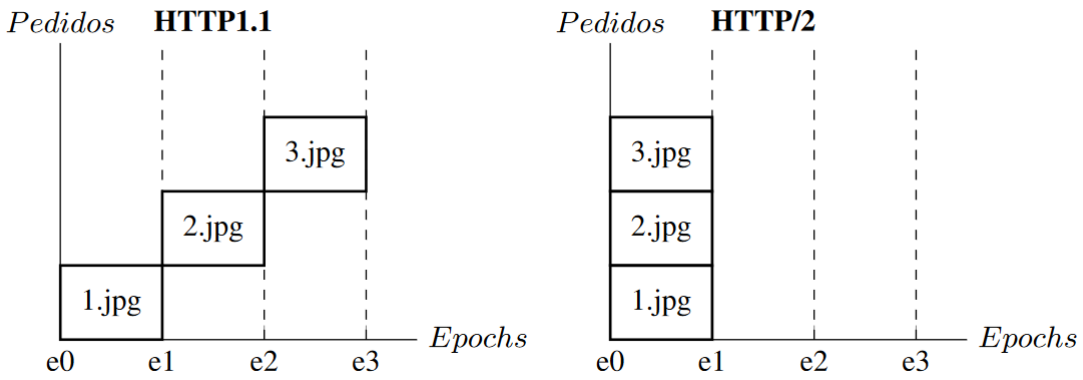

Figura 2.14: Comparação entre os protocolos HTTP 1.1 e HTTP/2

Através da conjugação das duas tecnologias apresentadas, a *framework* consegue oferecer aos administradores de rede uma ferramenta capaz de configurar e monitorizar os dispositivos de rede, através de um NMS. Deste modo, os dispositivos de rede, apenas têm de implementar um servidor gNMI, sem precisar de implementar outras ferramentas para diferentes funções, por exemplo, NETCONF, para configuração e SNMP, para monitorização.

O método *Subscribe* implementa o mecanismo de *streaming* bidirecional. Este método é utilizado para monitorização do dispositivo de rede, com base em subscrições (*streaming telemetry*), em que o NMS envia um pedido para o dispositivo de rede, para subscrever as alterações de estado de um certo *path* e o canal de comunicação TCP pode ou não, manter-se aberto, dependendo do modo de operação. O método disponibiliza três modos de operação diferentes:

- ONCE — Neste modo de operação, o cliente gNMI envia um *request* para o servidor gNMI, este responde com os dados pretendidos e, de seguida, fecha o canal de comunicação entre os dois. É um modo de operação similar à função *Get*, mas apresenta uma melhor performance, quando se tenta obter um objeto de grande dimensão do servidor, porque o método *Subscribe* pode responder ao cliente em *streaming*, isto é, o servidor vai enviando os dados à medida que são processados e o cliente não os recebe todos de uma vez, como acontece na função *Get*.

- POLL — Neste modo de operação, o cliente gNMI envia em *streaming* pedidos para o servidor, isto é, o cliente pretende adquirir informação sobre um determinado *path*, envia esses pedidos e,

de seguida, o servidor responde com a informação pretendida. Ao contrário do modo de operação ONCE, o canal mantém-se aberto, podendo o cliente fazer mais do que um pedido. Este modo de operação também tem uma *flag updates_only*, que muda o comportamento da operação. Quando esta *flag* está a *false*, a resposta do servidor vem com toda a informação solicitada em cada *path*, quando a *flag* está a *true*, a resposta do servidor vem só com os *paths* que alteraram de estado em relação ao último pedido do cliente.

- STREAM — Neste modo de operação, o cliente gNMI envia um *request* para o servidor gNMI com as subscrições que pretende configurar e/ou monitorizar, isto é, quais *paths* pretende receber informação. Esta informação pode ser obtida baseada em eventos (*ON_CHANGE*) ou em intervalos de tempo (*SAMPLE*). Diferente do modo de operação POLL, o cliente apenas envia um pedido de subscrição no início da conexão TCP, não podendo enviar mais pedidos naquela sessão TCP.

2.8 JSON-RPC

O JSON-RPC é um protocolo de *remote procedure call* (RPC). Existem duas versões principais do protocolo. A primeira versão (1.0) surgiu em 2005 [50] e a segunda versão (2.0) surgiu em 2010 [51].

2.8.1 Arquitetura e Funcionamento

Este protocolo define apenas dois objetos: *Request Object* e *Response Object*.

O *Request Object* é o objeto enviado pelo cliente ao servidor. Este objeto tem os seguintes componentes:

- *jsonrpc* : Identificador com a versão JSON-RPC da mensagem.
- *method* : Nome do método a ser invocado no servidor.
- *params* : Estrutura com os parâmetros para serem usados na invocação do método.
- *id* : Identificador estabelecido pelo cliente para identificar a mensagem.

O *Response Object* é o objeto enviado pelo servidor ao cliente. Este objeto tem os seguintes componentes:

- *jsonrpc* : Identificador com a versão JSON-RPC da mensagem.

- *result* : Resultado da invocação do método no servidor, caso a execução tenha ocorrido com sucesso.
- *error* : Estrutura que explicita o erro ocorrido, caso ocorra.
- *id* : Identificador estabelecido pelo cliente para identificar a mensagem.

É de salientar que *result* e o *error* são componentes mutuamente exclusivos, isto é, só aparece um, no *Request Object*.

Como o nome do protocolo indica, o protocolo utiliza a estrutura de dados *JavaScript Object Notation* (JSON) na codificação destes objetos. Na camada transporte, o protocolo tanto pode usar HTTP como também pode usar *sockets* TCP/IP.
Este protocolo pode ser utilizado desde a configuração de dispositivos de redes, como no caso do *SR Linux* da *Nokia*, até *blockchain* em que existe uma especificação do protocolo para Ethereum [52].

Na implementação de JSON-RPC, em *SR Linux*, estão definidos quatro métodos diferentes [53].

Tabela 2.9: Métodos suportados no *SR Linux*

Método	Descrição
get	Método para adquirir dados de estado e de configuração
set	Método para configurar algum elemento do sistema
validate	Método para validar se o sistema irá aceitar as configurações pretendidas
cli	Método para correr comandos do *CLI*

Podem utilizar-se diversas ferramentas para construir o *Request Object*, como, por exemplo, *cURL* ou *wget*. No Bloco de Código 2.2 observa-se um pedido JSON-RPC feito com a ferramenta *cURL* a um dispositivo *SR Linux*, e no Bloco de Código 2.3 observa-se a resposta a este pedido.

```
1 curl -d '{"jsonrpc": "2.0","id": 0,"method": "get",
2 "params": {"commands": [{"path": "/system/name","datastore": "state"}]}}'
3 http://admin:admin@172.20.20.2:80/jsonrpc
```

Bloco de Código 2.2: Exemplo de JSON-RPC *Request Object*

```
1 {"result": [{"host-name": "srl"}], "id": 0, "jsonrpc": "2.0"}
```

Bloco de Código 2.3: Exemplo de JSON-RPC *Response Object*

2.9 YANG

A linguagem de modelação de dados *Yet Another Next Generation* (YANG) foi originalmente desenvolvida para modelar a camada conteúdo do protocolo NETCONF [39]. O conteúdo a ser modelado e descrito na versão 1.0 da linguagem RFC 6020 [54] eram os dados de configuração e dados de estado de um dispositivo de rede, as *remote procedure calls* que podiam ser invocadas e as notificações.

Posteriormente, a linguagem foi adaptada para poder ser transversal a todos os protocolos de gestão das redes (NETCONF, RESTCONF, gNMI, etc) na camada conteúdo. Assim, surgiu a versão 1.1 da linguagem, descrita no RFC 7950 [14].

A linguagem YANG endereça alguns problemas levantados durante a conferência *"Overview of the 2002 IAB Network Management Workshop"*. Estes problemas estão expostos e descritos no RFC 3535 [55]. Abaixo, apresentam-se algumas características da linguagem YANG, conforme consta no RFC 6244 [56]:

- *Ease of use*: A linguagem YANG foi desenhada para ser facilmente compreensível e de rápida legibilidade.
- *Configuration and State data*: A linguagem permite diferenciar o que são dados de configuração e dados de estado (velocidade observada, contadores, etc.).
- *Task Oriented*: Um modelo YANG permite definir operações (*remote procedure calls*). Num modelo cliente-servidor, é possível que o cliente invoque uma operação no servidor, pela forma como esta está descrita no modelo YANG, associado àquele serviço.
- *Implementation Difficulty*: A flexibilidade dos módulos em YANG permite que estes sejam estendidos e adicionadas funcionalidades específicas, caso necessário.
- *Simple data modeling language*: A linguagem YANG pode ser utilizada em outros contextos, como, *on-box APIs*, ou, por exemplo, um CLI pode ser implementado em YANG (*Model Driven CLI*), o que permite simplificar a estrutura do CLI.

A transversalidade da camada conteúdo, entre os diversos protocolos de gestão das redes, possibilita uma maior automatização e flexibilidade na configuração dos dispositivos de rede, pois estes protocolos não têm de se adaptar aos dados em si, apenas têm de tratar as suas especificações nas restantes camadas. A Figura 2.15 apresenta o modelo em camadas dos protocolos de gestão de redes.

Protocolo	NETCONF	RESTCONF	gNMI/gRPC
Conteúdo	YANG Data Model		
Operações	<get> <get-config>	GET, POST, PATCH	getRequest, setRequest
Serialização	XML	JSON —— XML	Protobuf —— JSON
Transporte	SSH	HTTP —— HTTPS	HTTP/2

Figura 2.15: Camadas dos protocolos de gestão de redes

Como ilustrado na Figura 2.15, a camada conteúdo dos protocolos de gestão das redes NETCONF, RESTCONF e gNMI é transversal aos protocolos. Apenas as outras camadas é que são específicas de cada protocolo. Isto quer dizer que a forma como os dados são apresentados pelo dispositivo de rede é igual perante os diferentes protocolos, estes apenas têm de seguir a estrutura de dados que o dispositivo de rede utiliza.

Assim, a linguagem define um contrato formal entre um cliente e um servidor, uma API [57], sobre os dados que podem ser configurados ou não, bem como os dados que podem ser monitorizados.

2.9.1 Linguagem YANG

A linguagem *YANG* apresenta um conjunto de *keywords* que permitem a construção dos modelos de dados, a sua representação numa estrutura em árvore e reutilização. Na Tabela 2.10 estão descritas estas *keywords* bem como uma breve descrição das suas funcionalidades.

Tabela 2.10: Conjunto de *keywords* da linguagem

Keyword	Descrição
Augment	Estende uma hierarquia já existente
Container	Agrega um conjunto de nós
Extension	Adiciona novos nós ao modelo de dados
Feature	Indica partes opcionais do modelo de dados
Grouping	Agrupa um conjunto de nós num conjunto reutilizável
Key	Define o identificador de uma lista
Leaf	Representa um nó folha na hierarquia
Leaf-List	Define uma lista de nós folha
List	Representa uma lista de nós
Notification	Define os dados de uma notificação
Type	Define o conteúdo de uma *leaf*
Typedef	Mecanismo para criação de novos tipos
Uses	Permite usar os conjuntos definidos com a *keyword grouping*

Os modelos de dados escritos em YANG são divididos em módulos (*modules*) e, dentro de cada módulo, podemos ter diferentes contentores (*containers*) que agrupam os dados. Dentro de cada *container*, podemos ter outros *containers*, *leafs*, que corresponde à unidade mais básica do modelo de dados, *lists*, *leaf-list*, entre outros. As *leafs* que, tal como o nome indica, são o último nó da estrutura em árvore, têm de ter sempre um tipo associado. As *lists* agrupam um conjunto de *leafs*, por forma a criar um objeto com vários nós diferentes. As *lists* são identificadas por uma *key*.

Os módulos em YANG podem ser divididos em três secções: i) o cabeçalho do módulo, onde consta a versão da linguagem em que o módulo está escrito, o *namespace* do módulo, o prefixo, bem como meta-informação de contacto e uma breve descrição do módulo; ii) outra secção do módulo é a secção de revisão, que indica quando o módulo foi revisto; iii) a última secção de um módulo é a secção mais importante e é onde aparece a definição do modelo de dados.

No Bloco de Código 2.4 podemos observar um exemplo de um módulo YANG, retirado do RFC 7950 [14]. Neste exemplo é possível identificar as diferentes secções, a secção de cabeçalho entre as linhas 2 e 7, a secção de revisão nas linhas 9 a 11 e a secção de definição do módulo entre as linhas 13 e 34. Já no Bloco de Código 2.5 podemos observar a estrutura em árvore deste módulo.

```
1   module example-system {
2       yang-version 1.1;
3       namespace "urn:example:system";
4       prefix "sys";
5       organization "Example Inc.";
6       contact "joe@example.com"
7       description "The module for entities implementing the Example system.";
8
9       revision 2007-06-09 {
10          description "Initial revision.";
11      }
12
13      container system {
14          leaf host-name {
15              type string;
16              description "Hostname for this system.";
17          }
18          leaf-list domain-search {
```

```
19                  type string;
20                  description "List of domain names to search.";
21              }
22              container login {
23                  leaf message {
24                      type string;
25                      description "Message given at start of login session.";
26                  }
27                  list user {
28                      key "name";
29                      leaf name { type string; }
30                      leaf full-name { type string; }
31                      leaf class { type string; }
32                  }
33              }
34          }
```

Bloco de Código 2.4: Exemplo de um módulo YANG retirado do RFC 7950 [14]

Como podemos observar pelo Bloco de Código 2.4, é definido o módulo *example-system*. Este módulo agrupa, no *container system*, duas *leafs* e um segundo *container* denominado *login*. Este segundo *container* tem uma *leaf message* e uma *list* que agrupa um conjunto de *leafs* e, como chave da lista, a *leaf name*.

Como já foi dito, a linguagem YANG modela os dados numa estrutura em árvore. Por forma a visualizar-se esta estrutura, utilizou-se a ferramenta *open-source pyang* sobre o módulo descrito no Bloco de Código 2.4 para se obter o Bloco de Código 2.5.

```
1  module: example-system
2    +--rw system
3       +--rw host-name?       string
4       +--rw domain-search*   string
5       +--rw login
6          +--rw message?   string
7          +--rw user* [name]
8             +--rw name         string
9             +--rw full-name?   string
10            +--rw class?       string
```

Bloco de Código 2.5: Diagrama em árvore do módulo *YANG* retirado do RFC 7950 [14]

No exemplo do Bloco de Código 2.4 observamos vários componentes do módulo e como esses componentes se refletem na estrutura em árvore do Bloco de Código 2.5. Por exemplo, na linha 27 do Bloco de Código 2.4, observamos a declaração de um *list user*, esta lista é identificada por uma *key*, neste caso *name*. No Bloco de Código 2.5 na linha 7, temos a declaração da lista *user* com a chave *name*, e um nível abaixo, as suas *leaf's*.

Para referenciar um nó no modelo de dados YANG, a linguagem disponibiliza um mecanismo denominado YANG *Path*. Esta notação percorre a árvore de dados do modelo de dados para compor uma *string*, capaz de identificar um elemento. Por exemplo, no Bloco de Código 2.5, temos o *container system* que contém o *container* login que, por sua vez, contém a *list user*. Esta lista tem a *leaf name*, que também é a *key* da própria lista. A *string* resultante da identificação do elemento *full-name*, dentro da lista, é **system/login/user[name=xxxx]/full-name**.

Recorrendo aos módulos YANG, consegue-se descrever um modelo de dados para dados de configuração e dados de estado numa hierarquização em árvore. No caso do Bloco de Código 2.5 todos os objetos são dados de configuração, porque cada objeto tem o prefixo *rw* (*read-write*) mas, se algum objeto tivesse sido configurado com o parâmetro *config false*, apareceria *ro* (*read-only*) antes do objeto. Através dos mecanismos de *augmentation* e *deviation feature*, é possível criar modelos de dados, de uma forma modular, a partir de modelos mais genéricos.

2.9.2 Representação dos dados

A linguagem YANG apenas expressa a forma como os dados devem ser estruturados e não define nenhum formato para armazenamento dos dados. A representação dos dados modelados em YANG pode ser em XML ou em JSON.

A representação de YANG na linguagem XML é denominada *YANG Independent Notation* (YIN), em que existe uma correspondência direta entre as YANG *keywords* e *YIN elements*, o que preserva a estrutura e o conteúdo definido no módulo YANG.

A representação em JSON apesar de simples e intuitiva, o que torna este tipo de codificação muito popular em aplicações REST, apresenta algumas desvantagens na utilização com YANG. Como, por exemplo, na tradução de YANG para JSON do mecanismo de *namespace* ou a extensão de funcionalidades dos módulos baseadas nesse *namespace*. A forma encontrada, para endereçar os problemas de tradução entre YANG e JSON, está descrita no RFC 7951 (*"JSON Encoding of Data Modeled with YANG"*) [58].

2.10 Resumo

Este capítulo visou apresentar as principais tecnologias desenvolvidas na área da gestão e monitorização de redes, desde as primeiras tecnologias como SNMP, Syslog, CLI até às mais recentes como NETCONF, RESTCONF, gNMI. Explicaram-se os desafios e as opções tomadas em cada uma das tecnologias e a forma como se foram adaptando às realidades de cada época.

Através do estudo realizado sobre estas tecnologias foi possível formular a Tabela 2.11, que apresenta uma visão geral de cada tecnologia estudada e uma breve comparação entre cada uma usando parâmetros considerados pertinentes para comparação.

Tabela 2.11: Comparação entre os protocolos e tecnologias de gestão e monitorização das redes

	SNMP	**Syslog**	**CLI**	**NETCONF**	**RESTCONF**	**gNMI**	**JSON-RPC**
Normalizado	IETF	☒	☒	IETF	IETF	OpenConfig	JSON-RPC Working Group
Recursos	OIDs	☒	☒	*Paths*	*Paths*	*Paths*	*Paths*
Modelo de Dados	MIB	☒	☒	Módulos YANG	Módulos YANG	Módulos YANG	☒
Linguagem do Modelo de Dados	SMI	☒	☒	Yang	Yang	Yang	☒
Operações	SNMP	Não definidas	Não definidas	NETCONF	CRUD	gNMI	Não definidas
Codificação	ASN.1 BER	*Plain Text*	*Plain Text*	XML	JSON XML	*Protocol Buffers*	JSON
Protocolo de Transporte	UDP	UDP	SSH Telnet	SSH TCP	HTTP	HTTP/2	HTTP
Segurança	Apenas na versão SNMPv3	☒	☒	✓	✓	✓	☒

3

Telemetria

Conteúdo

3.1 Conceito

Com a evolução das redes, as tecnologias de monitorização das redes como SNMP, Syslog ou o CLI têm vindo a tornar-se cada vez mais desfasadas da realidade atual. Com esta premissa, desenvolveu-se o conceito de *Network Telemetry* ou também conhecido como *Streaming Telemetry*. Neste conceito, descrito no RFC 9232 [13], os dados de telemetria são obtidos baseando-se em subscrições de determinados elementos de um dispositivo de rede, em vez de *queries/polling* como acontece em SNMP. Na Figura 3.1, podemos observar estes dois modos de operação.

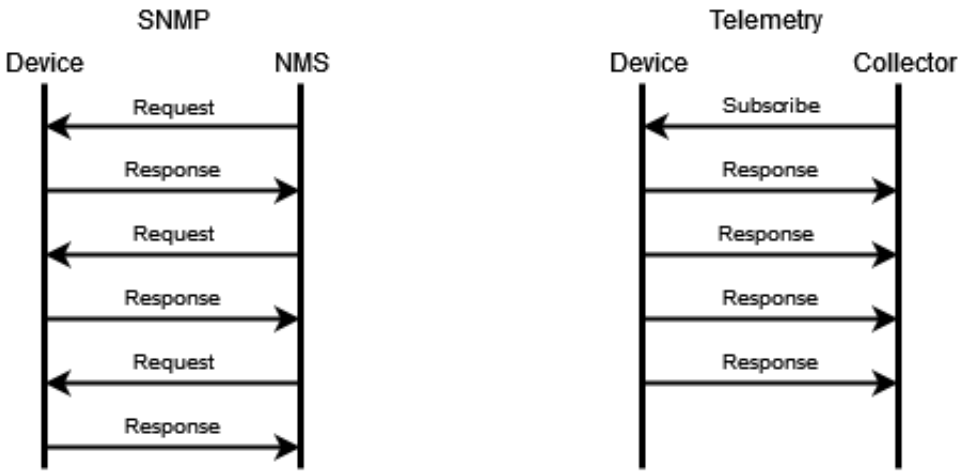

Figura 3.1: Modo de operação SNMP e *Network Telemetry*

Como observamos pela Figura 3.1, o modo de operação do SNMP é do tipo *request/response*. Neste tipo de comunicação, cada vez que o NMS pretenda obter uma informação do dispositivo de rede terá de enviar um pedido a este dispositivo de rede e este, por sua vez, irá responder com a informação solicitada. Por outro lado, o modo de operação em *Network Telemetry* é baseada em subscrições, isto é, o *collector* (dispositivo que consome os dados de telemetria) envia para o dispositivo de rede que elemento ou elementos pretenda obter informação e o dispositivo irá, dependendo do tipo de subscrição, enviar a informação para o *collector*. A principal vantagem observável entre estes dois conceitos é o tempo de comunicação dos dados. Enquanto em SNMP é preciso um round trip time (RTT) para comunicar a informação, em *Network Telemetry* os dados são enviados para o *collector* sem que este precise de os solicitar diretamente, isto é, basta que tenha uma subscrição para um dado elemento do dispositivo de rede que este irá enviar informação para o *collector*, poupando assim tempo de comunicação entre o *collector* e o dispositivo de rede.

3.2 Arquitectura

Na Figura 3.2, podemos observar a arquitetura proposta pelo grupo de trabalho *Internet Engineering Task Force* (IETF) para o conceito de *Network Telemetry*.

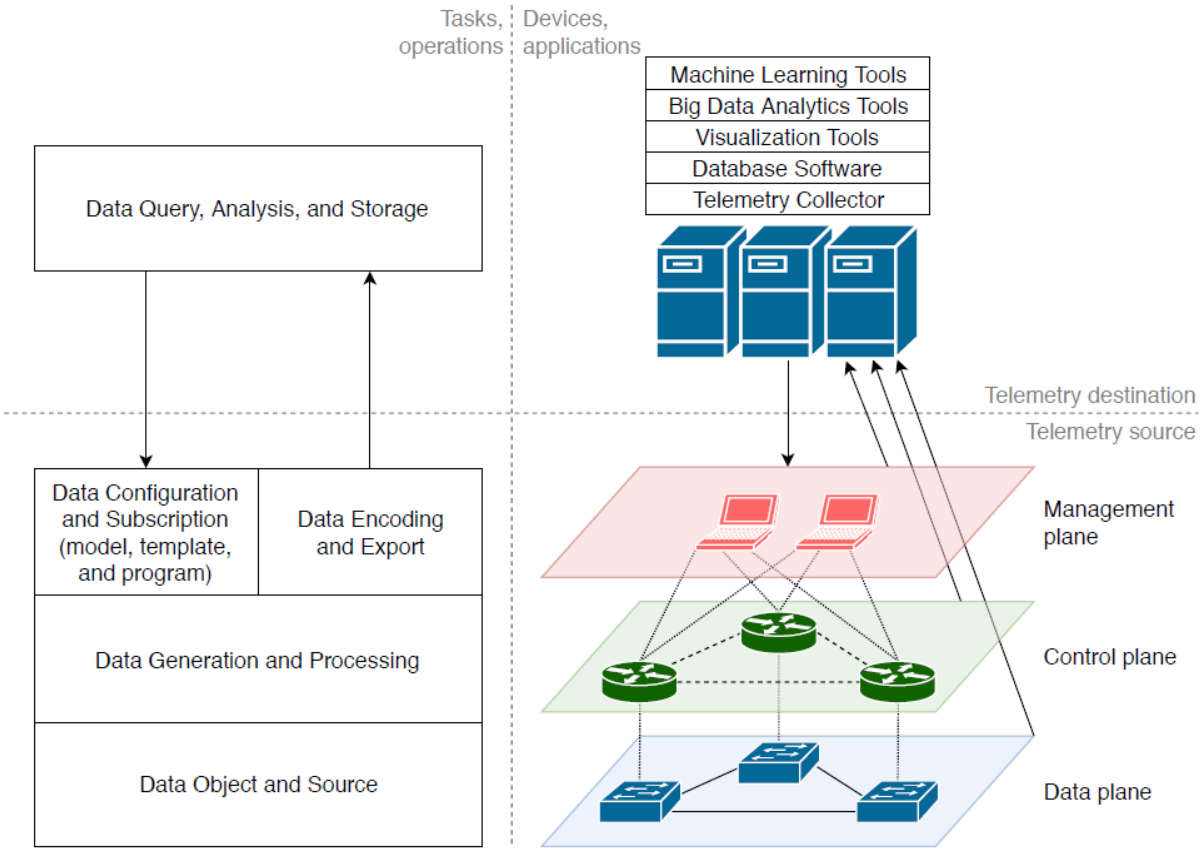

Figura 3.2: Arquitectura em *Network Telemetry* [10]

Na arquitetura proposta há dois grandes intervenientes [13]:

- *Telemetry Source*: São os dispositivos de redes que geram os dados de telemetria. É da responsabilidade dos dispositivos de rede gerar os dados, normalizando-os e exportando-os para o *collector*. Os dados de telemetria podem ser divididos em três planos distintos: i) *Data Plane*; ii) *Control Plane*; iii) *Management Plane*. Em cada um dos planos, a arquitetura divide em diferentes componentes e tarefas a realizar:
  - *Data Object and Source*: É o componente mais básico da arquitetura, é a fonte inicial dos dados, por exemplo, uma interface. Neste ponto, os dados encontram-se na sua forma mais básica (*raw data*).

  - *Data Generation and Processing*: Os dados são devidamente capturados e filtrados da fonte de origem e processados.
  - *Data Encoding and Exporting*: Nesta componente, os dados são devidamente codificados num protocolo de transporte e exportados.
  - *Data Configuration and Subscription*: Esta componente lida com as subscrições que o dispositivo de rede dispõe e redireciona os dados para os canais de transporte associados a cada subscrição.

- *Telemetry Destination*: *Collectors* ou *Consumers* são o destino dos dados de telemetria para serem consumidos. Dependendo do uso dos dados estes podem ter diversas utilidades para diferentes aplicações:
  - Aplicações de *Big Data*, *Machine Learning*, Inteligência Artificial.
  - Ferramentas de visualização gráfica dos dados.
  - Sistemas de armazenamento, para ser possível guardar os dados, eficazmente.
  - Aplicações que consumam os dados em *streaming* e façam o *parse* para outras aplicações.

Neste conceito de *Network Telemetry*, cada componente descrita é independente entre si, podendo ser tomadas diferentes opções dependendo do tipo de solução que se pretenda obter. Os próximos pontos discutem as principais opções que se tomam em cada área de intervenção.

3.3 Modelos de Dados

A linguagem de modelação de dados YANG, adotada por tecnologias como NETCONF ou RESTCONF, tornou-se um padrão neste campo, em que diversas organizações (IETF, OpenConfig, vendedores na área das redes) criam modelos *standards* para as diversas funcionalidades existentes nos dispositivos de redes. O repositório *YangModels/yang*[1] no GitHub tem uma coleção de modelos de dados publicados por estas organizações e utilizados pelos protocolos de gestão e monitorização das redes para configurar e monitorizar os dispositivos de rede. Os mesmos modelos de dados são usados pelos dispositivos de rede, para enviar os dados de telemetria. O uso de modelos de dados, como forma de comunicação e interação das funcionalidades existentes num dispositivo de rede, ajuda a que seja mais transparente a forma como estes dados circulam e são usados numa rede com diversos tipos e marcas de dispositivos de rede diferentes, o que beneficia a automatização da própria rede.

[1] https://github.com/YangModels/yang

3.4 Subscrição

Como já foi explicado, o conceito de *Network Telemetry* baseia-se na subscrição de elementos de um dispositivo de rede. Existem dois tipos de subscrição diferentes: a subscrição baseada em eventos (*on_change*) ou baseada em intervalos de tempo (*sample intervals*). Neste contexto, é importante perceber que tipo de subscrição é melhor para cada tipo de dados que queremos obter. A subscrição do tipo *on_change* é melhor para notificar o *collector* relativamente a alguma alteração de estado, por exemplo, uma interface do dispositivo de rede ter ido abaixo. O tipo de subscrição *sample intervals* envia, periodicamente, para o *collector* informação sobre um determinado elemento do dispositivo de rede, por exemplo, temperatura do CPU, que ao longo do tempo, tende a variar bastante.

3.4.1 Sessão

Existem duas formas diferentes de início de sessão de telemetria, isto é, como é efetuada a subscrição, no dispositivo de rede ou através de algum mecanismo de subscrição para o *collector*. O dispositivo de rede pode fazer *"dial-out"* para o *collector*, ou o *collector* pode efetuar *"dial-in"* para o dispositivo de rede. Na forma de *"dial-out"* temos a vantagem de não precisarmos de abrir portas no dispositivo de rede, mas, em contrapartida, tem de se configurar, em cada dispositivo de rede, a subscrição e o endereço do *collector*. Na forma de *"dial-in"*, é o *collector* que cria um canal de comunicação, em *streaming*, onde envia os elementos do dispositivo de rede que pretende obter dados. Em contrapartida, o dispositivo de rede tem de abrir uma porta de comunicação, o que pode criar um possível vetor de ataque. Esta última forma de iniciar uma sessão com o dispositivo de rede é a forma utilizada pelo protocolo de gestão *gNMI/gRPC*, explicado na secção 2.7.

3.5 Codificação dos dados

Após coleta e processamento dos dados no dispositivo de rede, estes têm de ser codificados num formato para serem exportados. Existem diferentes tipos de codificação, mas os mais usados, neste contexto, são JSON, XML, *Google Protocol Buffer "self-describing"* e *Google Protocol Buffer compact*. A melhor forma de comparar estes tipos de codificação é através do tamanho final dos dados codificados. Aqui, destaca-se o *Google Protocol Buffer compact* pois ao codificar os dados em binário leva a que o tamanho final seja bastante reduzido, comparativamente aos outros três tipos de codificação. Por outro lado, o XML, por utilizar *tags* para conter os dados e os próprios dados serem codificados em *strings*, apresenta o pior desempenho entre os quatro tipos de codificação.

3.6 Transporte

A discussão neste aspeto é sobre qual o melhor protocolo para enviar e receber dados de telemetria em *streaming*, UDP ou TCP. Temos de ter em consideração os comportamentos operacionais de cada protocolo. Em UDP, não há garantia de entrega, o que tem um grande impacto neste contexto de *streaming telemetry*. Se o *collector* não receber nenhuma mensagem, o que pode significar? Pode indicar que o dispositivo de rede está em baixo ou simplesmente que a informação não chega ao *collector*. No entanto, pelo facto do protocolo UDP não garantir a entrega, não se consegue aferir com exatidão o que aconteceu [59]. Com o protocolo TCP, por outro lado, como temos garantia de entrega, podemos assumir, para um tipo de subscrição baseado em intervalos de tempo (*sample_intervals*), que se o *collector* não receber informação do dispositivo de rede é porque este foi abaixo ou tornou-se inacessível. Mas o protocolo TCP não tem só vantagens, face ao protocolo UDP, o protocolo TCP é o mais lento.

3.7 Conclusão

O conceito de *Network Telemetry* descrito no RFC 9232 [13] é a concertação de várias opções tomadas pelos administradores de redes, ao longo dos últimos anos, na área da telemetria nas redes. Existindo vários mecanismos e tecnologias existentes, descritos no capítulo 2, que podem ser mapeados para este conceito.

	Management Plane	Control Plane	Forwarding Plane
Data Configuration and Subscribe	gNMI, NETCONF, RESTCONF, SNMP, YANG-Push	gNMI, NETCONF, RESTCONF, YANG-Push	NETCONF, RESTCONF YANG-Push
Data Generation and Process	MIB, YANG	YANG	IOAM, PSAMP, PBT, AM
Data Encoding and Export	gRPC, HTTP, TCP	BMP, TCP	IPFIX, UDP

Tabela 3.1: Mapeamento de tecnologias e mecanismos para *Network Telemetry* [13]

4

Requisitos e Implementação

Conteúdo

Neste capítulo, é descrito em maior detalhe, o problema associado a este trabalho, a arquitetura do sistema operativo SR Linux para o desenvolvimento de aplicações e a solução implementada para este problema. As aplicações desenvolvidas em SR Linux são denominadas agentes, por isso, ao longo do presente relatório, trataremos por agente a aplicação desenvolvida.

O sistema operativo SR Linux, apesar de dispor de novas tecnologias que permitem a recolha de dados dos dispositivos de rede de uma forma eficiente, através do uso de interfaces de gestão modernas como gNMI ou JSON-RPC, também tem de suportar tecnologias mais antigas. O objetivo desta dissertação foi desenvolver um agente SNMP que interaja com o sistema SR Linux, fazendo o mapeamento entre uma subscrição gNMI interna e traps SNMP.

4.1 Requisitos do agente

Como referido no capítulo 1, o sistema operativo SR Linux é o novo sistema operativo da empresa de telecomunicações Nokia para dispositivos de rede, nos centros de dados. Este sistema foi desenvolvido segundo o conceito de *"model driven"* e dispõe de tecnologias modernas que permitem a gestão e monitorização deste dispositivo. Como podemos observar, pela Figura 4.1, na camada de *Management* temos três tecnologias distintas, que já foram referidas no capítulo 2, e que permitem essa mesma gestão e monitorização.

Apesar de o uso destas tecnologias ser mais eficiente que outras, consideradas desadequadas ao contexto atual das redes, o sistema SR Linux tem também de suportar estas tecnologias. Assim, o objetivo principal desta dissertação foi desenvolver um agente SNMP. No desenvolvimento deste agente, teve-se em consideração os seguintes critérios:

- O agente SNMP terá como única função operacional o envio de SNMP Traps, sendo que não responderá a qualquer outra operação que o protocolo SNMP dispõe.
- O agente terá de ser configurável a partir das interfaces de gestão gNMI, JSON-RPC e CLI como se observa na Figura 4.1. Para isso, terá de dispor de um modelo de dados em YANG que permita a correta gestão do agente.
- O agente terá de apresentar uma estrutura capaz de representar um evento, isto é, uma estrutura que explicite em que condições é que uma SNMP Trap é enviada, bem como outras informações úteis ao evento.
- O agente terá de suportar o envio de Traps para múltiplos destinos.

4.2 Arquitetura do SR Linux

O sistema operativo SR Linux providencia um mecanismo de desenvolvimento de aplicações (*Software Development Kit* (SDK)) denominado *NetOps Development Kit* (NDK). A partir deste NDK, é possível integrar agentes no sistema operativo SR Linux com outras aplicações nativas ou não. A interação entre o agente e o resto do sistema é possível através do serviço NDK *gRPC*, que expõe um conjunto de métodos RPC que permitem interagir com o resto do sistema, inclusive a baixo nível, através da *Impart Database* (IDB). A IDB é responsável pela comunicação da informação entre os diferentes agentes e o sistema, e guarda a informação em *protobufs*. Podemos observar estas interações na Figura 4.1 [60].

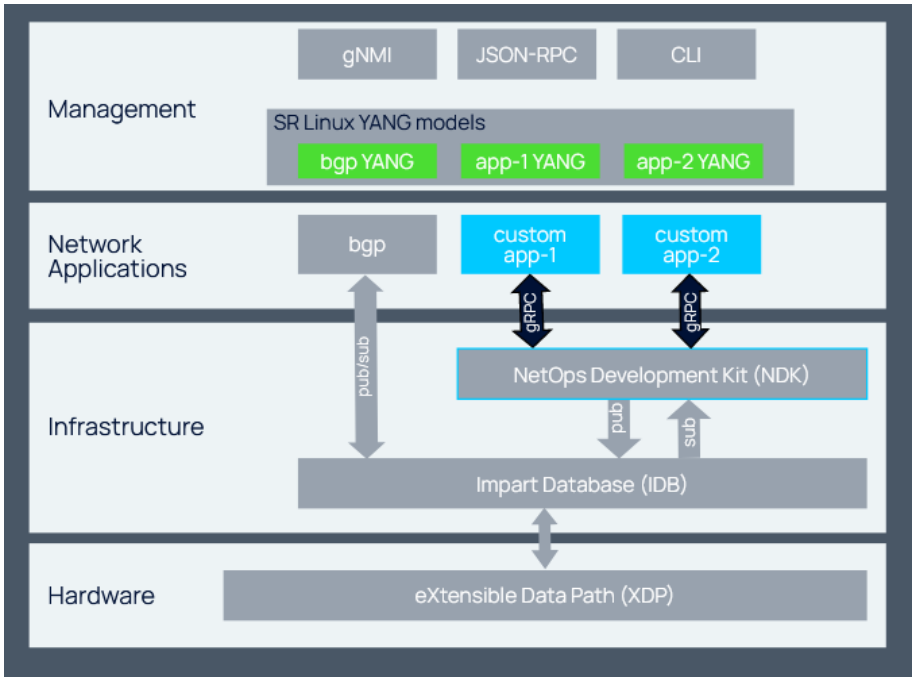

Figura 4.1: Integração de agentes no sistema SR Linux

Como é possível observar na Figura 4.1, o agente *custom app-1* interage com o NDK através de gRPC e o NDK interage com a IDB através de um mecanismo de *publish/subscribe*. Ao nível da gestão, vemos que a *custom app-1* tem de expor os seus modelos de dados YANG para o agente poder ser gerido através das tecnologias de gestão gNMI, JSON-RPC e CLI.

O SR Linux oferece um mecanismo denominado *Network Instance*, que permite fazer uma divisão virtual de encaminhamento, isto é, uma *Virtual Routing and Forwarding* (VRF). Cada instância de *network instance* tem a sua tabela de encaminhamento, a sua FIB, as suas interfaces e os protocolos de encaminhamento a correr nessa *network instance* [61]. Este mecanismo é tido em conta no desenho da solução, porque os *Collectors* podem estar em *network instances* diferentes.

O SR Linux utiliza a noção de *datastores*, explicada na secção 2.11, para gestão das configurações existentes no sistema. O SR Linux apresenta as seguintes *datastores*:

- *Running* — contém as configurações ativas no sistema.
- *State* — contém os dados de estado do sistema, isto é, dados que não podem ser alterados pelo administrador do dispositivo, por exemplo, estatísticas de uma interface.
- *Candidate* — contém as configurações que o utilizador pretende alterar. Quando um utilizador, via CLI, pretende fazer uma configuração entra no modo *candidate*, faz as alterações pretendidas e, depois, executa o *commit* para passar as configurações da *datastore candidate* para a *datastore running*.
- *Tools* — contém configurações que permitem executar comandos especiais, como reiniciar o dispositivo ou limpar as estatísticas de uma interface.

O NDK apresenta suporte a duas linguagens de programação Python e Go. Optou-se pelo desenvolvimento do agente em Python dada a experiência existente com esta linguagem por parte do discente.

Como já foi referido, para o agente ser gerido, através das interfaces de gestão que o sistema disponibiliza (gNMI, JSON-RPC e CLI), tem de ter o seu modelo de dados associado ao agente. O NDK sabe que modelo de dados corresponde a que agente através do ficheiro YAML do agente. Este ficheiro é responsável for fornecer informações ao NDK sobre o agente, nomeadamente o *path* do executável, o *path* do modelo de dados e os comandos que o agente suporta. Este ficheiro está detalhado no Apêndice C.10 [62].

A Figura 4.2 representa o fluxo de informação entre as interfaces de gestão (gNMI, JSON-RPC e CLI) e o agente. Quando é introduzida alguma configuração via alguma destas interfaces, é o serviço de *"mgmt srv"* que introduz estas configurações na IDB. Se o agente estiver à escuta de notificações de configuração irá recebê-las através do *"ndk mgr"* como representado neste fluxo. O mesmo acontece no sentido contrário, o agente pode introduzir dados na IDB que estarão acessíveis, a partir das interfaces de gestão.

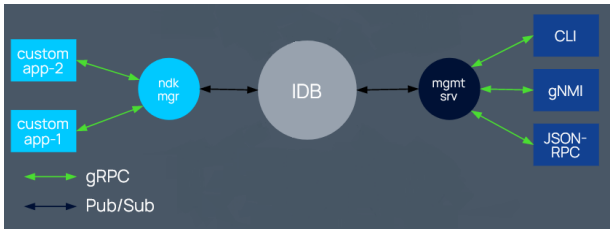

Figura 4.2: Canal de comunicação do agente para as Interfaces de Gestão

4.2.1 Métodos gRPC

Nesta secção, são detalhados alguns dos métodos gRPC que o NDK apresenta para interação entre o agente e o sistema SR Linux. Para a comunicação entre o NDK e o agente começar, terá de ser criado um canal gRPC. O NDK usa *localhost:50053*, para executar o servidor gRPC, que responderá aos pedidos que serão detalhados, a seguir. É de realçar que estes métodos RPC são os que foram usados no desenvolvimento deste agente e, por isso, serão aqui detalhados. O NDK oferece mais métodos gRPC para comunicação com o sistema. Estes estão disponíveis no repositório GitHub *nokia/srlinux-ndk-protobufs*[1], mas não serão aqui descritos.

4.2.1.1 Registo do Agente

Para que o sistema SR Linux saiba que existe um agente não nativo em execução, este tem de se registar no sistema através da função:

```
1 service sdk_service{
2     rpc AgentRegister(AgentRegistrationRequest) returns AgentRegistrationResponse
3 }
```

Bloco de Código 4.1: Função *AgentRegister*

Esta função recebe, como parâmetro de entrada, um *AgentRegistrationRequest*. Este objeto contém o nome do agente e um parâmetro chamado *agent_liveliness* que permite criar um mecanismo de *KeepAlive*, que será explicado na secção 4.2.1.4. A função retorna um objeto que contém: i) o estado do registo, *"kOk"* para registo sem problemas ou *"kFailed"* se o registo falhou; ii) se o registo falhou são retornados no campo *"error_str"* detalhes sobre o erro; iii) *"app_id"* que é o identificador do agente atribuído pelo *SDK manager*.

4.2.1.2 Registo de notificações

O agente pode fazer o registo de notificações junto no *SDK manager*. Estas notificações são alertas que, dependendo da subscrição, irão enviar para o agente, informações atualizadas sobre o estado de algum componente do sistema.

```
1 service sdk_service{
2     rpc NotificationRegister(NotificationRegisterRequest)
3                       returns NotificationRegisterResponse
4 }
```

Bloco de Código 4.2: Função *NotificationRegister*

[1] https://github.com/nokia/srlinux-ndk-protobufs/tree/v0.1.0/ndk

Esta função recebe, como parâmetro de entrada, um *NotificationRegisterRequest*. Este objeto contém um *stream_id*, a operação a realizar e que tipo de subscrição é que o pedido pretende fazer. A função retorna um objeto que contém: i) *stream_id* — um identificador da stream; ii) *sub_id* um identificador da subscrição, para poder ser eliminada; iii) *status* — o estado do registo.

4.2.1.3 Receção das notificações

Para receber as informações das subscrições ativas, o agente utiliza esta função bloqueante. A função é bloqueante, porque a função vai recebendo as notificações em *streaming*, durante o tempo de vida do agente.

```
1 service sdk_service{
2     rpc NotificationStream(NotificationStreamRequest)
3                       returns NotificationStreamResponse
4 }
```

Bloco de Código 4.3: Função *NotificationStream*

Esta função recebe, como parâmetro de entrada, um *NotificationStreamRequest*, este objeto contém um *stream_id*, o identificador retornado pela função *NotificationRegister*, que permite identificar a *stream* com as subscrições identificadas. A função retorna um objeto denominado por *NotificationStreamResponse* que contém uma lista de *Notification*. Este objeto *Notification* contém a informação pretendida pelas subscrições.

4.2.1.4 Mecanismo de *KeepAlive*

Esta função permite avisar o *SDK manager*, que o agente ainda se encontra em execução.

```
1 service sdk_service{
2     rpc KeepAlive(KeepAliveRequest)
3                       returns KeepAliveResponse
4 }
```

Bloco de Código 4.4: Função *KeepAlive*

A função recebe, como parâmetro de entrada, um *KeepAliveRequest*, este objeto é um objeto vazio, não contém nenhuma informação. A função retorna um objeto, denominado por *KeepAliveResponse* que contém o estado da execução da função. O *SDK manager* conclui que o agente parou a execução se não receber um *keepAlive*, no tempo registado com a função *AgentRegister*.

4.2.1.5 Atualização de estado do agente

Uma das funções mais importantes de um agente é a comunicação da sua informação de estado com o *SDK manager*. Esta troca de informação acontece através do serviço de telemetria disponibilizado pelo SR Linux e especificado no ficheiro *telemetry_service.proto*. Através dos métodos disponibilizados por este serviço, o sistema SR Linux, que utiliza modelos de dados YANG para expor os seus dados, introduz esses dados, para serem acessíveis por outros agentes, no sistema, e, para fora do sistema, através das tecnologias de gestão que o sistema suporta. O serviço apresenta dois métodos que permitem a correta troca de informação de telemetria com o *SDK manager*.

Adicionar dados de Telemetria

```
1 service telemetry_service{
2  rpc TelemetryAddOrUpdate(TelemetryUpdateRequest) returns TelemetryUpdateResponse
3 }
```

Bloco de Código 4.5: Função *TelemetryAddOrUpdate*

Esta função é responsável por adicionar dados de telemetria à *datastore state*. A função recebe como parâmetro de entrada o objeto *TelemetryUpdateRequest*. Neste objeto, é introduzido uma *key* e um objeto em JSON que permite a correta introdução dos dados no modelo de dados YANG definido para o agente, descrito no Bloco de Código 4.9. A função retorna um objeto que contém informação, se a chamada à função foi bem sucedida ou não.

Remover dados de Telemetria

```
1 service telemetry_service{
2  rpc TelemetryDelete(TelemetryDeleteRequest) returns TelemetryDeleteResponse
3 }
```

Bloco de Código 4.6: Função *TelemetryDelete*

Esta função é responsável por remover dados de telemetria à *datastore state*. A função recebe como parâmetro de entrada o objeto *TelemetryDeleteRequest*. Neste objeto, é introduzido uma *key* de modo a identificar o elemento YANG que é para eliminar. A função retorna um objeto que contém informação, se a chamada à função foi bem sucedida ou não.

4.3 Implementação do agente

Nesta secção, apresentaremos o fluxograma da nossa solução que mostra os componentes da mesma, as interações entre esses mesmos componentes, os requisitos operacionais da solução proposta, bem como outras considerações relativas ao desenvolvimento da solução idealizada. Este fluxograma representa os principais componentes do agente, sem especificar as classes e funções Python desenvolvidas. O código do agente está disponível no repositório GitHub *GascPT/snmp_agent* [16]. O Apêndice C detalha o código desenvolvido, para cada componente do fluxograma. Será dada, em cada componente, uma referência para o Apêndice C.

4.3.1 Fluxograma do agente

Na Figura 4.3, podemos observar o fluxograma da solução idealizada para o agente SNMP. Esta solução está dividida em duas fases de execução: i) a Fase Inicial, onde as configurações iniciais são introduzidas no agente; ii) a Fase RunTime que consiste num loop infinito, representado na Figura pelo símbolo azul, à direita, enquanto o agente estiver em execução. Ambas as fases serão detalhadas na secção 4.3.1.1.

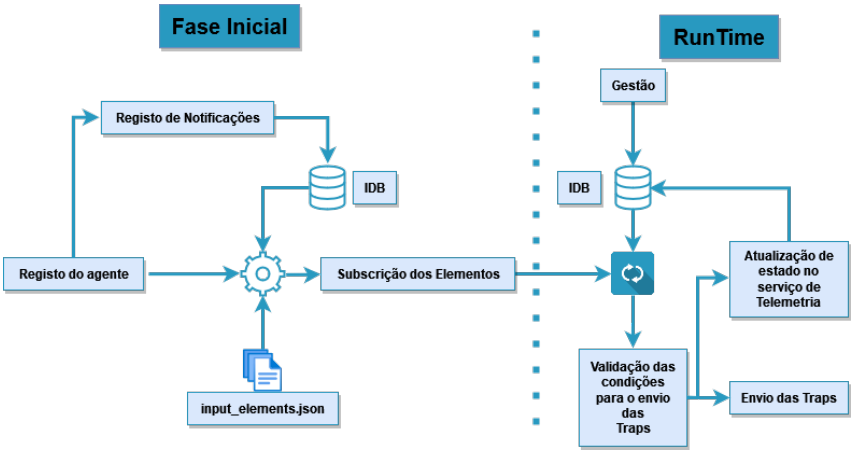

Figura 4.3: Fluxograma do agente SNMP

Na solução idealizada, introduziu-se o conceito de elemento, isto é, cada SNMP Trap está associada a um elemento que se pretende monitorizar. Este elemento é representado por um objeto em JSON que dispõe de diferentes parâmetros, que permitem fazer o mapeamento funcional da subscrição gNMI para SNMP Trap. A constituição de um elemento será detalhada na secção 4.3.1.1. Por exemplo, podemos estar a monitorizar, quando é que a temperatura do CPU fica abaixo de um certo valor. Neste caso, o elemento que reflete este evento terá informação suficiente para conseguir enviar a SNMP Trap nas condições certas.

4.3.1.1 Componentes da solução

Nesta secção, é detalhado cada componente do fluxograma da solução, representado na Figura 4.3.

4.3.1.1.1 Fase Inicial

Para o correto funcionamento do agente, este tem um conjunto de funções a realizar, antes de entrar no estado estável de execução. Essas funções são as seguintes:

4.3.1.1.1.1 Registo do agente

Como já referido na secção 4.2.1.1, o agente precisa de se registar perante o *SDK manager*, utilizando a função *AgentRegister()*, por forma a poder interagir com o mesmo. Este passo marca o arranque do agente, como representado no fluxograma da Figura 4.3 (Apêndice C.1).

4.3.1.1.1.2 Leitura do ficheiro de configuração *"input_elements.json"*

Na solução idealizada, propôs-se o uso de um ficheiro de configuração, no formato JSON, com a seguinte informação:

- *Targets*: contém informação sobre o destino das Traps;
- *Monitoring Elements*: contém informação sobre os elementos que se pretende monitorizar;

O *Targets* é uma lista de *target* que é um objeto composto por três parâmetros:

- *address* : Endereço de destino a usar no envio SNMP Trap, isto é, o endereço do *Collector*;
- *nw-instance* : *Network Instance* por onde enviar a SNMP Trap;
- *community_string* : *Community string* a utilizar no envio da SNMP Trap;

No Bloco de Código 4.7 podemos observar um exemplo deste objeto.

```
"targets": [
        {
            "nw-instance": "srbase-mgmt",
            "address": "172.20.20.1",
            "community_string": "public"
        },
        ...
]
```

Bloco de Código 4.7: Exemplo *Target* do ficheiro *input_elements.json*

O *Monitoring Elements* é uma lista de elementos. Cada elemento é um objeto composto pelos seguintes parâmetros:

- ***resource*** : caminho para o recurso que se pretende monitorizar. Dada a arquitetura modular com formato de árvore do SR Linux é possível indicar o recurso que se pretende monitorizar, através da notação YANG *Path*, como explicado na secção 2.9.
- ***parameter*** : *leaf* que se pretende monitorizar, dentro da *resource*.
- ***monitoring_condition*** : condição de monitorização, isto é, apenas se vai monitorizar um elemento, caso esta condição seja verdadeira. Este parâmetro pode não ser preenchido.
- ***resource_filter*** : se o *resource* tiver uma *key* genérica ("*"), este parâmetro filtra essa *key*, isto é, só vão ser monitorizadas as *keys* presentes neste parâmetro.
- ***trigger_condition*** : condição para envio da SNMP Trap.
- ***trigger_message*** : mensagem que será enviada se a condição indicada no campo *trigger_condition* for verdadeira.
- ***resolution_condition*** : condição para envio da SNMP Trap, neste caso de resolução.
- ***resolution_message*** : mensagem que será enviada se a condição indicada no campo *resolution_condition* for verdadeira.
- ***trap_oid*** : OID que será enviado na mensagem SNMP Trap.

Cada um dos parâmetros destes dois objetos foram pensados para endereçar os requisitos operacionais do agente. O agente irá utilizar o objeto *element*, para mapear corretamente as subscrições gNMI com o envio da SNMP Trap. À semelhança do ficheiro de configuração com o formato em JSON, o modelo de dados do agente YANG também reflete os parâmetros deste objeto. É possível observar as semelhanças destes dois formatos nos Blocos de Código 4.8 e 4.9.

Por exemplo, no Bloco de Código 4.8, temos um elemento que permite a monitorização do estado de todas as interfaces do sistema. O *resource* e o *parameter* apontam para o estado das interfaces **/interface[name=*]/oper-state**. A condição obriga a que apenas monitorizaremos as interfaces que tenham o *admin-state enable*. Como a *key* é genérica, queremos filtrar apenas as interfaces que tenham como *key ethernet**, *lag** ou *mgmt**. Se retirássemos algum destes elementos da lista, as interfaces referenciadas por esse elemento já não seriam monitorizadas. Por exemplo, se retirássemos *mgmt** todas as interfaces *mgmt0*, *mgmt1*, etc, não seriam monitorizadas. A condição para o envio da SNMP Trap é quando o *oper-state* passar de *up* para *down* e unicamente nessa condição e ordem.

Isto está refletido no parâmetro *trigger_condition* com o uso da seta que reflete essa condição. No desenvolvimento do agente, foram previstas as seguintes condições:

- **A->B** : O estado passar de A para B, sem passar em nenhum estado intermédio.
- **!=A** : O estado atual diferir de A, sendo que o estado anterior foi A.
- **A>** : O estado atual ser menor que A, sendo A um valor numérico.
- **A<** : O estado atual ser maior que A, sendo A um valor numérico.

À semelhança do campo *trigger_condition*, o campo *resolution_condition* usa a mesma notação para verificar as condições de envio das SNMP Traps. O campo *trigger_message* introduz na SNMP Trap a mensagem *oper-down-reason*, quando uma interface que estiver a ser monitorizada for abaixo, isto é, a condição imposta na *trigger_condition* for verdadeira.

Por outro lado, a *resolution_message* introduz na SNMP Trap a mensagem *operUP*, quando uma interface, que estiver a ser monitorizada, for acima, isto é, a condição imposta na *resolution_condition* for verdadeira. Por último, o OID a ser utilizado, no envio da SNMP Trap será neste exemplo **.1.3.6.1.4.1.94.2500.3001**. Na secção 4.3.1.1.2.4 será explicado, mais detalhadamente, o uso deste parâmetro, para o envio das SNMP Traps (Apêndice C.2).

```
"monitoring_elements": [
        {
            "resource": "interface[name=*]/",
            "parameter": "oper-state",
            "monitoring_condition": "admin-state=enable",
            "resource_filter": [
                "ethernet*",
                "lag*",
                "mgmt*"
            ],
            "trigger_condition": "up->down",
            "trigger_message": "oper-down-reason",
            "resolution_condition": "down->up",
            "resolution_message": "operUP",
            "trap_oid": ".1.3.6.1.4.1.94.2500.3001"
        },
        ...
]
```

Bloco de Código 4.8: Exemplo de *Monitoring Elements* do ficheiro *input_elements.json*

4.3.1.1.1.3 Registo de Notificações

Por forma a receber notificações do sistema, o agente precisa de se registar no SDK manager, através da função *NotificationRegister()*. O agente foi desenhado para apenas precisar de receber notificações do serviço *config_service.proto*. Este serviço recebe a informação sobre o agente da *datastore running*, e envia-a para o agente através da *stream* que o agente inicia com o serviço através da função *NotificationStream()*, como explicado na secção 4.2.1.3 (Apêndice C.3).

4.3.1.1.1.4 Leitura das configurações guardadas na IDB

O sistema SR Linux guarda as configurações dos agentes na *datastore running*, isto é, quando um agente se encontra em execução a *datastore running* contém as configurações ativas desse agente. Como explicado na secção 2.5, existe uma relação entre as *datastores candidate*, *running* e *state*. Estas *datastores* são geridas pelo "mgmt_srv" da Figura 4.2 sendo guardadas na IDB. À semelhança do ficheiro de configuração *"input_elements.json"*, o modelo de dados YANG do agente foi desenhado para garantir o correto mapeamento entre a subscrição gNMI, que será explicada na secção 4.3.1.1.1.5, e a SNMP Trap. Por isso, quando o agente recebe as notificações com as configurações existentes no sistema, este consegue facilmente introduzi-las nos objetos corretos (Apêndice C.4).

4.3.1.1.1.5 Subscrição dos Elementos

Posteriormente à leitura, tanto do ficheiro de configuração *"input_elements.json"* como da *datastore running*, o agente terá, em memória, todos os elementos que pretende monitorizar. Com recurso aos parâmetros *resource* e *parameter* de cada elemento, é possível criar para cada elemento um gNMI Path para a subscrição. Desta forma, o agente irá utilizar *unix sockets*, para comunicar com o servidor gNMI, internamente. Para isso, teve de habilitar-se essa opção no servidor gNMI do sistema SR Linux.

Utilizou-se a biblioteca pygnmi[2], para servir de cliente no agente. Esta biblioteca oferece suporte às operações existentes no gNMI, que estão detalhadas na secção 2.7. A função de subscrição disponibilizada pela biblioteca é bloqueante, pois abre uma *stream* com o servidor. Por isso, esta componente da solução foi implementada numa *thread*, à parte da execução principal do agente. Quando o agente recebe alguma mensagem por esta subscrição, colocará essa mensagem numa fila para ser processada pela execução principal do agente (Apêndice C.5).

[2] https://pypi.org/project/pygnmi/

4.3.1.1.2 Fase RunTime

Ao entrar nesta fase de execução, o agente encontra-se no estado estável, à escuta de alterações nos elementos que tem subscrito e à escuta de alterações nas configurações do próprio agente. Cada uma destas funcionalidades tem um conjunto de funções associada:

4.3.1.1.2.1 Gestão

No âmbito da gestão, é possível gerir o agente através das interfaces de gestão que o sistema SR Linux disponibiliza (gNMI, JSON-RPC, CLI), como representado na Figura 4.1. No Bloco de Código 4.9, podemos observar a estrutura hierarquizada em árvore do modelo de dados YANG do agente que as interfaces utilizam, para a interação com o agente. O ficheiro YANG do agente está detalhado no Apêndice C.11.

```
snmp-agent
+-- monitoring_elements
|   +-- element [resource]
|       +-- parameter
|       +-- monitoring_condition
|       +-- trigger_condition
|       +-- trigger_message
|       +-- resolution_condition
|       +-- resolution_message
|       +-- trap_oid
|       +-- resource_filter
+-- targets
    +-- target [address]
        +-- network-instance
        +-- community-string
```

Bloco de Código 4.9: Estrutura em árvore do modelo de dados YANG do agente

Em execução, é possível inserir, remover e alterar qualquer elemento deste modelo de dados. Na inserção de dados, é preciso ter em atenção que há parâmetros obrigatórios e outros não. Os parâmetros considerados obrigatórios no *container element* são os parâmetros *resource*, *parameter*,*trigger_condition* e *trap_oid*. No *container target* são os parâmetros *address* e *community-string*. A definição de parâmetros obrigatórios permite que as interfaces de gestão (gNMI, CLI, JSON-RPC) apenas possam fazer *commit*, quando tiverem informação suficiente para o correto envio das SNMP Traps. Quando acontece alguma destas operações ao nível das *datastores*, isto é, *commit* da *datastore candidate* para a *datastore running*, o ficheiro de configuração *input_elements.json* também é alterado, mantendo assim consistência entre estes dois componentes da solução (Apêndice C.6).

4.3.1.1.2.2 Validação das condições

Quando existe alguma alteração de estado, em algum dos elementos que o agente monitoriza, este recebe essa alteração, por via da subscrição detalhada na secção 4.3.1.1.1.5. Cabe a este componente do agente fazer o processamento da alteração de estado, de modo a garantir que cumpre as condições impostas para o envio da SNMP Trap, descritas no campo *trigger_condition* e *resolution_condition* de cada elemento. As possíveis combinações de condições de cada elemento estão detalhadas na secção 4.3.1.1.1.2 (Apêndice C.7).

4.3.1.1.2.3 Atualização de Estado

Como já foi explicado, o SDK manager disponibiliza um serviço de telemetria para a *datastore state*. Quando o agente envia uma SNMP Trap, associada a algum elemento, este apresenta um contador que será atualizado conforme forem enviadas as SNMP Traps (Apêndice C.8).

4.3.1.1.2.4 Envio das Traps

Caso estejam reunidas as condições para o envio das SNMP Traps, é nesta fase que são enviadas. A SNMP Trap vai ser enviada para todos os *Targets* configurados no agente, usando o endereço IP, a *network-instance* e a *community-string* que existem neste objeto. Do objeto *element* será utilizado o campo *trigger_message* se a condição válida for a *trigger_condition* ou vai ser utilizado o campo *resolution_message* se a condição válida for a *resolution_condition*. Deste elemento será também utilizado o *trap_oid*. Para o envio da SNMP Trap, usou-se a ferramenta *snmptrap* com o comando do Bloco de Código 4.10:

```
1 snmptrap -v 2c -c <community-string> <address> 0 <trap-oid> <trap-oid.1> s <string>
```

Bloco de Código 4.10: Comando para envio da SNMP Trap

Caso a condição verdadeira seja *trigger_condition*, será adicionado o .1 no segundo *trap_oid* do comando, como observamos no Bloco de Código 4.10. Caso a condição verdadeira seja *resolution_condition*, será adicionado .2 no segundo *trap_oid*. Desta forma, consegue-se diferenciar quando é despoletado um *trigger* ou um *resolution* sobre um dos eventos, que esteja configurado no agente (Apêndice C.9).

5

Testes e Resultados

Conteúdo

Neste capítulo, apresentamos os resultados obtidos para as simulações realizadas. O capítulo está dividido nas seguintes secções: em primeiro lugar, é descrita a avaliação proposta para o agente; em segundo lugar, é descrito o ambiente de virtualização *ContainerLab*; em terceiro lugar, são apresentados os cenários propostos para correta avaliação do sistema; em quarto lugar, são descritos os resultados obtidos nos cenários propostos.

5.1 Avaliação do agente

A avaliação do agente não se prende por métodos quantitativos, como, por exemplo, medição do tempo de envio da SNMP Trap, mas sim pela verificação do correto funcionamento das funcionalidades que o agente dispõe. Tendo em conta os requisitos propostos para o desenvolvimento do agente, a avaliação prende-se principalmente pela verificação das seguintes funcionalidades:

- **Ponto 1** - Mapeamento da subscrição gNMI e o evento ocorrido.
  - A) Verificação da utilização dos parâmetros *resource* e *parameter* na subscrição gNMI;
  - B) Verificação das condições (*trigger_condition* e *resolution_condition*);
  - C) Correta utilização dos parâmetros descritos no evento;
- **Ponto 2** - Envio das SNMP Traps;
  - A) Uso dos endereços IP e *Network Instances*.
  - B) Correta utilização da *community string* configurada.
- **Ponto 3** - Consistência entre o ficheiro de configuração, a *datastore running* e *state*;

Ressalve-se que nem todos os pontos de avaliação propostos se aplicam em todos os cenários de simulação.

Tendo em conta estes parâmetros, na secção 5.4, será feita a avaliação do agente nos cenários propostos. Por forma a analisar os pacotes enviados pelo agente, será utilizada a ferramenta WireShark[1]. Esta ferramenta permite desconstruir as várias camadas de um pacote, por forma a analisá-lo. Ao longo da secção 5.4, serão demonstradas várias capturas Wireshark, que foram obtidas durante a execução do agente e nas condições dos cenários associados.

[1] https://www.wireshark.org/

5.2 ContainerLab

Para virtualização do sistema operativo SR Linux, utilizou-se a ferramenta de virtualização Containerlab. Esta ferramenta fornece uma CLI que permite a orquestração e a gestão de topologias de rede, baseadas em *containers*, em que cada dispositivo de rede corresponde a um *container* diferente. O ContainerLab inicia os *containers*, o que cria um ambiente de virtualização, criando ligações entre os *containers* presentes na topologia, o que permite criar diferentes tipos de topologias de rede, à escolha dos utilizadores. A ferramenta também permite a gestão do ciclo de vida destes dispositivos e os devidos acessos aos dispositivos da rede. O Apêndice A descreve a instalação desta ferramenta, bem como comandos básicos, para correta utilização da ferramenta.

5.3 Cenários e Condições

Para verificação do correto funcionamento do agente, propuseram-se três cenários diferentes de simulações.

- **Cenário 1** - Um dispositivo SR Linux ligado a um *Collector*. Neste caso o *Collector* será o *host* que corre o ContainerLab.

- **Cenário 2** - Um dispositivo SR Linux, a correr o agente, ligado a diferentes dispositivos SR Linux. Onde o primeiro, o que corre o agente, está ligado a um *Collector*, neste caso o *Collector* será o *host* que corre o ContainerLab.

- **Cenário 3** - Um dispositivo SR Linux, a correr o agente, ligado a três *Collectors*, um deles o *host* que corre o ContainerLab.

Para cada um dos cenários propostos, o agente é posto em execução com SNMP Traps, de modo a refletir possíveis pontos de falha. Diferentes elementos serão tidos em conta nestas simulações.

5.4 Resultados

Nesta secção, apresentamos os resultados que obtivemos para as simulações que realizámos. Em cada um dos cenários, será tida em conta a avaliação proposta na secção 5.1, para demonstração do correto funcionamento do agente.

5.4.1 Cenário 1

Este é o primeiro cenário de simulação, o mais básico entre os três cenários propostos, para avaliar o agente. Neste caso, o cenário consiste num dispositivo SR Linux ligado a um *Collector*, como ilustrado na Figura 5.1.

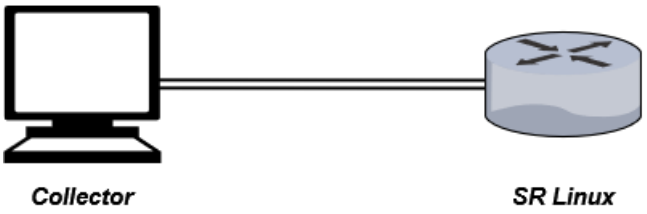

Figura 5.1: Cenário 1

Neste cenário, o *Collector* está ligado ao SR Linux, através da *network instance* de gestão (*mgmt*). O Collector é o próprio *host* que executa o ContainerLab e corre a ferramenta, *snmptrapd*, para escutar as SNMP Traps a ele destinado. Para gestão do dispositivo de rede, o Containerlab define o *host* com o IP 172.20.20.1 (este IP é semelhante nos restantes cenários propostos). Como descrito na secção 5.1, para se avaliar o agente tem de se confirmar o correto funcionamento do mesmo, segundo os parâmetros descritos nessa secção. Assim, para verificar o correto funcionamento do agente para este cenário, configurou-se o agente com o seguinte evento, cuja estrutura está explicada na secção 4.3.1.1.1.2.

```
"monitoring_elements": [
        {
            "resource": "interface[name=*]/",
            "parameter": "oper-state",
            "monitoring_condition": "admin-state=enable",
            "resource_filter": [
                "ethernet*",
                "lag*",
                "mgmt*"
            ],
            "trigger_condition": "up->down",
            "trigger_message": "oper-down-reason",
            "resolution_condition": "down->up",
            "resolution_message": "operUP",
            "trap_oid": ".1.3.6.1.4.1.94.2500.3001"
        }
]
```

Bloco de Código 5.1: Elemento de Teste para o cenário 1

Este elemento permite enviar uma SNMP Trap ao *Collector*, quando uma interface for abaixo, isto é, o *oper-state* da interface passar de *up* para *down*. Nestas condições realizou-se a simulação, desligando a interface ethernet-1/1 do *SR Linux*, obtendo-se assim o cenário de simulação adequado para despoletar o evento descrito no Bloco de Código 5.1.

```
Frame 955: 190 bytes on wire (1520 bits), 190 bytes captured (1520 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 172.20.20.2, Dst: 172.20.20.1
User Datagram Protocol, Src Port: 42027, Dst Port: 162
Simple Network Management Protocol
   version: v2c (1)
   community: public
▾ data: snmpV2-trap (7)
   ▾ snmpV2-trap
        request-id: 1316021191
        error-status: noError (0)
        error-index: 0
      ▾ variable-bindings: 3 items
        ▸ 1.3.6.1.2.1.1.3.0: 0
        ▸ 1.3.6.1.6.3.1.1.4.1.0: 1.3.6.1.4.1.94.2500.3001 (iso.3.6.1.4.1.94.2500.3001)
        ▸ 1.3.6.1.4.1.94.2500.3001.1: "oper-down-reason: interface[name=ethernet-1/1]/oper-state"
```

Figura 5.2: Cenário 1 - Captura Wireshark no Collector

```
element "interface[name=*]/" {
    traps_generated 1
}
```

Figura 5.3: Cenário 1 - Datastore *State* no agente

A Figura 5.2 foi capturada no *Collector* e mostra que a SNMP Trap foi corretamente recebida. Nesta Figura, pode observar-se o uso correto do endereço IP de destino (endereço do *Collector* — 172.20.20.1), o correto uso da porta de destino (162) e, na camada SNMP do pacote, podemos observar o uso da *community string*, a versão usada, bem como os *variable bindings*. Neste caso, esta SNMP Trap apresenta três *variable bindings* diferentes. O primeiro *variable binding* diz respeito aos *timeticks* e é obrigatório o envio, por falta desta informação no agente é sempre enviado o valor 0 neste campo. O segundo *variable binding* é a OID configurada no evento. O terceiro *variable binding* é uma string, como foi adicionado o .1 ao OID original, sabemos que este foi despoletado pelo *trigger_condition* do evento configurado, o que coincide com a simulação proposta. A Figura 5.3 mostra que a *datastore state* foi atualizada corretamente, segundo a lógica proposta para o agente.

De acordo com a avaliação proposta na secção 5.1, neste cenário, o agente cumpre com alguns dos pontos propostos e que são percetíveis para este cenário, pois faz: **1.A)** o correto mapeamento do que recebe pela subscrição gNMI para o evento descrito, **1.B)** distingue propriamente o *trigger_condition* com o *resolution_condition*, **2.A) e 2.B)** realiza o correto envio da SNMP Trap para o destino configurado e a **3)** correta atualização da *datastore state*.

5.4.2 Cenário 2

O segundo cenário de simulação consiste num dispositivo SR Linux, a correr o agente, ligado a diferentes dispositivos SR Linux, onde o primeiro dispositivo, o que corre o agente, está ligado a um *Colletor*, como ilustrado na Figura 5.4.

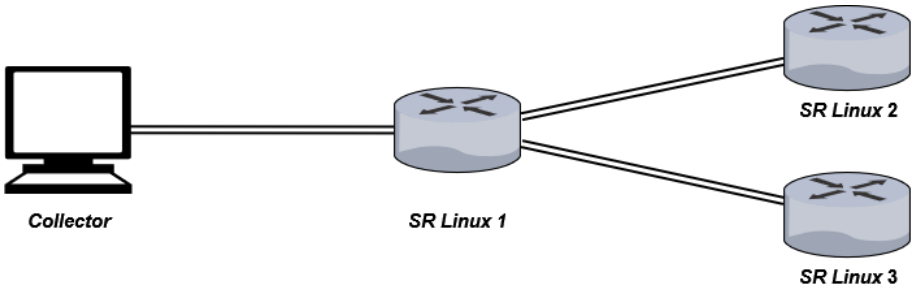

Figura 5.4: Cenário 2

Neste cenário, é verificado o correto funcionamento do agente, para eventos influenciados por condições externas ao próprio agente, por exemplo, uma sessão BGP com um vizinho cair. Neste cenário, é posto à prova o correto mapeamento do agente, passando certos requisitos operacionais que irão ser explicados mais adiante. Semelhante ao cenário 1, o *Collector* é o próprio *host* que corre o Containerlab. Assim, para verificar o correto funcionamento do agente, para estas condições, configurou-se o agente com o seguinte evento, cuja estrutura de evento está explicada na secção 4.3.1.1.1.2.

```
"monitoring_elements": [
    {
        "resource": "/network-instance[name=X]/protocols/bgp/neighbor[peer-address=*]/",
        "parameter": "session-state",
        "monitoring_condition": "admin-state=enable",
        "resource_filter": [
        ],
        "trigger_condition": "!=established",
        "trigger_message": "bgpNeighborClosedTCPConn",
        "resolution_condition":"==established",
        "resolution_message":"bgpNeighborEstablished",
        "trap_oid":".1.3.6.1.4.1.94.2500.3002"
    }
]
```

Bloco de Código 5.2: Elemento de Teste para o cenário 2

Para a concretização desta simulação, configuraram-se duas sessões BGP diferentes, em duas *Network Instances* diferentes, denominadas SRL1-SRL2 e SRL1-SRL3. Para individualizar cada SNMP Trap e evento correspondente, usou-se o evento descrito no Bloco de Código 5.2, em que consoante a *Network Instance* introduziu-se o nome da mesma, na linha 3 do elemento. A ordem de eventos da simulação foi a seguinte:

- **Passo 1** - Desligou-se o vizinho BGP no SRL2.
- **Passo 2** - Ligou-se o vizinho BGP no SRL2.
- **Passo 3** - Desligou-se o vizinho BGP no SRL3.
- **Passo 4** - Ligou-se o vizinho BGP no SRL3.

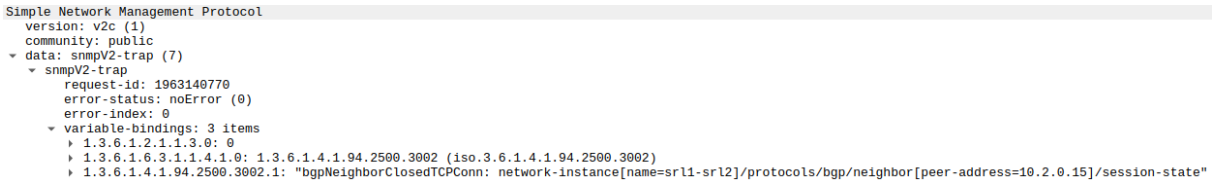

Figura 5.5: Cenário 2 - SNMP Trap recebida pelo Colletor associada à *Network Instance* SRL1-SRL2

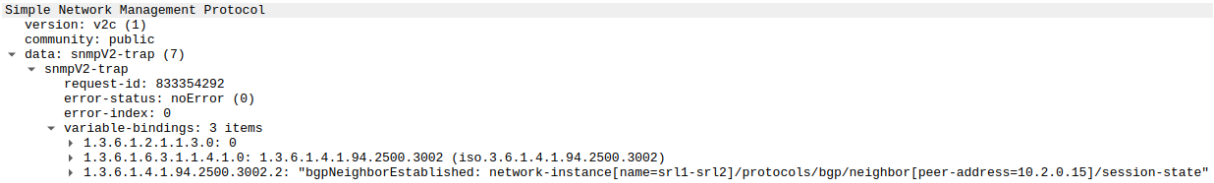

Figura 5.6: Cenário 2 - SNMP Trap recebida pelo Colletor associada à *Network Instance* SRL1-SRL2

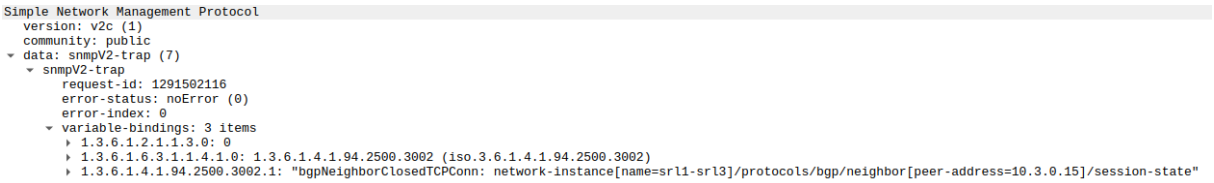

Figura 5.7: Cenário 2 - SNMP Trap recebida pelo Colletor associada à *Network Instance* SRL1-SRL3

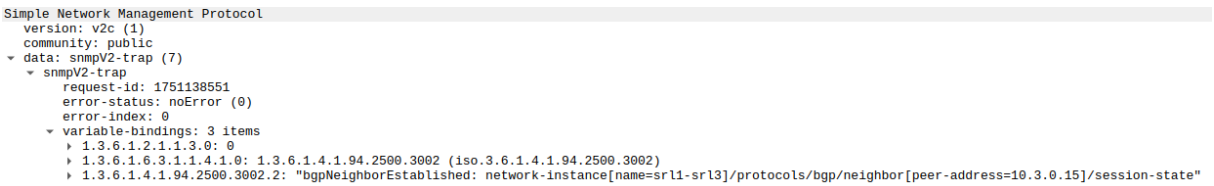

Figura 5.8: Cenário 2 - SNMP Trap recebida pelo Colletor associada à *Network Instance* SRL1-SRL3

As Figuras 5.5 e 5.6 demonstram o correto envio da SNMP Trap, quando se desligou o vizinho BGP entre o SR Linux, que corre o agente, e o SRL2. À semelhança destas Figuras, as Figuras 5.7 e 5.8 demonstram o correto envio da SNMP Trap, quando se desligou o vizinho BGP no SRL3. Vale a pena referir que estas capturas WireShark, representadas pelas Figuras 5.5, 5.6, 5.7 e 5.8, foram obtidas no *Collector*. Pela última linha de cada Figura, é possível observar o evento ocorrido em cada SNMP Trap. Nas Figuras 5.5 e 5.7, o evento *bgpNeighborClosedConn* e nas Figuras 5.6 e 5.8, o evento *bgpNeighborEstablished*, seguindo a ordem de eventos que se propôs para este cenário.

Através deste cenário de simulação, é possível verificar o correto mapeamento do evento gerado pela subscrição gNMI e o elemento configurado, bem como o correto funcionamento da *trigger_condition* e da *resolution_condition*. Neste caso, o *session-state* do elemento pode ter diferentes estados, mas só é despoletado o evento, quando passa de *established* para outro estado diferente e não quando existe transição entre estados diferentes, que não *established*, como se pretende pela definição do elemento. Também se pode verificar a mesma lógica no *resolution_condition*, em que só se observa o envio desta SNMP Trap, quando o estado passa de um estado anterior qualquer para *established*.

```
snmp-agent {
    monitoring_elements {
        element "/network-instance[name=srl1-srl2]/protocols/bgp/neighbor[peer-address=*]/" {
            traps_generated 2
        }
        element "/network-instance[name=srl1-srl3]/protocols/bgp/neighbor[peer-address=*]/" {
            traps_generated 2
        }
    }
}
```

Figura 5.9: Cenário 2 - *Datastore State*

Pela Figura 5.9 podemos verificar a correta atualização da *datastore state*, em que houve separação das SNMP Traps geradas por cada elemento corretamente.

Segundo a avaliação proposta na secção 5.1, neste cenário o agente cumpre só alguns dos pontos propostos e que são percetíveis para este cenário, pois faz: **1.A)** o correto mapeamento do que recebe pela subscrição gNMI para o evento descrito; **1.B)** distingue propriamente o *trigger_condition* com o *resolution_condition*, ambos estes pontos podem ser verificados pelas Figuras 5.5, 5.6, 5.7 e 5.8; **1.C)** pelas Figuras já mencionadas, é possível verificar o correto uso do resto dos parâmetros do elemento, como o uso do campo *trigger_message* nas Figuras 5.5 e 5.7 e do campo *resolution_message* nas Figuras 5.6 e 5.8; **2.A) e 2.B)** realiza o correto envio da SNMP Trap para o destino configurado; **3)** correta atualização da *datastore state* com a informação pretendida.

5.4.3 Cenário 3

O terceiro cenário de simulação consiste num dispositivo SR Linux, a correr o agente, ligado a três *Collectors*, como ilustrado na Figura 5.10.

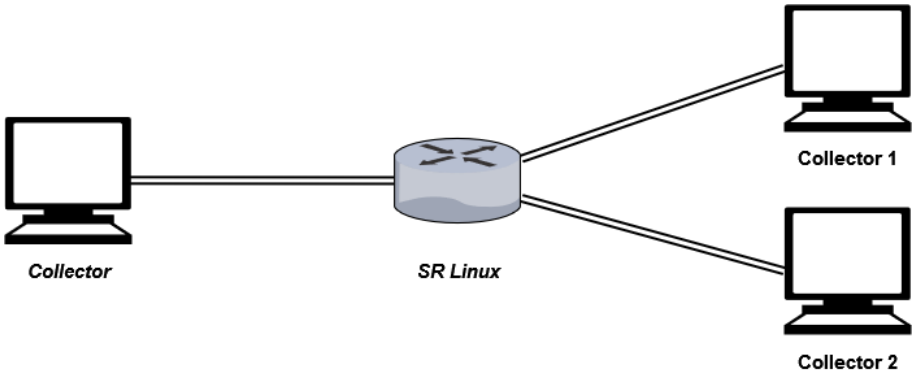

Figura 5.10: Cenário 3

Neste cenário, é verificado o correto funcionamento do agente para o envio de SNMP Traps para três *Collectors* diferentes. Estes *Collectors* estão configurados no SR Linux em três *Network Instances* diferentes. Assim, para verificar o correto funcionamento do agente para este cenário configurou-se o agente com evento descrito no Bloco de Código 5.3.

```
"monitoring_elements": [
        {
            "resource": "/system/json-rpc-server/",
            "parameter": "admin-state",
            "monitoring_condition": "",
            "resource_filter": [],
            "trigger_condition": "enable->disable",
            "trigger_message": "AdminStateDisable",
            "resolution_condition": "disable->enable",
            "resolution_message": "AdminStateEnable",
            "trap_oid": ".1.3.6.1.4.1.94.2500.3004"
        }
]
```

Bloco de Código 5.3: Elemento de Teste para o cenário 3

Este elemento permite enviar uma SNMP Trap aos *Collectors*, quando o servidor JSON-RPC for abaixo, isto é, o *admin-state* da interface passar de *enable* para *disable*. Além deste elemento de teste, também foram configurados os três *Collectors* nos *Targets* do agente.

```
"targets": [
        {
            "nw-instance": "srbase-mgmt",
            "address": "172.20.20.1",
            "community_string": "public"
        },
        {
            "nw-instance": "srbase-srl1-Collector1",
            "address": "10.1.0.15",
            "community_string": "toCollector1"
        },
        {
            "nw-instance": "srbase-srl1-Collector2",
            "address": "10.2.0.15",
            "community_string": "toCollector2"
        }
    ]
```

Bloco de Código 5.4: *Targets* para o cenário 3

Como se pode observar pelo Bloco de Código 5.4, os três *Collectors* estão configurados em três *Network Instances* diferentes. O primeiro *Target* da lista é igual ao dos outros cenários de simulação, utiliza a *network instance* de gestão para o envio da SNMP Trap para o *Collector*, este *Collector* é o *host* que corre o ContainerLab. Os outros dois *Targets* foram imagens Linux que foram introduzidas na topologia virtualizada pelo ContainerLab. Podemos também observar que os *Targets* estão configurados com diferentes *community strings*. Usaram-se diferentes valores para poder demonstrar o correto uso deste parâmetro pelo agente, aquando do envio das SNMP Traps.

Para captura dos pacotes nos dois *targets* introduzidos na topologia do ContainerLab, teve de executar-se o comando descrito no Bloco de Código 5.5 no *host*, identificando o *container* onde se pretende capturar os pacotes:

```
docker run --rm --net=container:clab-srl01-collector1 -v ~/Desktop/:/tcpdump
    kaazing/tcpdump
```

Bloco de Código 5.5: Comando para captura de pacotes dentro do Containerlab

O evento executado durante a simulação consistiu apenas em ligar o serviço JSON-RPC no dispositivo SR Linux em que corre o agente.

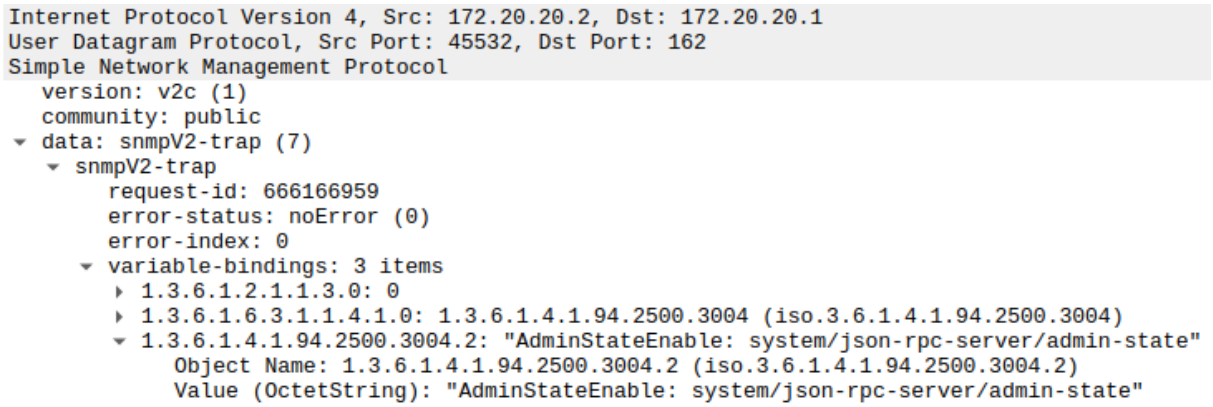

Figura 5.11: Cenário 3 - SNMP Trap enviado para o *Collector* associado ao *host*

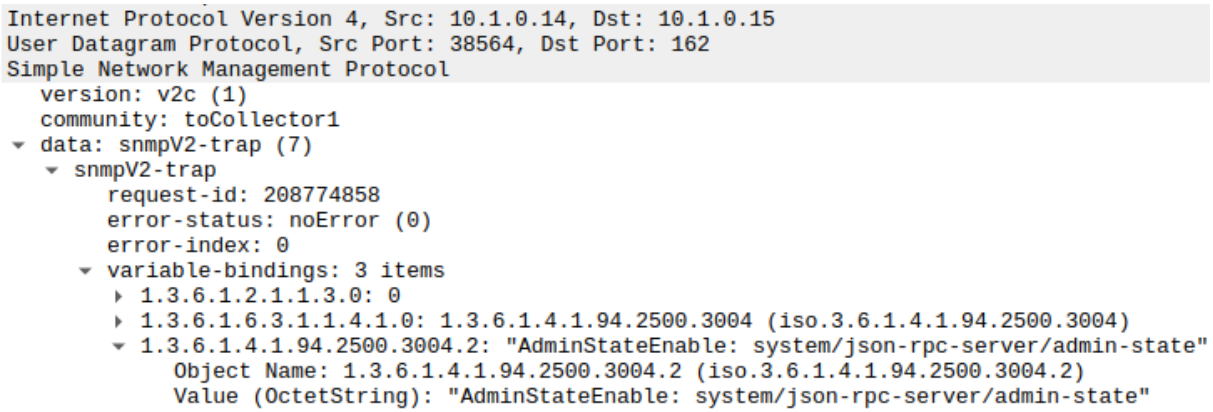

Figura 5.12: Cenário 3 - SNMP Trap enviado para o *Collector* 1

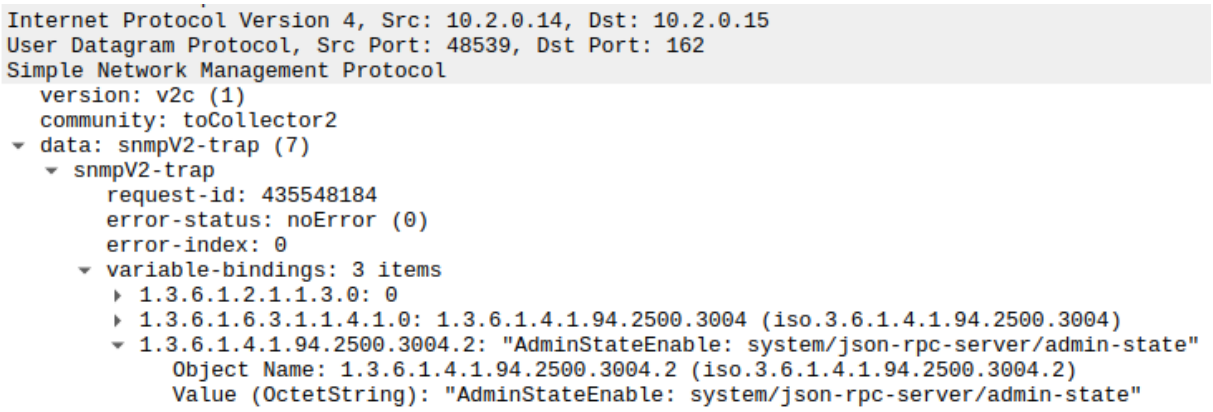

Figura 5.13: Cenário 3 - SNMP Trap enviado para o *Collector* 2

Como podemos observar pelas Figuras 5.11, 5.12 e 5.13, todos os *Collectors* receberam a SNMP Trap enviada. Verificamos o correto uso do endereço de IP de destino e da *community string* definida para cada um dos *Targets*. Tendo em conta a última linha de cada captura, é possível observar a descrição do evento ocorrido, que coincide com a simulação efetuada neste cenário.

```
snmp-agent {
    monitoring_elements {
        element /system/json-rpc-server/ {
            traps_generated 1
        }
    }
}
```

Figura 5.14: Cenário 3 - *Datastore State*

Apesar de terem sido enviadas três SNMP Traps para os três *Collectors* distintos, apenas foi contabilizado uma vez o envio neste elemento, como se pretendia.

Segundo a avaliação proposta na secção 5.1, neste cenário, o agente cumpre com alguns dos pontos propostos e que são percetíveis para este cenário, pois faz: **2.A)** o correto uso dos endereços de IP definidos e o envio das SNMP Traps através das *Network Instances* corretas; **2.B)** o correto uso da *community string* na camada SNMP do pacote enviado; **3)** a correta atualização da *datastore state* com a informação pretendida.

5.5 Conclusão

Neste Capítulo demonstrou-se a verificação do correto funcionamento do agente, tendo-se executado três cenários distintos de simulação e demonstrado, para cada cenário, como foi avaliado o agente, tendo em conta as funcionalidades. Além destes cenários, outras experiências e simulações foram executadas, no decorrer do desenvolvimento do agente. O Apêndice B apresenta uma compilação de elementos, usados para verificação do agente, além daqueles que foram tidos em conta nestes cenários de simulação.

6

Conclusão

Conteúdo

6.1 Conclusões

O objetivo principal desta dissertação foi o desenvolvimento de um agente SNMP para o sistema operativo *SR Linux*. Este agente serve como interface entre duas tecnologias de monitorização das redes gNMI e SNMP.

Com isto em mente, primeiramente, foi feita a pesquisa sobre o estado de arte na área da monitorização das redes. Uma área vasta e cheia de conceitos e protocolos, desenvolvidos ao longo dos anos, de forma a colmatar as necessidades de cada época em que foram desenvolvidos, como no caso do SNMP, NETCONF, RESTCONF, gNMI.

Além do estudo destes protocolos, foi estudado também o conceito de *Network Telemetry*, um conceito relativamente recente na área e que tenta enquadrar estas tecnologias e conceitos numa *framework*, para uma mais fácil interpretação do fluxo e consumos dos dados de telemetria nas topologias de rede.

Com base nos conceitos e protocolos estudados, estabeleceu-se um fluxograma de execução do agente. Este fluxograma de execução teria de ter em conta a integração com o NDK do SR Linux e, por isso, um estudo sobre o próprio sistema SR Linux foi tido em conta no presente documento.

No desenho do agente, tentou-se que este fosse o mais genérico possível e, desta forma, é possível adicionar eventos no agente, de uma forma dinâmica. A outra solução passaria por fixar que elementos do sistema é que o agente estaria à escuta de eventos e só para esses eventos é que o agente iria enviar as SNMP Traps. Quanto ao desempenho do agente, verificou-se que este cumpria todos os requisitos funcionais propostos para esta dissertação.

Em suma, esta dissertação permitiu o estudo e desenvolvimento de uma aplicação (agente), num sistema operativo estruturado e pensado com tecnologias atuais e virado para a modularidade do próprio sistema, como é o uso de MD-CLI e dos modelos de dados YANG, como forma de ver as diferentes funcionalidades do sistema, mas que precisa de suportar tecnologias mais antigas, neste caso SNMP, para uma correta integração em sistemas de gestão já existentes.

6.2 Trabalho futuro

Os próximos passos no desenvolvimento deste agente deveriam incidir principalmente na resolução de alguns *bugs* que o agente ainda apresenta em situações muito específicas e numa funcionalidade que não foi implementada, que é uma separação lógica dos eventos associados a cada *Target*. Neste caso, cada *Target* configurado no agente estaria à escuta de apenas um número predefinido de eventos e não de todos os eventos como acontece atualmente. À parte disto, o agente deveria ser posto à prova, em contextos reais, para garantir a correta execução do mesmo, neste tipo de contextos.

Bibliografia

[1] Data center fabric. https://www.nokia.com/networks/data-center/data-center-fabric/ (acedido a 2022-10-12).

[2] Cisco. Network Management Reference Architecture. https://www.cisco.com/en/US/technologies/collateral/tk869/tk769/white_paper_c11-453503.pdf (acedido a 2022-04-19).

[3] Entendendo a itil v3. https://freshservice.com/br/itil/itil-v3// (acedido a 2022-04-19).

[4] What is itil? itil process and itil framework tutorial. https://www.guru99.com/itil-framework-process.html (acedido a 2022-04-19).

[5] William Stallings. *SNMP, SNMPv2, and RMON Practical Network Management*. Addison-Wesley, 1996.

[6] Karel Hromek. *Analyser of Log Entries from Central Logging Server*. PhD thesis, Masaryk University Faculty of Informatics, 2009.

[7] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (netconf). RFC 6241, RFC Editor, Junho 2011. http://www.rfc-editor.org/rfc/rfc6241.txt (acedido a 2022-10-12).

[8] Network automation using yang models across xe, xr, & nx. https://yang-prog-lab.ciscolive.com/pod/5/mdp_foundations/introduction_to_netconf (acedido a 2022-10-12).

[9] Richard Janík. *Implementation of the protocol RESTCONF in GNU/Linux*. PhD thesis, Masaryk University Faculty of Informatics, 2015.

[10] Attila Fábián. Monitoring modern networks with network telemetry. Bachelor's thesis, Budapest University of Technology and Economics, 2019.

[11] Mark Burgess and Thomas Schaaf. Integrating cfengine, itil and enterprise processes, 2008.

[12] C. Lonvick. The bsd syslog protocol. RFC 3164, RFC Editor, Agosto 2001. http://www.rfc-editor.org/rfc/rfc3164.txt (acedido a 2022-10-12).

[13] H. Song, F. Qin, P. Martinez-Julia, L. Ciavaglia, and A. Wang. Network telemetry framework. RFC 9232, RFC Editor, Maio 2022. https://datatracker.ietf.org/doc/rfc9232/ (acedido a 2022-10-12).

[14] M. Bjorklund. The yang 1.1 data modeling language. RFC 7950, RFC Editor, Agosto 2016. http://www.rfc-editor.org/rfc/rfc7950.txt (acedido a 2022-10-12).

[15] Karneliuk. Model-driven command line interface (md-cli) in nokia sr os 16.0. https://karneliuk.com/2018/06/model-driven-command-line-interface-md-cli-in-nokia-sr-os-16-0/ (acedido a 2022-03-12).

[16] Guilherme Cardoso. Gascpt/snmp_agent: This is the repository for the snmp agent for sr linux. https://github.com/GascPT/snmp_agent (acedido a 2022-10-12).

[17] Edmund Wong. Network monitoring fundamentals and standards. https://www.cse.wustl.edu/~jain/cis788-97/ftp/net_monitoring/#[Stallings-Book]. (acedido a 2022-04-19).

[18] Tech-faq - fcaps. https://www.tech-faq.com/fcaps.html. (acedido a 2022-04-19).

[19] Pankaj Goyal, Rao Mikkilineni, and Murthy Ganti. Fcaps in the business services fabric model. In *2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pages 45–51, 2009.

[20] Kenneth Osigwelem, A.C. Akukwe, Agbakwuru Alphons, and Elochukwu Ukwandu. The use of fcaps and itil in managing the network of a medium to large public sector organisation. *Asian Journal of Information Technology*, 10:240–248, Junho 2011.

[21] Jack Probst. Anatomy of a service a practical guide to defining it services. page 4, Setembro 2009. https://www.alaska.edu/files/oit/ITSM_Program/Anatomy-of-a-Service-White-Paper.pdf (acedido a 2022-04-19).

[22] Continual service improvement processes: Itil intermediate csi. https://www.simplilearn.com/tutorials/itil-tutorial/continual-service-improvement-processess (acedido a 2022-04-19).

[23] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. Simple network management protocol (snmp). STD 15, RFC Editor, Maio 1990. http://www.rfc-editor.org/rfc/rfc1157.txt (acedido a 2022-10-12).

[24] J. Schonwalder, A. Pras, and J.-P. Martin-Flatin. On the future of internet management technologies. *IEEE Communications Magazine*, 41(10):90–97, 2003.

[25] Abubucker Shaffi and Mohaned Al-Obaidy. Managing network components using snmp. *International Journal of Scientific Knowledge*, pages 11–18, 04 2013.

[26] DPS Telecom. What's the Default SNMP Port Number? Is SNMP TCP or UDP? https://www.dpstele.com/snmp/transport-requirements-udp-tcp.php (acedido a 2021-11-12).

[27] Keith McCloghrie, David Perkins, and Juergen Schoenwaelder. Structure of management information version 2 (smiv2). STD 58, RFC Editor, Abril 1999. http://www.rfc-editor.org/rfc/rfc2578.txt (acedido a 2022-10-12).

[28] Marshall T. Rose and Keith McCloghrie. Structure and identification of management information for tcp/ip-based internets. STD 16, RFC Editor, Maio 1990. http://www.rfc-editor.org/rfc/rfc1155.txt (acedido a 2022-10-12).

[29] Keith McCloghrie and Marshall T. Rose. Management information base for network management of tcp/ip-based internets:mib-ii. STD 17, RFC Editor, Março 1991. http://www.rfc-editor.org/rfc/rfc1213.txt (acedido a 2022-10-12).

[30] R. Presuhn. Management information base (mib) for the simple network management protocol (snmp). STD 62, RFC Editor, Dezembro 2002. http://www.rfc-editor.org/rfc/rfc3418.txt (acedido a 2022-10-12).

[31] D. Harrington, R. Presuhn, and B. Wijnen. An architecture for describing simple network management protocol (snmp) management frameworks. STD 62, RFC Editor, Dezembro 2002. http://www.rfc-editor.org/rfc/rfc3411.txt (acedido a 2022-10-12).

[32] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Introduction to version 2 of the internet-standard network management framework. RFC 1441, RFC Editor, April 1993. http://www.rfc-editor.org/rfc/rfc1441.txt (acedido a 2022-10-12).

[33] U. Blumenthal and B. Wijnen. User-based security model (usm) for version 3 of the simple network management protocol (snmpv3). RFC 2574, RFC Editor, Abril 1999. http://www.rfc-editor.org/rfc/rfc2574.txt (acedido a 2022-10-12).

[34] James Yu and Imad Al Ajarmeh. An empirical study of the netconf protocol. In *2010 Sixth International Conference on Networking and Services*, pages 253–258, 2010.

[35] J. Schoenwaelder. Simple network management protocol over transmission control protocol transport mapping. RFC 3430, RFC Editor, Dezembro 2002. http://www.rfc-editor.org/rfc/rfc3430.txt (acedido a 2022-10-12).

[36] Mi-Jung Choi, Hyoun-Mi Choi, J.W. Hong, and Hong-Taek Ju. Xml-based configuration management for ip network devices. *IEEE Communications Magazine*, 42(7):84–91, 2004.

[37] R. Gerhards. The syslog protocol. RFC 5424, RFC Editor, Março 2009. http://www.rfc-editor.org/rfc/rfc5424.txt (acedido a 2022-10-12).

[38] Alexander S. Gillis. What is screen scraping and how does it work?, Fevereiro 2020. https://www.techtarget.com/searchdatacenter/definition/screen-scraping (acedido a 2022-09-12).

[39] R. Enns. Netconf configuration protocol. RFC 4741, RFC Editor, Dezembro 2006. http://www.rfc-editor.org/rfc/rfc4741.txt (acedido a 2022-10-12).

[40] M. Wasserman and T. Goddard. Using the netconf configuration protocol over secure shell (ssh). RFC 4742, RFC Editor, Dezembro 2006. http://www.rfc-editor.org/rfc/rfc4742.txt (acedido a 2022-10-12).

[41] E. Lear and K. Crozier. Using the netconf protocol over the blocks extensible exchange protocol (beep). RFC 4744, RFC Editor, Dezembro 2006. http://www.rfc-editor.org/rfc/rfc4744.txt (acedido a 2022-10-12).

[42] T. Goddard. Using netconf over the simple object access protocol (soap). RFC 4743, RFC Editor, Dezembro 2006. http://www.rfc-editor.org/rfc/rfc4743.txt (acedido a 2022-10-12).

[43] M. Badra. Netconf over transport layer security (tls). RFC 5539, RFC Editor, Maio 2009. http://www.rfc-editor.org/rfc/rfc5539.txt (acedido a 2022-10-12).

[44] A. Bierman, M. Bjorklund, and K. Watsen. Restconf protocol. RFC 8040, RFC Editor, Janeiro 2017. http://www.rfc-editor.org/rfc/rfc8040.txt (acedido a 2022-10-12).

[45] Openconfig. grpc network management interface. https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md, 2018.

[46] Rob Shakir, Anees Shaikh, Paul Borman, Marcus Hines, Carl Lebsack, and Chris Morrow. gRPC Network Management Interface (gNMI). Internet-Draft draft-openconfig-rtgwg-gnmi-spec-01, Internet Engineering Task Force, March 2018. https://datatracker.ietf.org/doc/draft-openconfig-rtgwg-gnmi-spec/01/ (acedido a 2022-10-12).

[47] 2021 gRPC Authors. About gRPC. https://grpc.io/about/ (acedido a 2021-11-24).

[48] OpenConfig. gnmi - grpc network management interface. https://github.com/openconfig/gnmi/tree/master/proto/gnmi (acedido a 2022-10-12).

[49] Kasun Indrasiri and Danesh Kuruppu. *GRPC: Up and running: Building cloud native applications with go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[50] JSON-RPC google group. JSON-RPC 1.0 Specification. https://www.jsonrpc.org/specification_v1 (acedido a 2021-11-16).

[51] JSON-RPC google group. JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification (acedido a 2021-11-16).

[52] Chris Ward. JSON RPC API. https://eth.wiki/json-rpc/API (acedido a 2021-11-16).

[53] NOKIA. Json interface. https://documentation.nokia.com/srlinux/SR_Linux_HTML_R21-11/SysMgmt_Guide/json-interface.html#ai9ersv4qj. (acedido a 2022-01-06).

[54] M. Bjorklund. Yang - a data modeling language for the network configuration protocol (netconf). RFC 6020, RFC Editor, Outubro 2010. http://www.rfc-editor.org/rfc/rfc6020.txt (acedido a 2022-10-12).

[55] J. Schoenwaelder. Overview of the 2002 iab network management workshop. RFC 3535, RFC Editor, Maio 2003. http://www.rfc-editor.org/rfc/rfc3535.txt (acedido a 2022-06-10).

[56] P. Shafer. An architecture for network management using netconf and yang. RFC 6244, RFC Editor, Junho 2011. http://www.rfc-editor.org/rfc/rfc6244.txt (acedido a 2022-10-12).

[57] Benoit Claise, Joe Clarke, and Jan Lindblad. *Network programmability with YANG: the structure of network automation with YANG, NETCONF, RESTCONF, and gNMI*. Addison-Wesley, 2019.

[58] L. Lhotka. Json encoding of data modeled with yang. RFC 7951, RFC Editor, Agosto 2016. http://www.rfc-editor.org/rfc/rfc7951.txt (acedido a 2022-10-12).

[59] Guangying Zheng, Tianran Zhou, and Alexander Clemm. Udp based publication channel for streaming telemetry. Internet-Draft draft-ietf-netconf-udp-pub-channel-05, IETF Secretariat, Março 2019. https://www.ietf.org/archive/id/draft-ietf-netconf-udp-pub-channel-05.txt (acedido a 2022-10-12).

[60] Nokia. Ndk architecture. https://learn.srlinux.dev/ndk/guide/architecture/. (acedido a 2022-01-10).

[61] Nokia. Network instances. https://learn.srlinux.dev/kb/netwinstance/ (acedido a 2022-10-12).

[62] Nokia. Management interfaces. https://learn.srlinux.dev/kb/mgmt/ (acedido a 2022-10-12).

ContainerLab

Neste Apêndice são descritos alguns dos comandos básicos, para correta utilização da ferramenta de virtualização ContainerLab.

A.1 Instalação

O ContainerLab é uma ferramenta de virtualização baseada em sistemas Debian. De modo a que seja possível instalar o ContainerLab neste tipo de sistemas é necessário que:

- O utilizador tenha privilégios *sudo*.
- Docker instalado.

À parte destes requisitos, para instalar corretamente o ContainerLab basta correr um dos dois comandos descrito no Bloco de Código A.1.

```
1 bash -c "$(curl -sL https://get.containerlab.dev)"
2 bash -c "$(wget -qO - https://get.containerlab.dev)"
```

Bloco de Código A.1: Comandos para Instalar o ContainerLab

A.2 Virtualização da topologia

Para executar o Containerlab é necessário ter um ficheiro que descreva a topologia de rede. Este ficheiro tem o formato YAML. No Bloco de Código A.2 é possível observar um exemplo da estrutura deste ficheiro. No ficheiro é possível configurar o *name* usado na topologia, os *nodes* da topologia e que tipo de nós são.

```
1  name: srl02
2
3  topology:
4    nodes:
5      srl1:
6        kind: srl
7        image: ghcr.io/nokia/srlinux
8      srl2:
9        kind: srl
10       image: ghcr.io/nokia/srlinux
11
12   links:
13     - endpoints: ["srl1:e1-1", "srl2:e1-1"]
```

Bloco de Código A.2: Exemplo do ficheiro YAML com uma topologia de rede

Para instanciar os dispositivos de rede que estão descritos no ficheiro YAML é necessário correr o comando descrito no Bloco de Código A.3:

```
sudo containerlab deploy -t example.yml
```

Bloco de Código A.3: Comando para *deployment* da topologia de rede

Este comando faz o *deployment* dos dispositivos de rede presentes na topologia. Estes dispositivos são contentores virtuais *docker*.

A.3 Guardar a topologia

Para guardar configurações que tenham sido executadas nos dispositivos de rede no Containerlab é necessário executar o comando descrito no Bloco de Código A.4:

```
sudo containerlab save -t example.yml
```

Bloco de Código A.4: Comando para guardar a topologia de rede

A.4 Destruir a topologia

Para destruir a topologia do Containerlab, isto é, os *containers* é necessário executar o comando descrito no Bloco de Código A.5:

```
sudo containerlab destroy -t example.yml
```

Bloco de Código A.5: Comando para destruir a topologia de rede

A.5 Visualização da topologia

É possível ter uma visualização gráfica da topologia descrita no ficheiro, através do uso do comando descrito no Bloco de Código A.6:

```
sudo containerlab graph -t example.yml
```

Bloco de Código A.6: Comando para visualizar a topologia de rede

B

Elementos Testados

Neste Apêndice são descritos todos os elementos de teste usados durante o desenvolvimento do agente.

```
1   {
2       "resource": "interface[name=ethernet-1/2]/",
3       "parameter": "oper-state",
4       "monitoring_condition": "",
5       "resource_filter": [],
6       "trigger_condition": "up->down",
7       "trigger_message": "oper-down-reason",
8       "resolution_condition":"down->up",
9       "resolution_message":"operUP",
10      "trap_oid":".1.3.6.1.4.1.94.2500.3001"
11  }
```

Bloco de Código B.1: Elemento Testado 1

Este elemento monitoriza o estado da interface ethernet-1/2, sem condição de monitorização e sem *resource_filter*, pois têm uma *key* específica de uma interface. Quando o *oper-state* passar do estado *up* para *down* acontecerá um SNMP Trap a avisar do evento, quando o *oper-state* passar do estado *down* para *up* acontecerá também uma SNMP Trap, neste caso, para alertar que houve resolução do evento anterior.

```
1   {
2       "resource": "interface[name=*]/",
```

```
3           "parameter": "oper-state",
4           "monitoring_condition": "admin-state=enable",
5           "resource_filter": [
6               "ethernet*",
7               "lag*",
8               "mgmt*"
9           ],
10          "trigger_condition": "up->down",
11          "trigger_message": "oper-down-reason",
12          "resolution_condition":"down->up",
13          "resolution_message":"operUP",
14          "trap_oid":".1.3.6.1.4.1.94.2500.3001"
15      }
```

Bloco de Código B.2: Elemento Testado 2

Este elemento monitoriza o estado de todas as interfaces do dispositivo, com condição de monitorização de só querermos as interfaces com *admin-state=enable* e com *resource_filter*. Quando o *oper-state* passar do estado *up* para *down* acontecerá um SNMP Trap a avisar do evento; quando o *oper-state* passar do estado *down* para *up* acontecerá também uma SNMP Trap, neste caso, para alertar que houve resolução do evento anterior.

```
1       {
2           "resource": "/network-instance[name=*]/protocols/bgp/neighbor[peer-address=*]/",
3           "parameter": "session-state",
4           "monitoring_condition": "admin-state=enable",
5           "resource_filter": [
6
7           ],
8           "trigger_condition": "!=established",
9           "trigger_message": "bgpNeighborClosedTCPConn",
10          "resolution_condition":"==established",
11          "resolution_message":"bgpNeighborEstablished",
12          "trap_oid":".1.3.6.1.4.1.94.2500.3002"
13      }
```

Bloco de Código B.3: Elemento Testado 3

Este elemento monitoriza o estado das sessões BGP entre os *peers* em *network instances* diferentes, com condição de monitorização de só querermos as *network instances* com *admin-state=enable* e sem *resource_filter*. Quando o *session-state* passar do estado *established* para outro qualquer acontecerá um SNMP Trap a avisar do evento, quando o *session-state* passar de um estado qualquer para *established* acontecerá também uma SNMP Trap, neste caso, para avisar que houve resolução do evento anterior.

```
1       {
2           "resource": "/system/lldp/interface[name=*]/",
3           "parameter": "oper-state",
4           "monitoring_condition": "admin-state=enable",
5           "resource_filter": [
6
7           ],
8           "trigger_condition": "up->down",
9           "trigger_message": "LLDPinterfaceDown",
10          "resolution_condition":"down->up",
11          "resolution_message":"LLDPinterfaceUp",
12          "trap_oid":".1.3.6.1.4.1.94.2500.3003"
```

```
13      }
```

Bloco de Código B.4: Elemento Testado 4

Este elemento monitoriza o estado das interfaces LLDP do sistema, com condição de monitorização de só querermos as interfaces com *admin-state=enable* e sem *resource_filter*. Quando o *oper-state* passar do estado *up* para *down* acontecerá um SNMP Trap a avisar do evento, quando o *oper-state* passar do estado *down* para *up* acontecerá também uma SNMP Trap, neste caso a avisar que houve resolução para o evento anterior.

```
1       {
2           "resource": "/system/json-rpc-server/",
3           "parameter": "admin-state",
4           "monitoring_condition": "",
5           "resource_filter": [],
6           "trigger_condition": "enable->disable",
7           "trigger_message": "ADMIN STATE DISABLE",
8           "resolution_condition": "disable->enable",
9           "resolution_message": "ADMIN STATE ENABLE",
10          "trap_oid": ".1.3.6.1.4.1.94.2500.3004"
11      }
```

Bloco de Código B.5: Elemento Testado 5

Este elemento monitoriza o estado do servidor JSON-RPC do sistema, sem condição de monitorização e sem *resource_filter*. Quando o *admin-state* passar do estado *enable* para *disable* acontecerá um SNMP Trap a avisar do evento, quando o *oper-state* passar do estado *disable* para *enable* acontecerá também uma SNMP Trap, neste caso a avisar que houve resolução para o evento anterior.

```
1       {
2           "resource": "/system/app-management/application[name=*]/",
3           "parameter": "state",
4           "monitoring_condition": "",
5           "resource_filter": [],
6           "trigger_condition": "==exited",
7           "trigger_message": "Application Down",
8           "resolution_condition": "==running",
9           "resolution_message": "Application Up",
10          "trap_oid": ".1.3.6.1.4.1.94.2500.3005"
11      }
```

Bloco de Código B.6: Elemento Testado 6

Este elemento monitoriza o estado de todas as aplicações que estão em execução no sistema, sem condição de monitorização e sem *resource_filter*. Quando o *state* for igual a *exited* acontecerá um SNMP Trap a avisar do evento, quando o *state* passar de um estado qualquer para *running* acontecerá também uma SNMP Trap, neste caso a avisar que houve resolução para o evento anterior.

C

Implementação

Neste Apêndice é descrita a implementação do agente, mostrando o código-fonte em Python dos componentes descritos no Fluxograma, representado na Figura 4.3.

O código está dividido essencialmente em três ficheiros distintos:

- *snmp_agent.py* — Ficheiro principal do agente;
- *target.py* — Ficheiro que define a classe Target do agente;
- *element.py* — Ficheiro que define a classe Element do agente;

C.1 Registo do Agente

Run() — Método responsável por fazer o registo do agente no SDK. Este método também é responsável por lançar as *threads* **send_keep_alive()**, **subscribe_thread()**. É neste método que o agente executará o loop infinito à espera da receção de eventos gNMI. Este método encontra-se no ficheiro *snmp_agent.py*.

```
            # GLOBAL VARIABLES
agent_name ='snmp_agent'
FILENAME = 'input_elements'
channel = grpc.insecure_channel('127.0.0.1:50053')
metadata = [('agent_name', agent_name)]
stub = sdk_service_pb2_grpc.SdkMgrServiceStub(channel)
# GNMI Server
host = ('unix:///opt/srlinux/var/run/sr_gnmi_server', 57400)
# Queue with the entries of subscribe function
queue = queue.Queue()
# Logger Class
log = MyLogger("Logger")
# Main Variables of the agent
global_paths = []
targets = []
elements = []

def Run():
    response = stub.AgentRegister(request=sdk_service_pb2.AgentRegistrationRequest(agent_liveliness=5), metadata=metadata)
    log.info(f"Registration response : {response.status}")
    # Send Keep Alives
    th = threading.Thread(target=send_keep_alive)
    th.start()
    #Subscribe Notifications
    request=sdk_service_pb2.NotificationRegisterRequest(op=sdk_service_pb2.NotificationRegisterRequest.Create)
    create_subscription_response = stub.NotificationRegister(request=request, metadata=metadata)
    if create_subscription_response.status == sdk_common_pb2.SdkMgrStatus.Value("kSdkMgrFailed"):
        log.info("Notification register Failed")

    stream_id = create_subscription_response.stream_id
    log.info(f"Create subscription response received. Stream_id : {stream_id}")
    Subscribe_Notifications(stream_id)

    if os.path.exists(FILENAME):
        addStatusToMemory(None,FILENAME)
        time.sleep(0.5)
    notificationStreamThread = threading.Thread(target=NotificationStreamThread, args=(stream_id,))
    notificationStreamThread.start()
    # Add Global Paths from Elements
    addGlobalPaths()
    # START OF GNMI SUBSCRIBE THREAD
    x = threading.Thread(target=subscribe_thread, args=(global_paths,))
    x.start()
    # Infinite Loop to send SNMP traps
    while True:
        while not queue.empty():
            entry = queue.get()
            entry = entry['update']['update']
            #log.info(f"{entry}  :::::  QUEUE SIZE --> {queue.qsize()}\n ")
            processEntry(entry)
    sys.exit()
    return True
```

Bloco de Código C.1: Registo do Agente

C.2 Leitura do ficheiro de configuração "input_elements.json"

addStatusToMemory() — este método tem duas partes, a primeira parte será detalhada na secção C.4. A segunda parte do método trata da leitura do ficheiro de configuração e a introdução dos dados nas estruturas respetivas do agente. Executa dois métodos, de forma a adicionar os *Elements* e os *Targets* lidos do ficheiro na *datastore state*, o método **addElementsToTelemetry()** para os *Elements* e o método **addTargetsToTelemetry()** para os *Targets*.

```
def addStatusToMemory(obj, filename = None):
    global targets, elements, gnmi_credentials
    ...
    # From File
    else:
        try:
            with open(filename) as f:
                file_data_as_json = json.load(f)

                # Read GNMI Credentials
                aux = file_data_as_json['gnmi_credentials']
                gnmi_credentials = Credentials(aux['user'],aux['password'])


                # Add Elements to the global variable
                for element in file_data_as_json['monitoring_elements']:
                    resource = addBrackets(element['resource'])

                    # Mandatory parameters
                    parameter = element['parameter']
                    trigger_condition = element['trigger_condition']
                    trap_oid = element['trap_oid']

                    # Optional parameters
                    if "resource_filter" in element:
                        resource_filter = element['resource_filter']
                    else:
                        resource_filter = []

                    if "monitoring_condition" in element:
                        monitoring_condition = element['monitoring_condition']
                    else:
                        monitoring_condition = ""

                    if "trigger_message" in element:
                        trigger_message = element['trigger_message']
                    else:
                        trigger_message = ""

                    if "resolution_condition" in element:
                        resolution_condition = element['resolution_condition']
                    else:
                        resolution_condition = ""

                    if "resolution_message" in element:
                        resolution_message = element['resolution_message']
                    else:
                        resolution_message = ""

                    e = Element(resource,
                                parameter,
                                monitoring_condition,
                                resource_filter,
                                trigger_condition,
                                trigger_message,
                                resolution_condition,
                                resolution_message,
                                trap_oid,
                                False,
                                "")

                    # Add element to monitoring to the list of elements
                    elements.append(e)

                # Add to Telemetry
                addElementsToTelemetry()

                # Add targets to the global variable
                for target in file_data_as_json['targets']:
                    flag = False
```

```
71                      for t in targets:
72                          if target['address'] == t.get_address() and target['nw-instance'] == t.get_nw():
73                              flag = True
74
75                      if not flag:
76                          nw_instance = target['nw-instance']
77                          address = target['address']
78                          targets.append(Target(nw_instance,address))
79
80                  # Add Targets to State
81                  if not len(targets) == 0:
82                      addTargetsToTelemetry()
83
84                  # Add to Config
85                  addStatusToConfigDataStore()
86
87
88          except Exception as e:
89              logging.info(f"Exception caught while reading file :: {e}")
90              #Set programed status as false
91              return False
```

Bloco de Código C.2: Leitura do ficheiro de configuração

C.3 Registo de Notificações

Subscribe_Notifications() — Método responsável por chamar o método **subscribe()**, para registo das notificações de *config*, se o agente quisesse escutar outros serviços seria aqui que seriam configurados;
Subscribe() — Método responsável pela subscrição no SDK das notificações de *config*;
NotificationStreamThread() — *Thread* que escuta as notificações recebidas pelo SDK, na variável *stream_response*, a função entra num loop infinito, na linha 26;
Handle_Notification() — Quando o agente recebe uma notificação, é este método que faz o processamento dessa notificação, em três operações distintas. Pode haver criação de elementos no agente, alteração de elementos ou remoção de elementos, conforme a operação serão executados os métodos respetivos;

```
1   def Subscribe_Notifications(stream_id):
2       if not stream_id:
3           log.info("Stream ID not sent.")
4           return False
5
6       ##Subscribe to Config Notifications - configs added by the snmp-agent
7       Subscribe(stream_id)
8
9   def Subscribe(stream_id):
10      op = sdk_service_pb2.NotificationRegisterRequest.AddSubscription
11      entry = config_service_pb2.ConfigSubscriptionRequest()
12      request = sdk_service_pb2.NotificationRegisterRequest(op=op, stream_id=stream_id, config=entry)
13
14      subscription_response = stub.NotificationRegister(request=request, metadata=metadata)
15
16      if subscription_response.status == sdk_common_pb2.SdkMgrStatus.Value("kSdkMgrFailed"):
17          log.info("Subscription Config register Failed")
18
19      log.info('Status of subscription response for config :: {}'.format(subscription_response.status))
20
21  def NotificationStreamThread(stream_id):
```

```
22      sub_stub = sdk_service_pb2_grpc.SdkNotificationServiceStub(channel)
23      stream_request = sdk_service_pb2.NotificationStreamRequest(stream_id=stream_id)
24      stream_response = sub_stub.NotificationStream(stream_request, metadata=metadata)
25      time.sleep(2)
26      for r in stream_response:
27          #log.info(r.notification)
28          for obj in r.notification:
29              Handle_Notification(obj)
30
31  def Handle_Notification(obj) -> bool:
32      if obj.HasField('config') and obj.config.key.js_path != ".commit.end":
33          if obj.config.op == 0: # Add Config Operation
34              if "snmp_agent" in obj.config.key.js_path:
35                  addStatusToMemory(obj)
36          elif obj.config.op == 1: # Change Configuration
37              log.info("Change HANDLER") # TODO
38              changeStatusOfMemory(obj)
39          elif obj.config.op == 2: # Delete Configuration
40              delStatusofMemory(obj)
41          else:
42              log.info("SOMETHING GONE WRONG")
43      #always return
44      return False
```

Bloco de Código C.3: Registo de Notificações

C.4 Leitura das configurações guardadas na IDB

addStatusToMemory() — este método tem duas partes, sendo que a segunda parte já foi detalhada na secção C.2. A primeira parte, que é a que interessa para esta componente, é responsável pela introdução dos dados provenientes de notificações nas estruturas do agente, esta função também é responsável pela atualização do ficheiro de configuração “input_elements.json” das alterações existentes. Este método encontra-se no ficheiro *snmp_agent.py*.

```
1   def addStatusToMemory(obj, filename = None):
2       global targets, elements, gnmi_credentials
3       # From Notification
4       if filename == None:
5           #Check if are target config
6           if obj.config.key.js_path == ".snmp_agent.targets.target":
7               #Check if exits any config
8               if not obj.config.data.json == "{\n}\n":
9                   notification_targets = json.loads(obj.config.data.json)
10                  # One notification for entry
11                  address = obj.config.key.keys[0]
12                  nw_instance = notification_targets['target']['network_instance']['value']
13
14                  # Verify if exists in targets
15                  flag = False
16                  for t in targets:
17                      if t.get_address() == address and t.get_nw() == nw_instance:
18                          flag = True
19
20                  #Add to Telemetry and targets element
21                  if not flag:
22                      targets.append(Target(nw_instance,address))
23                      addTargetsToTelemetry()
24
25                  # Add to the File
26                  with open(FILENAME,"r+") as f:
27                      file_data = json.load(f)
28                      aux_targets = []
29                      targets_original = []
```

```
                for target in file_data['targets']:
                    aux_targets.append(target)

                #Diff between global state targets and targets written on the file
                for t in targets:
                    targets_original.append(t.get_JSON())

                diff_targets = diffTwoJSONObjects(targets_original,aux_targets)

                if not len(diff_targets) == 0:
                    for t in diff_targets:
                        file_data['targets'].append(t)

                    f.seek(0)
                    f.write(json.dumps(file_data, indent=4))
                    f.truncate()

        # Check if are configuration of a element
        elif obj.config.key.js_path == ".snmp_agent.monitoring_elements.element":
            #Check if exits any config
            if not obj.config.data.json == "{\n}\n":
                element_json = json.loads(obj.config.data.json)['element']
                # Desiralization of the elements into variables
                resource = obj.config.key.keys[0]
                # Mandatory parameters
                parameter = element_json['parameter']['value']
                trigger_condition = element_json['trigger_condition']['value']
                trap_oid = element_json['trap_oid']['value']

                # Optional parameters
                if "resource_filter" in element_json:
                    resource_filter = [x['value'] for x in element_json['resource_filter']]
                else:
                    resource_filter = []

                if "monitoring_condition" in element_json:
                    monitoring_condition = element_json['monitoring_condition']['value']
                else:
                    monitoring_condition = ""

                if "trigger_message" in element_json:
                    trigger_message = element_json['trigger_message']['value']
                else:
                    trigger_message = ""

                if "resolution_condition" in element_json:
                    resolution_condition = element_json['resolution_condition']['value']
                else:
                    resolution_condition = ""

                if "resolution_message" in element_json:
                    resolution_message = element_json['resolution_message']['value']
                else:
                    resolution_message = ""

                element = Element(resource,
                            parameter,
                            monitoring_condition,
                            resource_filter,
                            trigger_condition,
                            trigger_message,
                            resolution_condition,
                            resolution_message,
                            trap_oid,
                            False,
                            "")

                # Verify if exists in elements
                flag = False
                for e in elements:
```

```
100                     if e.getResource() == resource: # Verify if key list
101                         flag = True
102
103                 # Add element to monitoring to the list of elements
104                 if not flag:
105                     elements.append(element)
106                     # Add to Telemetry
107                     addElementsToTelemetry()
108
109                 # Add to the File
110                 with open(FILENAME,"r+") as f:
111                     file_data = json.load(f)
112                     aux_elements = []
113                     elements_original = []
114
115                     for element in file_data['monitoring_elements']:
116                         aux_elements.append(element)
117
118                     #Diff between global state elements and elements written on the file
119                     for e in elements:
120                         elements_original.append(e.getJSON())
121
122                     diff_elements = diffTwoJSONObjects(elements_original,aux_elements)
123
124                     if not len(diff_elements) == 0:
125                         for e in diff_elements:
126                             file_data['monitoring_elements'].append(e)
127
128                         f.seek(0)
129                         f.write(json.dumps(file_data, indent=4))
130                         f.truncate()
131         ...
```

Bloco de Código C.4: Leitura das configurações guardadas na datastore *config*

C.5 Subscrição dos Elementos

subscribe_thread() — Método responsável pela subscrição do estado dos elementos, no servidor gNMI. Esta função é chamada na função **run()**, detalhada no Bloco de Código C.1, e recebe como *input* os *paths* dos elementos que se pretendem monitorizar. Esta função é executada como uma *thread*, pois o *for* da linha 24, do Bloco de Código C.5, é bloqueante, uma vez que fica à espera de notificações das subscrições ativas que ao receber insere na *queue*, para ser processada pelo agente. Este método encontra-se no ficheiro *snmp_agent.py*.

```
1   def subscribe_thread(paths):
2       subscribe = {
3           'subscription': [
4           ],
5           'use_aliases': False,
6           'mode': 'stream',
7           'encoding': 'json_ietf'
8       }
9       for path in paths:
10          subscribe['subscription'].append(
11                  {
12                      'path': path,
13                      'mode': 'on_change',
14                      'sample_interval': 1000000000
15                  }
16          )
17      global gnmi_credentials
18      with gNMIclient(target=host, username=gnmi_credentials.getUser(), password=gnmi_credentials.getPassword(), insecure=True, debug = True) as gc:
```

```
19          telemetry_stream = gc.subscribe(subscribe=subscribe)
20          #pygnmi implements this 'for' as infinite loop
21          for telemetry_entry in telemetry_stream:
22              telemetry_entry_str = telemetryParser(telemetry_entry)
23              if not "sync_response" in telemetry_entry_str :
24                  queue.put(telemetry_entry_str)
```

Bloco de Código C.5: Subscrição dos Elementos

C.6 Gestão

No âmbito da gestão, temos três métodos principais que, dependendo do tipo de operação recebida pela notificação no método **Handle_Notification()**, serão executados. Não será detalhado o método **addStatusToMemory()**, pois já foi apresentado nas secções C.2 e C.4:

addStatusToMemory() — adiciona a configuração recebida às estruturas respetivas, criando os novos elementos no ficheiro de configuração e na *datastore state*.

changeStatusOfMemory() — altera a configuração recebida com o elemento respetivo, alterando o elemento no ficheiro de configuração e na *datastore state*.

delStatusofMemory() — elimina a configuração recebida do agente, eliminando o elemento do ficheiro de configuração e na *datastore state*.

```
1   def addStatusToMemory():
2       ...
3
4   def changeStatusOfMemory(obj):
5       log.info(obj)
6       #Check if are target config
7       if obj.config.key.js_path == ".snmp_agent.targets.target":
8           #Check if exits any config
9           if not obj.config.data.json == "{\n}\n":
10              #Return the JSON from the object
11              notification_target = json.loads(obj.config.data.json)
12              # One notification for entry
13              address = obj.config.key.keys[0]
14              nw_instance = notification_target['target']['network_instance']['value']
15              for t in targets:
16                  if t.get_address() == address :
17                      t.set_nw(nw_instance)
18
19              #Add to Telemetry and targets element
20              addTargetsToTelemetry()
21              # Add to the File
22              with open(FILENAME,"r+") as f:
23                  file_data = json.load(f)
24                  for target in file_data['targets']:
25                      if target['address'] == address:
26                          target['nw-instance'] = nw_instance
27                  # Eliminate duplicates
28                  aux_targets = []
29                  for target in file_data['targets']:
30                      if target not in aux_targets :
31                          aux_targets.append(target)
32                  file_data['targets'] = aux_targets
33                  f.seek(0)
34                  f.write(json.dumps(file_data, indent=4))
35                  f.truncate()
36
37      # Check if are configuration of a element
38      elif obj.config.key.js_path == ".snmp_agent.monitoring_elements.element":
```

```
39          #Check if exits any config
40          if not obj.config.data.json == "{\n}\n":
41              element_json = json.loads(obj.config.data.json)['element']
42              # Desiralization of the elements into variables
43              resource = obj.config.key.keys[0]
44              # Mandatory parameters
45              parameter = element_json['parameter']['value']
46              trigger_condition = element_json['trigger_condition']['value']
47              trap_oid = element_json['trap_oid']['value']
48              # Optional parameters
49              if "resource_filter" in element_json:
50                  resource_filter = [x['value'] for x in element_json['resource_filter']]
51              else:
52                  resource_filter = []
53
54              if "monitoring_condition" in element_json:
55                  monitoring_condition = element_json['monitoring_condition']['value']
56              else:
57                  monitoring_condition = ""
58
59              if "trigger_message" in element_json:
60                  trigger_message = element_json['trigger_message']['value']
61              else:
62                  trigger_message = ""
63
64              if "resolution_condition" in element_json:
65                  resolution_condition = element_json['resolution_condition']['value']
66              else:
67                  resolution_condition = ""
68
69              if "resolution_message" in element_json:
70                  resolution_message = element_json['resolution_message']['value']
71              else:
72                  resolution_message = ""
73
74              for e in elements:
75                  if e.getResource() == resource: # Verify if key list
76                      e.setChangeOperation(parameter,
77                                           monitoring_condition,
78                                           resource_filter,
79                                           trigger_condition,
80                                           trigger_message,
81                                           resolution_condition,
82                                           resolution_message,
83                                           trap_oid)
84
85              addElementsToTelemetry()
86              # Add to the File
87              with open(FILENAME,"r+") as f:
88                  file_data = json.load(f)
89                  elements_original = []
90                  for e in elements:
91                      elements_original.append(e.getJSON())
92
93                  file_data['monitoring_elements'] = elements_original
94
95                  f.seek(0)
96                  f.write(json.dumps(file_data, indent=4))
97                  f.truncate()
98
99  def delStatusofMemory(obj):
100     global elements, targets
101     #REMOVE TARGET ELEMENT
102     if obj.config.key.js_path == ".snmp_agent.targets.target":
103         #Remove From Memory
104         key = obj.config.key.keys[0]
105         for i in range(len(targets)):
106             if targets[i].get_address() == key:
107                 targets.pop(i)
108                 break
```

```
109
110             # Update Targets to State
111             removeTargetsOfTelemetry(key)
112             # Update Config
113             addStatusToConfigDataStore()
114
115             #Remove From File
116             with open(FILENAME,"r+") as f:
117                 file_data = json.load(f)
118                 for i in range(len(file_data['targets'])):
119                     if file_data['targets'][i]['address'] == key:
120                         file_data['targets'].pop(i)
121                         break
122
123                 f.seek(0)
124                 f.write(json.dumps(file_data, indent=4))
125                 f.truncate()
126         #REMOVE MONITORING ELEMENT
127         elif obj.config.key.js_path == ".snmp_agent.monitoring_elements.element":
128             #Remove From Memory
129             key = obj.config.key.keys[0]
130             for i in range(len(elements)):
131                 if elements[i].getResource() == key:
132                     elements.pop(i)
133                     break
134
135             # Update Elements to State
136             removeElementsOfTelemetry(key)
137             # Update Config
138             addStatusToConfigDataStore()
139
140             #Remove From FIle
141             with open(FILENAME,"r+") as f:
142                 file_data = json.load(f)
143                 for i in range(len(file_data['monitoring_elements'])):
144                     if file_data['monitoring_elements'][i]['resource'] == key:
145                         file_data['monitoring_elements'].pop(i)
146                         break
147
148                 f.seek(0)
149                 f.write(json.dumps(file_data, indent=4))
150                 f.truncate()
151
152 def addStatusToConfigDataStore():
153     global elements, targets
154     # Add Targets
155     for t in targets:
156         data = t.gNMISetOperation(log)
157         gnmiSET(data)
158
159     # Add Elements
160     for e in elements:
161         data = e.gNMISetOperation(log)
162         gnmiSET(data)
163
164 def gnmiSET(update_path) -> None:
165     global gnmi_credentials
166     with gNMIclient(target= host, username=gnmi_credentials.getUser(), password=gnmi_credentials.getPassword(), insecure=True, debug = True) as gc
            :
167         try:
168             data = gc.set(update=update_path,encoding="json_ietf")
169             #log.info(f"gNMI SET Operation ::: {data}")
170         except Exception as e:
171             log.info(e)
```

Bloco de Código C.6: Gestão

C.7 Validação das condições

processEntry() — Método responsável por processar os eventos retornados pela subscrição gNMI, utiliza métodos da classe *element.py* para implementar a lógica funcional do agente;
verifyIfEntryBelongs() — Método responsável pela verificação se a informação recebida pela subscrição gNMI pertence ao próprio elemento;
getSubPathsKeys() — Método responsável pela obtenção da sub-lista de elementos na classe, pode acontecer para um elemento genérico ter vários elementos específicos e cada elemento específico ter de ter o seu estado associado;
updateStatus() — Método responsável pela verificação do *trigger_condition* ou do *resolution_condition* e também pela atualização do estado do elemento;
verifyIfIsMonitoringPath() — Método que verifica se a informação recebida pela subscrição diz respeito a alterações do *parameter* ou do *monitoring_condition*;
checkFilter() — Método responsável por verificar se a informação recebida pertence ao conjunto *resource_filter*;

```
def processEntry(entry):
    global elements
    for e in entry:
        log.info("Original Entry : " + str(e))
        entry_path, value = cleanUPPath(e)
        for element in elements:
            path = element.getResource()
            if element.verifyIfEntryBelongs(log,entry_path,value):
                if entry_path in element.getSubPathsKeys():
                    result, event, base_trap_oid, specified_trap_oid = element.updateStatus(log,entry_path,value)
                    if result:
                        log.info(f"SEND SNMP TRAP with the Event: {event}")
                        sendToAllTargets(event + ": " + entry_path,base_trap_oid,specified_trap_oid)
                        element.set_traps_generated(element.get_traps_generated() + 1)
                        addElementsToTelemetry()
                    break
                else:
                    if element.verifyIfIsMonitoringPath(entry_path):
                        resource = "/".join(entry_path.split("/")[:-1])
                        key_path = resource +"/"+ element.getParameter()
                        element.setSubPath(log,key_path,value)
                        break
                    elif not element.verifyIfIsMonitoringPath(entry_path):
                        if element.checkFilter(log,entry_path):
                            element.addPath(entry_path,value)
                        break
```

Bloco de Código C.7: Validação das condições

C.8 Atualização de Estado

addTargetsToTelemetry() — método responsável por construir o JSON com as informações sobre os *Targets*, para adicionar à *datastore state*;
removeTargetsOfTelemetry() — método responsável por construir o JSON com as informações sobre os *Targets*, para remover da *datastore state*;

addElementsToTelemetry() — método responsável por construir o JSON com as informações sobre os *Elements*, para adicionar à *datastore state*;

removeElementsOfTelemetry() — método responsável por construir o JSON com as informações sobre os *Elements*, para remover da *datastore state*;

Add_Telemetry() — método responsável por adicionar no NDK o JSON recebido pelo parâmetro;

Delete_Telemetry() — método responsável por eliminar do NDK o JSON recebido pelo parâmetro;

```
def addTargetsToTelemetry() -> None:
    global targets
    for t in targets:
        js_path = f'.{agent_name}.targets.target{{.address=="{t.get_address()}"}}'
        json_content = { 'network-instance': f'{t.get_nw()}'}
        r = Add_Telemetry(js_path,json.dumps(json_content))

def removeTargetsOfTelemetry(address) -> None:
    js_path = f'.{agent_name}.targets.target{{.address=="{address}"}}'
    r = Delete_Telemetry(js_path)

def addElementsToTelemetry() -> None:
    global elements
    base_path = '.' + agent_name + ".monitoring_elements.element"
    for e in elements:
        js_key = e.getKey()
        js_path = base_path + f'{{.resource=="{js_key}"}}'
        json_content = e.getJSONElement()
        r = Add_Telemetry(js_path,json.dumps(json_content))

def removeElementsOfTelemetry(resource) -> None:
    js_path = f'.{agent_name}.monitoring_elements.element{{.resource=="{resource}"}}'
    r = Delete_Telemetry(js_path)

def Add_Telemetry(js_path, js_data):
    telemetry_stub = telemetry_service_pb2_grpc.SdkMgrTelemetryServiceStub(channel)
    telemetry_update_request = telemetry_service_pb2.TelemetryUpdateRequest()
    telemetry_info = telemetry_update_request.state.add()
    telemetry_info.key.js_path = js_path
    telemetry_info.data.json_content = js_data
    #log.info(f"Telemetry_Update_Request :: {telemetry_update_request}")
    telemetry_response = telemetry_stub.TelemetryAddOrUpdate(request=telemetry_update_request, metadata=metadata)
    return telemetry_response

def Delete_Telemetry(js_path):
    telemetry_stub = telemetry_service_pb2_grpc.SdkMgrTelemetryServiceStub(channel)
    telemetry_delete_request = telemetry_service_pb2.TelemetryDeleteRequest()
    telemetry_delete = telemetry_delete_request.key.add()
    telemetry_delete.js_path = js_path
    #log.info(f"Telemetry_Delete_Request :: {telemetry_delete_request}")
    telemetry_response = telemetry_stub.TelemetryDelete(request=telemetry_delete_request, metadata=metadata)
    return telemetry_response
```

Bloco de Código C.8: Atualização de Estado

C.9 Envio das Traps

sendToAllTargets() — Método responsável por enviar as SNMP Traps, para todos os Targets configurados, recebe como *input* o evento ocorrido (*msg_entry*) e os OIDs (t1 e t2), a utilizar no envio da SNMP Trap. Este método encontra-se no ficheiro *snmp_agent.py*.

sendSNMPTrap() — Método responsável por enviar as SNMP Traps, recebe como *input* o evento

ocorrido (*msg_entry*), os OIDs (trap_oid_1 e trap_oid_2) a utilizar no envio da SNMP Trap e o endereço do *Collector* (*target*).

```
def sendToAllTargets(msg_entry,t1,t2):
    global targets
    for target in targets:
        sendSNMPTrap(msg_entry,target,t1,t2)

def sendSNMPTrap(msg_entry,target,trap_oid_1,trap_oid_2):
    p = f"ip netns exec {target.get_nw()} snmptrap -v 2c -c public {target.get_address()} 0 {trap_oid_1} {trap_oid_2} s \"{msg_entry}\""
    out = subprocess.run(p, shell=True)
    return None
```

Bloco de Código C.9: Envio das Traps

C.10 Ficheiro YAML

Ficheiro YAML do agente com informação importante para o NDK, nomeadamente, *path* para o executável do agente, *path* para o modelo de dados do agente e outros comandos que o agente executa.

```
snmp_agent:
    path: /etc/opt/srlinux/appmgr/user_agents/
    launch-command: /etc/opt/srlinux/appmgr/user_agents/snmp_agent.sh
    search-command: /bin/bash /etc/opt/srlinux/appmgr/user_agents/snmp_agent.sh
    wait-for-config: No
    failure-threshold: 100
    failure-action: wait=forever
    yang-modules:
        names:
            - "snmp_agent"
        source-directories:
            - "/etc/opt/srlinux/appmgr/user_agents/yang/"
```

Bloco de Código C.10: Ficheiro YAML do agente

C.11 Ficheiro YANG

Ficheiro YANG do agente com o modelo de dados que o agente utiliza.

```
module snmp_agent {
    yang-version "1.1";
    namespace "urn:srl_test/snmp-agent";
    prefix "snmp_agent";
    revision "2022-03-14" {
        description "Initial revision";
    }
    grouping snmp_agent-top {
        description "Top level grouping for snmp_agent sample app";
        container snmp-agent{
            container monitoring_elements{
                list element{
                    key resource;
                    leaf resource {
                        type string;
                        mandatory true;
                        description "Path of the resource to monitoring";
                    }
                    leaf parameter {
                        type string;
                        mandatory true;
```

```
22                          description "Element status to keep track";
23                      }
24                      leaf monitoring_condition {
25                          type string;
26                          description "Element status to keep track";
27                      }
28                      leaf-list resource_filter {
29                          type string;
30                          description "Filter";
31                      }
32                      leaf trigger_condition {
33                          type string;
34                          mandatory true;
35                          description "Condition that will trigger the trap";
36                      }
37                      leaf trigger_message {
38                          type string;
39                          description "MIB Event";
40                      }
41                      leaf resolution_condition {
42                          type string;
43                          description "Resolution Trigger Condition";
44                      }
45                      leaf resolution_message {
46                          type string;
47                          description "Resolution message";
48                      }
49                      leaf trap_oid {
50                          type string;
51                          mandatory true;
52                          description "Trap OID to use";
53                      }
54                      leaf traps_generated {
55                          type string;
56                          config false;
57                          description "How many traps this element produces";
58                      }
59                  }
60              }
61              container targets{
62                  list target{
63                      key address;
64                      leaf address {
65                          type string;
66                          mandatory true;
67                          description "Address of target";
68                      }
69                      leaf network-instance {
70                          type string;
71                          description "Network-instance to use";
72                      }
73                      leaf community-string {
74                          type string;
75                          description "Community-string to use";
76                      }
77                  }
78              }
79          }
80      }
81      uses snmp_agent-top;
82  }
```

Bloco de Código C.11: Ficheiro YANG do agente