FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Monitorização de um Sistema Publish-Subscribe ROS para Enumeração e Deteção de Intrusões

João Pedro Xavier Araújo

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Ricardo Santos Morla

19 de Julho de 2019

Resumo

A industria encontra-se numa fase revolucionaria conhecida por quarta revolução industrial. Esta permite a ligação entre o meio físico e o meio digital, dando origem a sistemas ciberfísicos, *robots* integrados e comunicações entre entidades físicas e digitais. Estas redes podem ser implementadas em plataformas como Robot Operating System que fornece protocolos de comunicação iguais para todos os elementos da rede, ultrapassando o obstáculo de compatibilidade entre estes.

Este conceito ainda é recente e, consequentemente, a segurança deste tipo de redes é o aspeto mais importante a desenvolver. A monitorização e enumeração são conceitos muito importantes para a segurança de um sistema, que possibilitam o combate de atividade maliciosa através do conhecimento do sistema, dos seus elementos e padrões de comunicação.

Desta forma, este trabalho de dissertação procura desenvolver uma estrutura ou mecanismo que enumere elementos e a atividade criada por estes numa rede implementada em Robot Operating System e que apresente esta informação de forma clara e concisa. O mecanismo procura enumerar os vários elementos de uma rede que comuniquem entre si e identificar a suas interações, permitindo o conhecimento da rede, proporcionando uma melhor monitorização destes sistemas e possibilitando a facilitando a deteção ou até prevenção de falhas de segurança, anomalias, ataques ou intrusões.

Abstract

The Industry is in a revolutionary phase known as the fourth industrial revolution. This allows a connection between the physical and the digital environment, giving rise to cyberphysical systems and allowing the comunication between phisical and digital entities. These networks can be deployed on platforms such as Robot Operating System that provide uniform comunication protocols to all network elements, overcoming the compatibility hurdle between them.

This concept is still recent and, consequently, the security of this type of networks is the most important aspect to develop. Monitoring and enumeration are important to the security of this systems, which enables the fight against malicious activity through knowledge of the system, its elements and communication patterns.

Thus, this dissertation work seeks to develop a structure or mechanism that identifies and enumerates elements and the activity created by them in a network implemented in Robot Operating System and that presents this information clearly and concisely. The mechanism seeks to enumerate the various elements of a network that communicate with each other and identify their interactions, enabling knowledge of the network, providing better monitoring of these systems and facilitating the detection or even prevention of security breaches, anomalies, attacks or intrusions.

Agradecimentos

A realização desta dissertação de mestrado contou com importantes apoios e incentivos sem os quais não se teria tornado uma realidade e aos quais estou extremamente grato.

Ao professor Ricardo Santos Morla pela sua orientação, apoio, disponibilidade e total colaboração no solucionar de dúvidas e problemas que foram surgindo ao longo da realização deste trabalho.

Agradecimentos para todos os meus amigos pelo apoio, paciência e por todos os momentos que passámos.

Por último, gostaria também de agradecer à minha família, especialmente aos meus pais por todo o suporte que me deram, tudo que fizeram por mim e por nunca duvidar de mim.

O meu sincero muito obrigado.

João Pedro Xavier Araújo

Conteúdo

Lista de Figuras

Abreviaturas e Símbolos

API	Application Programming Interface
CoAP	Constrained Application Protocol
DoS	Denial-of-Service
HTTP	Hypertext Transfer Protocol
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System
IIoT	Industrial Internet of Things
IoT	Internet of Things
IP	Internet Protocol
IPS	Intrusion Prevention System
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
NIDS	Network-based Intrusion Detection System
OS	Operating System
PPA	Personal Package Archives
ROS	Robot Operating System
RPC	Remote Procedure Call
URL	Uniform Resource Locator
XML	Extensible Markup Language

Capítulo 1

Introdução

1.1 Enquadramento

Com o aparecimento da quarta revolução industrial, também conhecida como Indústria 4.0, a Internet of Things (IoT) também está a chegar ao ramo empresarial e industrial de forma a ampliar a eficácia e eficiência das suas operações.

Industrial Internet of Things (IIoT) refere-se a esta variante de IoT para o ramo empresarial e sobretudo para o ramo industrial, com um grande foco em aplicações industriais, onde possa ser necessário comunicação entre máquinas ou dispositivos, elevadas quantidades de processamento e armazenamento de dados, devido à quantidade de dispositivos, *machine learning*, entre outros.

IIoT vai além dos dispositivos normais de consumo e da interconexão de dispositivos físicos normalmente associados à IoT. Este torna possível uma interconexão entre redes de processos operacionais e sistemas de controlo industrial, incluindo interfaces homem-máquina, controlo de supervisão e aquisição de dados, sistemas de controlo distribuído e controladores lógicos programáveis.

Para facilitar a implementação deste tipo de redes, existem plataformas como Node-RED e Robot Operating System que possibilitam a comunicação entre dispositivos, usando protocolos de interface entre estes.

1.2 Motivação

O conceito já não é novo mas a sua implementação ainda não é muito comum e tem muitos aspetos onde pode ser desenvolvida e melhorada, mesmo existindo plataformas que suportem redes IoT.

Um desses aspetos é a segurança, onde dispositivos, como sensores, câmaras, e mecanismos robóticos, ente outros, podem ser alvo para atacantes que explorem as falhas de segurança no software destes dispositivos ou nos protocolos de comunicação entre os vários dispositivos, podendo comprometer o seu funcionamento, adquirir informação importante que estes contenham ou obter controlo destes usando-os para outros fins como plataforma para ataques a outros dispositivos.

Sistemas de deteção ou prevenção de ataques não são inexistentes. Existem até sistemas de deteção *open-source*, como é o caso do Suricata e do Snort, mas ainda existe falta de suporte para redes IIoT.

1.3 Objetivos

O conhecimento de uma rede e dos seus elementos é crucial para a deteção de padrões irregulares nas comunicações ou até na deteção da presença de elementos intrusos na rede. Esta dissertação tem como objetivo a criação de uma ferramenta de apoio à monitorização, análise e simplificação na identificação de padrões de comunicação ou dos próprios elementos de uma rede.

Capítulo 2

Revisão Bibliográfica

2.1 Plataformas IIoT

2.1.1 Node-RED

Node-RED [1][2] é uma ferramenta programável *open-source* para conectar vários dispositivos de *hardware* ou *software* com diferentes protocolos, *drivers* e APIs, podendo assim criar redes entre os vários sistemas de forma poderem comunicar entre si, e pode ser instalado localmente em vários sistemas operativos como Windows, Linux ou macOS, em dispositivos como o raspberry pi ou sistemas Android, bem como em sistemas *Cloud*.

Originalmente desenvolvida pela IBM's Emerging Technology Services e fazendo agora parte da JS Foundation, foi implementada com a *framework* Node.js num *runtime* JavaScript e com um editor visual que pode ser acedido através de um *browser* e é baseado numa programação orientada a componentes conhecida por *flow-based* onde a interface gráfica web consiste numa representação da rede através de uma arquitetura de grafos onde os nós deste interligam os vários processos que podem ser facilmente programados, bastando desenhar a rede pretendida sem a necessidade de escrever linhas de código ou compreender o código automaticamente escrito pela ferramenta.

Esta ferramenta tem ainda uma comunidade na qual extensões, com base em ficheiros JSON, podem ser instaladas de forma a aumentar a funcionalidade dos nós de forma a amplificar o potencial desta.

2.1.2 ROS

Robot Operating System [3][4] ou simplesmente ROS, é uma plataforma de desenvolvimento de software para *robots* que fornece inúmeros serviços como comunicação e controlo de dispositivos. Tal como o Node-RED, os processos desta estão organizados numa arquitetura de grafos, criando uma rede de processo da mesmo dispositivo ou dispositivos diferentes. As duas plataformas mencionadas também fornecem o serviço *publish-subscribe* onde cada nó pode publicar ou

subscrever a serviços de outros nós, que é gerido por um serviço central da plataforma. O *Constrained Application Protocol* (CoAP) [5] e o *Message Queuing Telemetry Transport* (MQTT) [6] são dois protocolos que implementam serviços *publish-subscribe* e são frequentemente usados em redes IoT, possibilitando comunicação entre dispositivos em nós da rede diferentes. Cada nó necessita de pedir a este serviço central para efetuar qualquer ação, podendo o nó pertencer à mesma máquina que aloja o serviço central ou outro dispositivo.

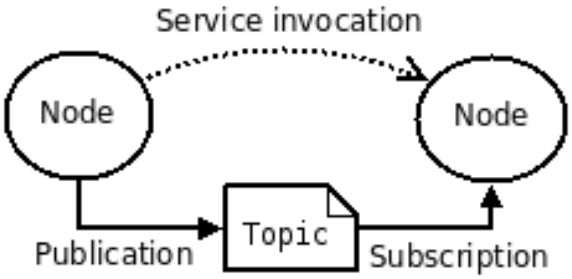

Figura 2.1: Conceito Básico de Robot Operating System

A ferramenta utiliza dois protocolos de comunicações diferentes. Um para a gestão do estado dos nós, *master* e os seus tópicos e serviços e negociação de comunicações, chamado XML-RPC. O segundo protocolo tem duas variantes. Uma é baseada no protocolo de transporte TCP e outro no UDP. Estes são os protocolos de transporte da ferramenta ROS para a troca de informação dos próprios tópicos ou serviços.

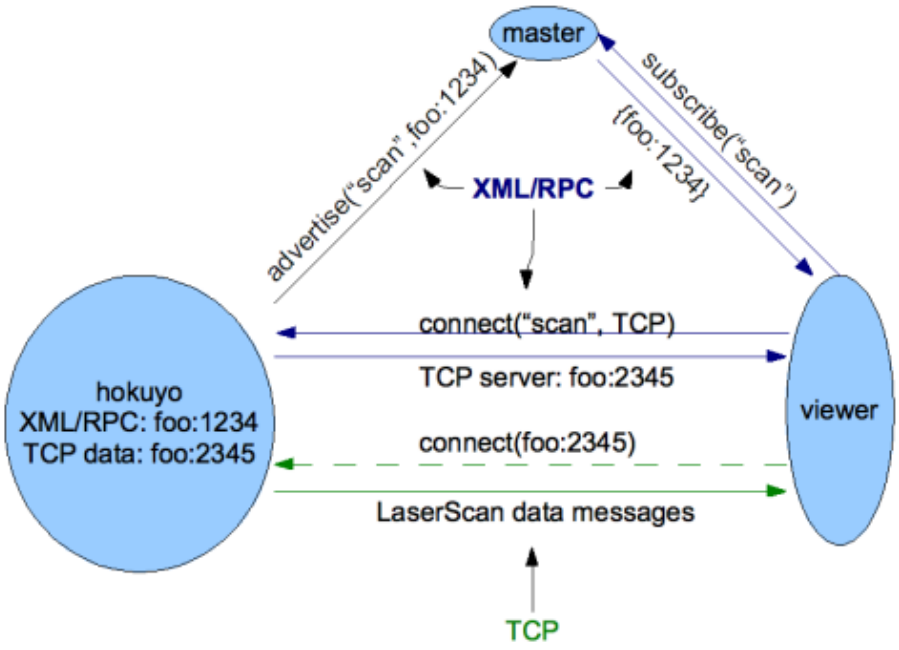

Figura 2.2: Diagrama de Comunicações da ferramenta Robot Operating System

2.1.2.1 XML-RPC

O protocolo XML-RPC que é baseado em HTTP e funciona como um protocolo de pedido-resposta, com o uma codificação em XML. As siglas RPC significam 'Remote Procedure Call', ou seja, permite fazer pedidos procedimentos a executar ou pedidos para máquinas noutros endereços de uma rede com uma codificação como se tratasse de um procedimento local, sem ter de se preocupar com detalhes da interação remota. Na figura 2.3 e 2.4 podemos observar um exemplo de um pedido e de uma resposta entre dois processos diferentes, respetivamente.

Figura 2.3: Exemplo de um pedido registerPublisher do protocolo XML-RPC

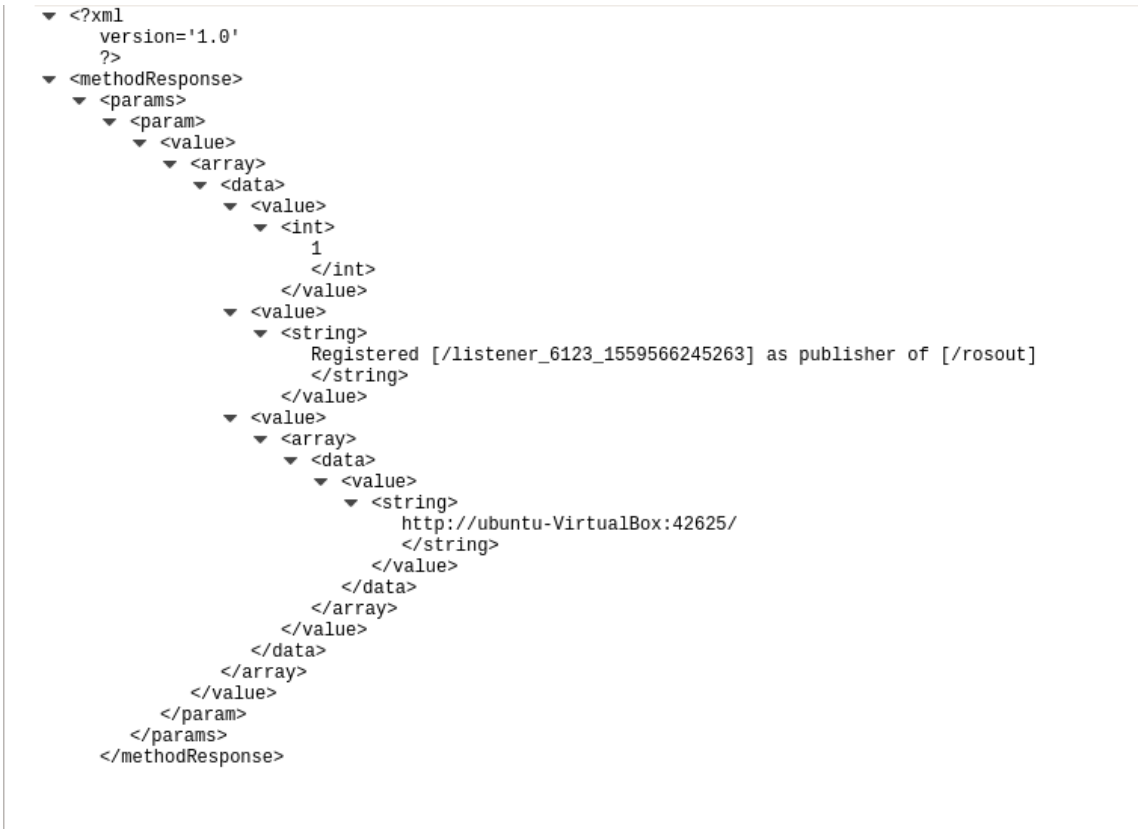

Figura 2.4: Exemplo de uma resposta a um pedido registerPublisher do protocolo XML-RPC

Nas comunicações que usam o protocolo XML-RPC da ferramentas ROS, existem inúmeros métodos que podem ser usados pelo *master* e pelos nós. Aqui serão enumerados alguns dos métodos de registo e estado da rede mais relevantes, apresentados na seguinte lista:

- **registerService**: Este método regista a entidade que invocou este método como o provedor de um serviço especificado na função.
- **unregisterService**: Este método tem a função inversa ao método anterior. Caso a entidade que invoque este método não seja fornecedora do serviço em questão, o retorno indica que não houve nenhuma alteração no registo dos provedores e nos seus serviços.
- **registerSubscriber**: Forma de subscrição da entidade que invocou o método a um tópico. O subscritor recebe uma lista dos publicadores do tópico especificado no retorno desta função. Caso haja novos publicadores para o mesmo tópico, o subscritor é informado através do método *publisherUpdate*.
- **unregisterSubscriber**: Este método termina o registo de uma entidade a um tópico. Se a última variável do retorno for zero, implica que a entidade não estava subscrita ao tópico e, portanto, não houve nenhuma modificação.
- **registerPublisher**: Registo de uma entidade como publicador de um tópico. O retorno deste método contém a lista de subscritores ao tópico especificado.

- **unregisterPublisher**: Este método tem a função oposta ao método anterior, apenas informando o sucesso do pedido à entidade chamadora da função.
- **publisherUpdate**: Este método informa o destinatário dos publicadores atuais do tópico especificado.
- **requestTopic**: Pedido de alocação de um canal para comunicação. O pedido é enviado com uma lista dos protocolos de comunicação desejados e o retorna com o protocolo selecionado, entre outros parâmetros que possibilitem a conexão entre as entidades.

2.2 Vulnerabilidades em Plataformas IIoT

Nesta secção serão apresentadas vários tipos de vulnerabilidades [7][8] e ataques conhecidos [9][10], nomeadamente para plataformas IIoT.

2.2.1 Tipos de Vulnerabilidades

- **Disponibilidade**: Comprometimento de comunicação como alteração do protocolo *routing*, influenciar o modo de operação modificando rotas, criando *loops*, *gerando* erros, *modificando* atrasos de mensagens, podendo ser feito a nível físico através de *jamming* ou interferências, *Denial of Service* e outros métodos que limitem o acesso a serviços. O próprio dispositivo ou *router* pode ficar indisponível por exaustão dos seus recursos como a capacidade de processamento ou memória, podendo ser atingido por vulnerabilidades de aplicações ou instalação de *software* malicioso.
- **Confidencialidade**: risco de exposição de informação como estados dos nós da rede ou dispositivo, informação de recursos como memória disponível ou serviços em execução, podendo explorar vulnerabilidades destes, ou mesmo dados pessoais ou de entidades por falta de encriptação. Este tipo de vulnerabilidade também está presente ao nível da rede onde existe também riscos de exposição como a informação de rotas e topologia da rede que pode resultar na identificação de equipamentos que poderão ter vulnerabilidades que afetem o resto da rede.
- **Autenticação**: risco entidades maliciosas se fazerem passar por outra entidade e desta forma obter os privilégios sobre recursos ou serviços, resultando de falhas ou não uso de protocolos com certificados digitais de identificação.

2.2.2 Ataques na Camada de Rede

- **Clone ID:** cópia e uso de identificação de outra entidade com o objetivo de ganhar acesso a informação destinada à entidade original.
- **Sybil:** semelhante ao ataque *Clone ID*, este ataque obtém várias identidades e atribui estas ao mesmo nó, podendo fazê-lo para grande parte das entidades da rede.

- **Hello Flooding:** Mensagens *Hello* são usadas como anúncio da existência de um dispositivo para outros nós. O atacante pode transmitir estas mensagens constantemente causando um aumento no tráfego em todos os nós da rede e podendo baixar a capacidade desta.

- **Wormhole:** Este tipo de ataque requer pelo menos dois atacantes. Cria uma ligação entre os vários atacantes e congestionando significativamente os nós que fazem a ligação entre os atacantes, causando uma interrupção de comunicações de uma rede ou secção desta.

- **Denial of Service:** Neste tipo de ataque, é negado acesso a serviços ou comunicação com outros nós a entidades de uma rede, tornando estes recursos indisponíveis. Existem variados métodos de o aplicar. Um desses métodos é o congestionamento de um nó, de tal forma que a quantidade de informação vá além da sua capacidade de processamento.

- **Selective Forwarding:** seleção e transmissão de pacotes, podendo este filtrar protocolos específicos, de forma a interromper a transmissão de outros. Pode resultar noutros tipos de ataques como *Denial of Service*.

2.2.3 Vulnerabilidades na Camada de Aplicação

- **Data Privacy:** Encriptação é um conceito muito importante na privacidade de informação, não permitindo que atacantes obtenham toda a informação facilmente, sem necessidade de realizar desencriptação. Um dos exemplos é a falta de encriptação de credenciais.

- **Autentication:** Em caso de existência de mecanismos de autenticação com credenciais, o envio da mensagens de conexão do um dispositivo para um *broker* pode ser feito sem encriptação, contendo as credenciais legíveis e vulneráveis a uma entidade atacante que esteja escutar tráfego na rede.

- **Data Integrity:** Este tipo de vulnerabilidade existe quando existe a possibilidade da alteração de informação, por exemplo, no envio de mensagens, caso não haja nenhum mecanismo de confirmação da integridade da informação. No caso de um sistema de publicação e subscrição de serviços, o atacante pode alterar o tópico do serviço enviado pelo publicador ou pode resultar na alteração do *link* de atualização do *firmware*, podendo resultar na instalação de *software* malicioso. Uma das consequências que podem resultar da instalação de *software* malicioso é a transformação destes dispositivos em *botnets*, ficando sobre o controlo de outras entidades que podem obter informação destes e controlá-los para fazer ataques.

2.3 Características dos Sistemas de Deteção de Intrusão

Nesta secção serão apresentadas as caraterísticas gerais de sistemas de deteção de intrusão [11][7], em particular, sistemas de deteção para IoT. Existem dois tipos de sistemas de deteção de intrusão, sendo estes os seguintes:

- **NIDS (Network-based IDS):** monitoriza uma ou várias redes a partir de um ou vários pontos de acesso nas redes.
- **HIDS (Host-based IDS):** contido nos próprios dispositivos e não só pode monitorizar tráfego interno como chamadas do sistema, processos ativos e comunicação entre estes ou modificação de ficheiros do sistema, como também uma monitorização do tráfego de rede visto pelo *host*.

Estes sistemas podem ainda ser caraterizados nos seguintes tópicos, que serão abrangidos nas subsecções seguintes:

- Estratégias de Localização
- Métodos de Deteção
- Estratégia de Validação

2.3.1 Estratégias de Localização

Numa rede IoT, um IDS pode ser colocado no *router* que conecta esta a outras redes, em *hosts* ou mesmo em dispositivos físicos da rede. As seguintes subsecções apresentam as principais estratégias de localização de um IDS.

2.3.1.1 Localização Distribuída

Uma das estratégias é a localização distribuída, onde são colocados IDSs em todos os nós físicos da rede, podendo assim monitorizar recursos internos e até outros nós que comuniquem com o nó *host*. Esta estratégia pode ser optimizada configurando os nós de forma hierárquica onde alguns nós são configurados para monitorizar determinados nós, de maneira a evitar que todos os nós monitorizem as mesmas actividades.

2.3.1.2 Localização Centralizada

Nesta estratégia, a localização do IDS é centralizada, por exemplo, é colocado no *router* ou num *host*. Neste tipo de localização, o IDS tem uma completa monitorização de tráfego entre os nós da rede e o exterior e vice-versa, mas pode ter dificuldade em monitorizar tráfego entre nós da rede em caso de ataques realizados no interior desta.

2.3.1.3 Localização Híbrida

Existem ainda duas abordagem que podem ser designadas como híbridas. A primeira abordagem implica a organização da rede em regiões onde IDSs são colocados em determinados nós que ficam encarregues de tráfego de intra-região e inter-regiões, distribuindo a monitorização por dispositivos e redes diferentes. A segunda opção consiste na presença de um IDS central no *router* que delimita a fronteira da rede para além dos sistemas colocados em nós estratégicos como os da primeira abordagem.

2.3.2 Métodos de Deteção

Neste subsecção será descrita a teoria dos métodos de deteção mais tradicionais e em que se baseiam. Estes métodos podem evoluir para outros mais complexos e sofisticados como a deteção baseada em *Data Mining*, que utiliza elevada quantidade de informação para deteção de atividades anómalas, ou deteção baseada em Estatísticas, em que aplica modelos estatísticos para confirmar a existência de intrusão, ou até baseada em Inteligência Artificial ou *Machine Learning*, aprendendo progressivamente através de modelos matemáticos.

2.3.2.1 Deteção baseada em Assinatura

Em deteção baseada em assinatura, o método de deteção de ataques consiste em comparar padrões de ações no sistema com padrões de ataques guardados numa base de dados do IDS. Este método é eficaz na deteção de ataques conhecidos pelo sistema, mas é pouco útil na defesa contra ataques com padrões desconhecidos.

2.3.2.2 Deteção baseada em Anomalias

Este tipo de deteção de atividades maliciosas compara as atividades de um sistema com o comportamento normal deste e gera alertas se o desvio do comportamento for maior que um limite determinado. Este método é eficiente e tem bons resultados se o sistema realizar tarefas simples mas pode gerar muitos alertas falsos se o sistema realizar tarefas mais rebuscadas, caso as atividades não se assemelhe ao comportamento normal predefinido no IDS.

2.3.2.3 Deteção baseada em Especificação

Este método de deteção usa o mesmo principio que a deteção baseada em anomalias em termos de deteção de ataques pelo desvio em relação ao normal funcionamento deste mas requer limites de deteção para quando o sistema não se encontra em funcionamento adequado sejam feitos por especialistas de forma a reduzir a probabilidade de detetar um falso positivo e assim melhorando a eficácia de deteção.

2.3.3 Estratégia de Validação

Validação consiste na verificação da eficácia do modelo de deteção de intrusão. Existem enumeras formas de validação de um modelo, podendo estas serem distinguidas em duas categorias de fonte da informação em que cada modelo de validação se baseia. Estas fontes de informação são os peritos ou entidades com experiência na matéria ou com dados experimentais. A primeira fornece uma visão do modelo mais qualitativa e subjectiva do modelo, enquanto a segunda proporciona uma visão mais quantitativa e objetiva.

Os modelos podem também ser separados em classes em termos de estratégias de validação. Podem ser classificados pelos seguintes métodos:

- Hipotético: validação por hipótese podendo ter pouca relação com a realidade.

- Teórico: validação com suporte em argumentos precisos ou formais.
- Simulação: validação com base em informação obtida na simulação do sistema
- Empírica: validação com base em experiências com sistemas reais.
- Nenhum: Nenhum tipo de validação é efetuado.

2.4 IDS Open-Source

2.4.1 Suricata

Suricata [12] é uma sistema *open-source* de monitorização de redes, deteção e prevenção de intrusão em tempo real. Este sistema deteta ameaças conhecidas e atividades maliciosas, bem como anomalias que possam ocorrer na rede que monitoriza, sendo capaz de inspecionar elevado fluxo de tráfego devido à sua capacidade *multi threading*. É ainda capaz de detetar inúmeros protocolos de comunicação em qualquer porta e usa protocolos de *input* e *output* frequentemente utilizados outras ferramentas, permitindo alta compatibilidade com outros sistemas e ferramentas, inclusive ferramentas ou extensões para IoT.

Uma destas extensões para IoT é o Suricata-IoT [13] que suporta dispositivos IoT, as plataformas Raspberry Pi e Itron Riva Edge e protocolos como o CoAP [5] e o MQTT [6], que são frequentemente utilizados em redes IoT.

2.4.2 Snort

O Snort [14] é um sistema de detecção e prevenção de intrusão *open-source* capaz de realizar análises de tráfego em tempo real. O Snort realiza análise de protocolo, pesquisa e correspondência de conteúdo. Também pode ser usado para detetar ataques, ataques de URL semântico, *buffer overflows*, entre outros.

Este pode ser configurado em três modos principais: *packet sniffer*, *packet logger* e IDS/IPS. No primeiro modo, o programa imprime para a consola os pacotes da rede lidos, no segundo modo armazena os pacotes em memória e no terceiro modo monitoriza o tráfego de rede e analisa-o em relação a um conjunto de regras definido pelo utilizador.

2.5 Ferramentas de Captura de Pacotes

2.5.1 Wireshark

A ferramenta Wireshark é *packet analyzer open-source* que permite a analise e monitorização de redes. No final de 1990, era conhecido como Ethereal, que era usado para capturar e analisar pacotes. No entanto, no verão de 2006, devido a questões legais, foi renomeado para WIRESHARK.

O Wireshark consegue interpretar diversos protocolos como HTTP, TCP, UDP, VoIP, e muito mais. Este é baseado na ferramenta *libpcap* e é capaz de capturar pacotes em tempo real, análise de comunicações e protocolos.

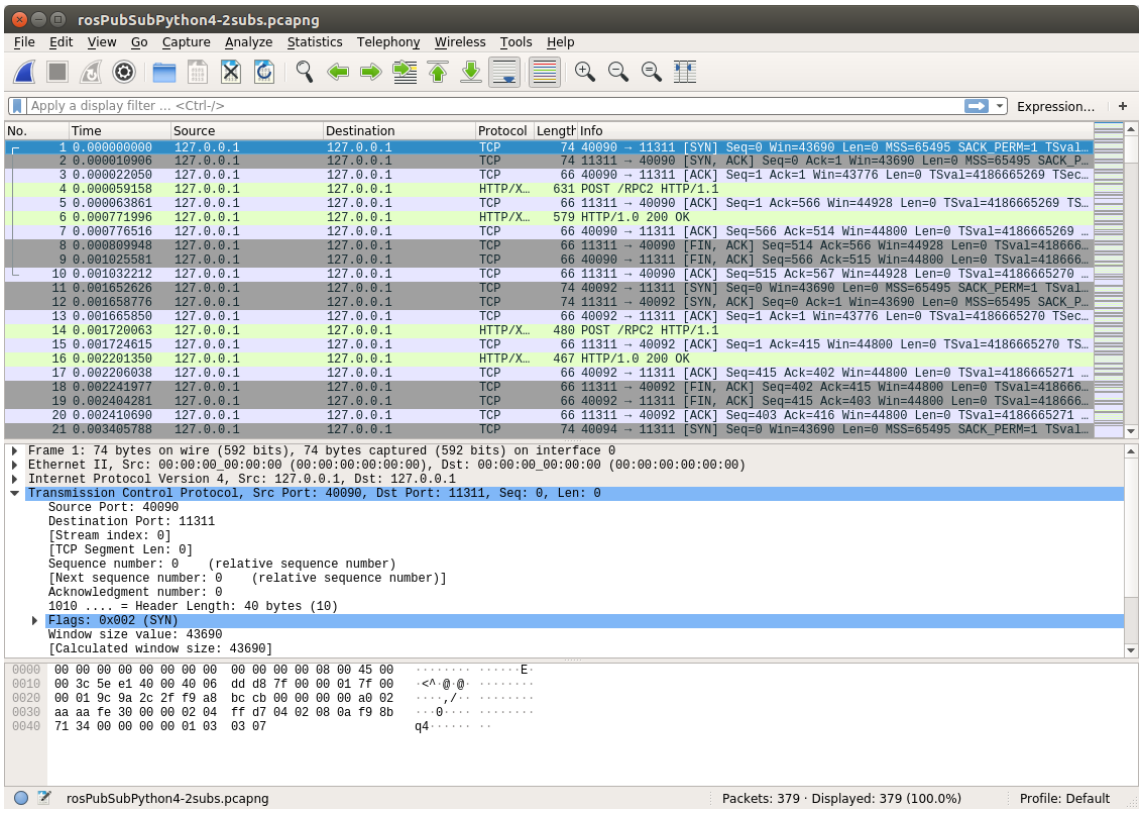

Figura 2.5: Visualização da Ferramenta Wireshark em Execução

Wireshark é uma ferramenta multi-plataforma, simples de descarregar e instalar. Esta pode ser executada em plataformas como UNIX (NetBSD, OpenBSD, Apple Mac OS X, etc.), LINUX (Debian, Ubuntu, Slackware, etc.), Windows (XP, Vista, 7, 10, etc.).

A parte básica do software Wireshark é a ferramenta *pcap*, também conhecida como *Wincap* quando se trata de um sistema operativo *windows*, e permite que o Wireshark seja executado no sistema. O modo promíscuo é uma característica principal do Wireshark, que permite a captura de pacotes através da rede.

2.6 Conclusão

Após uma revisão bibliográfica sobre a matéria em questão, verificou-se o pouco desenvolvimento da segurança para plataformas IIoT, em particular, na utilização de sistemas de deteção intrusão *open-source* nestas mesmas redes.

Capítulo 3

Caraterização do Problema

Com o avanço da *Internet* surgiu um novo conceito de inovação industrial. Este conceito chama-se Industria 4.0 e tem como objetivo aumentar a produtividade, eficiência, eficácia e disponibilidade de qualquer tipo de sistemas, através da conexão de vários elementos, permitindo a comunicação entre estes. Esta conectividade pode ser feita entre dispositivos ou mecanismos como sensores, *robots*, bases de dados, e outros sistemas, e podem facilmente organizados e supervisionados a partir de ferramentas de monitorização já existentes.

Algumas destas ferramentas, já mencionadas na secção 2.1 do capitulo 2, são plataformas como o Robot Operating System, ou ROS, e Node-RED. Porém a segurança destas plataformas ou ferramentas é um aspeto pouco explorado mas com grande potencial de desenvolvimento, o que leva a redes destes domínios poderem apresentar várias vulnerabilidades e estarem sobre forte ameaça de ataques ou intrusões.

Robot Operating System foi a plataforma escolhida para este trabalho. Esta possibilita a comunicação entre entidades físicas e virtuais através de um sistema de publicação e subscrição onde cada entidade pode publicar e/ou subscrever a serviços disponibilizados por terceiros. Como as entidades podem ter diversos protocolos que podem não ter compatibilidade entre eles, esta plataforma fornece um servidor central, também conhecido como 'master' ou 'broker', que fica encarregue do estado da rede. Este permite que entidades se conectem e publiquem os seus serviços e subscrevam a outros, informando qual o conteúdo existente e quem o fornece e permite estabelecimento de conexões utilizando os seus protocolos de comunicação.

Porém, para que seja possível a prevenção de vulnerabilidades ou ataques, é necessário ter conhecimento do estado da rede de forma a prevenir a alertar entidades intrusas, por exemplo, as que entidades estão conectadas, quais ou quantos serviços estão disponíveis, quais os publicadores ou subscritores, entre outros. Este processo é conhecido como enumeração. Este é extremamente importante no âmbito da segurança de uma rede. É utilizado por atacantes para tomar conhecimento de potenciais vulnerabilidades, mas também é utilizado de forma defensiva para identificação de falhas, anomalias ou até ataques e intrusões.

Na figura 3.1 é ilustrado o problema em questão, onde é apresentado dois exemplos de elementos da rede, um publicador e um subscritor, e o *broker* ROS também conhecido como ROS

MASTER. Estes elementos comunicam entre si através de uma interface de rede. Todas as comunicações entre cada um dos elementos são direcionadas por essa interface.

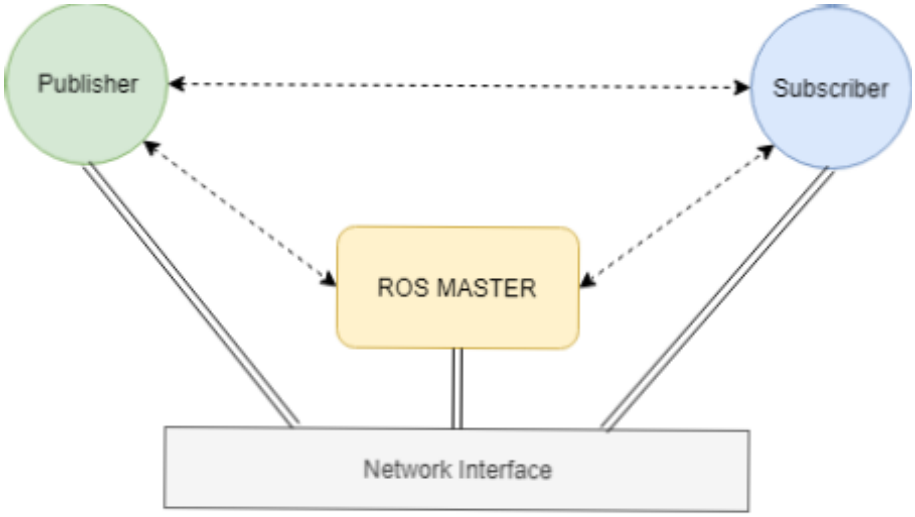

Figura 3.1: Diagrama do Problema

Este trabalho procura solucionar a identificação e enumeração das comunicações e elementos que estejam em comunicação ativa num ambiente Robot Operating System exemplificado na imagem anterior.

Capítulo 4

Solução Proposta

A solução proposta procura resolver o desafio do problema através do desenvolvimento de um mecanismo que capture as comunicações e as interprete de forma a identificar e enumerar as entidades da rede e as suas comunicações. A figura 4.1 ilustra a estrutura proposta que consiste na captura das comunicações na interface de rede ao qual o *broker* ROS MASTER está conectado fisicamente, sendo este o único ponto da rede por onde as ligações entre os elementos da rede e o ROS MASTER atravessam. Após a sua captura, estas são filtradas por um elemento da estrutura e por fim interpretadas pelo programa desenvolvido que imprime as ligações e os elementos da rede envolvidos.

Para simplificar o processo de trabalho, foi utilizada a ferramenta de captura de pacotes Wireshark, descrita na secção 2.5.1 do capitulo 2, de forma a facilitar a análise dos pacotes de dados trocados entre as entidades em questão e para guardar fragmentos de comunicação, possibilitando a execução do mesmo teste inúmeras vezes sem a necessidade da execução do teste em tempo real.

Todas as comunicações são capturadas, pelo que grande parte destas não são relevantes ao problema, o que complica a sua análise. Para ultrapassar este obstáculo, foi utilizado uma segunda ferramenta com o intuito de filtrar toda a informação irrelevante e apenas adquirir informação interessante. Esta ferramenta é o Suricata, já mencionada na secção 2.4 do capitulo 2. Esta possibilita criar alertas quando mensagens contenham padrões de informação desejados e guardar em ficheiros para mais tarde serem analisados.

Por último, essa informação necessita de ser processada. Nesse âmbito, foi criado um pequeno programa que lê a informação filtrada e apresenta-a numa linguagem mais compreensível para o utilizador. Nas seguintes secções serão apresentadas as soluções para as regras da ferramenta de filtragem Suricata e para o programa desenvolvido.

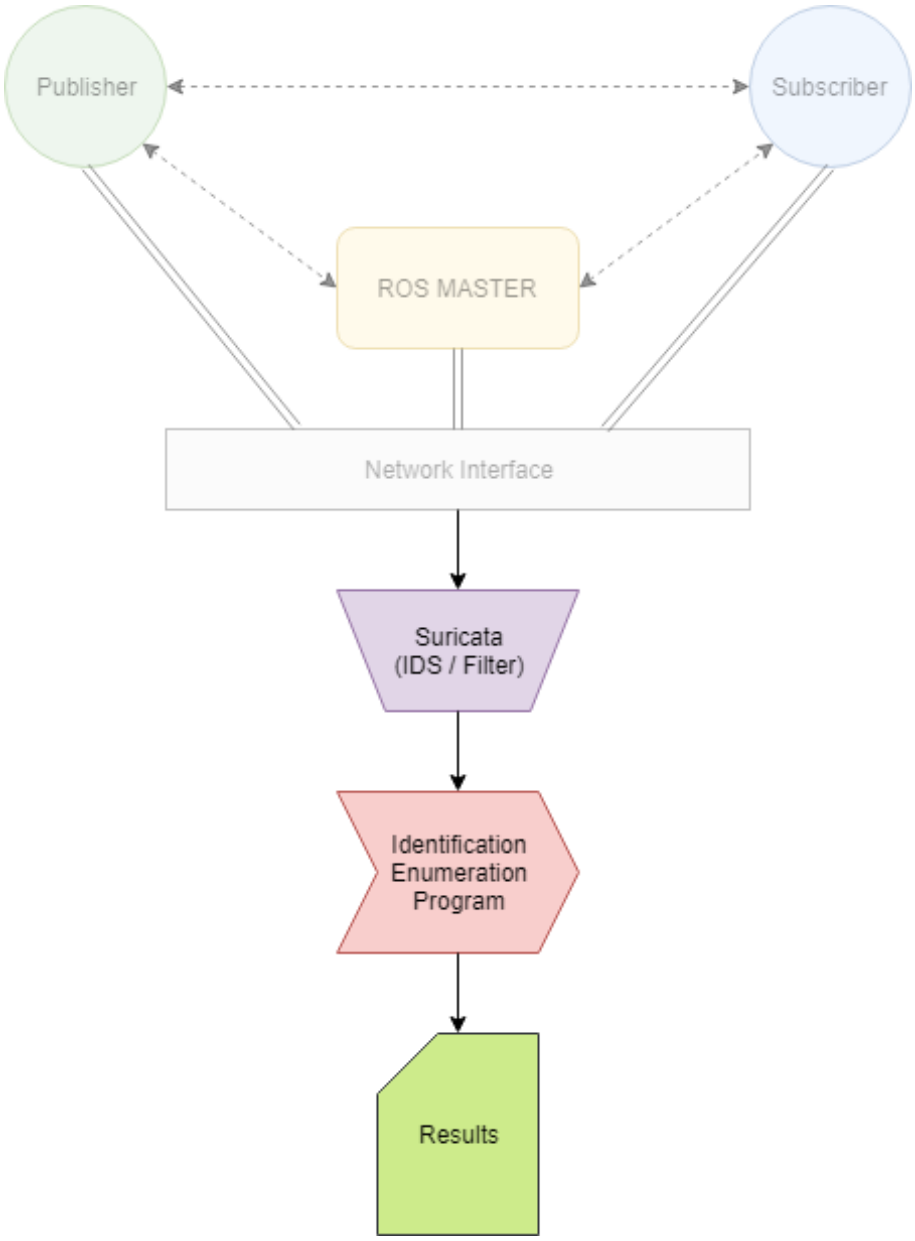

Figura 4.1: Diagrama da Solução Proposta

4.1 Suricata

Após concluída a instalação e configuração desta ferramenta, descrita na secção A.2 do capitulo A, formam criadas regras simples com o intuito de filtrar apenas as mensagens relevantes. Estas regras foram implementadas num ficheiro com o nome 'rules.rules' situado no directório das regras do suricata, que por norma, pode ser encontrado em:

```
/etc/suricata/rules
```

As regras são definidas por três partes:

- ***Action***: Este é o primeiro segmento da regra e tem como função definir a ação que será tomada pela ferramenta. A ferramenta pode deixar passar, bloquear ou apenas alertar quando informação corresponde com a regra. As opções são *pass*, *reject*, *drop* ou *alert*.
- ***Header***: Este segmento indica qual o protocolo e quais os endereços de origem e destino pretendidos.
- ***Rule Options***: Por último, este segmento pode conter variadas opções de filtragem de conteúdo e que mensagens são publicadas na ocorrência de tal conteúdo.

Uma regra exemplo pode ser visualizada na figura 4.2, onde se consegue distinguir as três partes da regra, *Action* a vermelho, *Header* a verde e *Rule Options* a azul.

```
alert tcp 192.168.1.1 1024 - > 172.16.1.1 80 (msg:"ET DOS
Possible SolarWinds TFTP Server Read Request Denial Of Service
Attempt"; content:"|00 01 01|"; depth:3; content:"NETASCII";
reference:url,www.exploit-db.com/exploits/12683/;
reference:url,doc.emergingthreats.net/2011673;
classtype:attempted-dos; sid:2011673; rev:3;)
```

Figura 4.2: Regra Exemplo do Suricata

O protocolo alvo é o protocolo de comunicação XML-RPC, sumariado em 2.1.2.1, por ser o protocolo utilizado pela ferramenta ROS para o registo de nós, serviços, tópicos e negociação de comunicações, conseguindo assim obter informação sobre os elementos da rede e os seus serviços. Este é composto por pedidos e respostas em que os pedidos contêm o nome dos métodos a que cada mensagem corresponde. Desta forma é possível filtrar e alertar quando as mensagens do protocolo em questão aconteceram.

Foram então criadas várias regras, uma para cada um dos tipos de mensagem mais relevantes, como se pode observar na figura 4.3.

```
alert tcp any any -> any any (msg:"getPid"; sid:1; content:"getPid";)
alert tcp any any -> any any (msg:"registerService"; sid:2; content:"registerService";)
alert tcp any any -> any any (msg:"registerPublisher"; sid:3; content:"registerPublisher";)
alert tcp any any -> any any (msg:"registerSubscriber"; sid:4; content:"registerSubscriber";)
alert tcp any any -> any any (msg:"publisherUpdate"; sid:5; content:"publisherUpdate";)
alert tcp any any -> any any (msg:"requestTopic"; sid:6; content:"requestTopic";)
```

Figura 4.3: Regra Utilizadas

Apesar das diversas ações que as regras disponibilizam, estas apenas têm de notificar a ocorrência de eventos, pelo que em todas foi definido a parte *Action* em modo *Alert*, como se pode verificar pela figura anterior.

A parte *Header* começa por identificar o protocolo de transporte, que se definiu como TCP, seguindo dos endereços IP e portos de origem e destino. Como o objetivo é capturar todas as comunicações relevantes, foi atribuído 'any' aos endereços e aos portos de origem e destino.

A última parte, *Rule Options*, contém apenas a o conteúdo pretendido no parâmetro 'content', a mensagem respetiva em 'msg' e o parâmetro obrigatório 'sid' que é o número de identificação da regra.

Após concluída a a configuração do Suricata e das suas regras, esta está pronta a ser utilizada. Como esta foi ser utilizada em modo de leitura de captura de pacotes, apenas é necessário invocar o seguinte comando num terminal, substituindo 'pcap-file-path' pelo ficheiro de captura de pacotes a analisar:

```
suricata −r 'pcap−file−path'
```

Após sucedida a execução, a informação filtrada é gravada num ficheiro de *output* criado pela ferramenta. Por norma, este pode ser encontrado em:

```
/var/log/suricata/fast.log
```

Um exemplo do resultado obtido após a execução da ferramenta pode ser observado na figura 4.4. Cada linha representa um evento com a informação detetada por uma das regras criadas, onde o primeiro argumento é a data e hora da interceção da mensagem, o parâmetro 'sid' seguido do conteudo do parametro 'msg' e, no fim da linha, o protocolo e os endereços IP e os seus portos de origem e destino, respetivamente.

```
06/05/2019-13:23:14.534026  [**] [1:4:0] registerPublisher [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40090 -> 127.0.0.1:11311
06/05/2019-13:23:14.599258  [**] [1:8:0] publisherUpdate [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:39368 -> 127.0.1.1:36215
06/05/2019-13:23:22.155705  [**] [1:4:0] registerPublisher [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40106 -> 127.0.0.1:11311
06/05/2019-13:23:14.701151  [**] [1:9:0] requestTopic [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:52994 -> 127.0.1.1:41363
06/05/2019-13:23:22.159737  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40110 -> 127.0.0.1:11311
06/05/2019-13:23:22.164726  [**] [1:6:0] registerSubscriber [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40116 -> 127.0.0.1:11311
06/05/2019-13:23:22.211742  [**] [1:8:0] publisherUpdate [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:39368 -> 127.0.1.1:36215
06/05/2019-13:23:22.313073  [**] [1:9:0] requestTopic [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:44060 -> 127.0.1.1:39923
06/05/2019-13:23:29.495355  [**] [1:8:0] publisherUpdate [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:53016 -> 127.0.1.1:41363
06/05/2019-13:23:29.500965  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40132 -> 127.0.0.1:11311
06/05/2019-13:23:29.595973  [**] [1:8:0] publisherUpdate [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:39368 -> 127.0.1.1:36215
06/05/2019-13:23:29.504432  [**] [1:9:0] requestTopic [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:50942 -> 127.0.1.1:44869
06/05/2019-13:23:29.697400  [**] [1:9:0] requestTopic [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:50954 -> 127.0.1.1:44869
06/05/2019-13:23:14.539812  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40096 -> 127.0.0.1:11311
06/05/2019-13:23:14.542876  [**] [1:6:0] registerSubscriber [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40100 -> 127.0.0.1:11311
06/05/2019-13:23:22.161576  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40112 -> 127.0.0.1:11311
06/05/2019-13:23:29.496196  [**] [1:8:0] publisherUpdate [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:44068 -> 127.0.1.1:39923
06/05/2019-13:23:29.505607  [**] [1:9:0] requestTopic [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:50944 -> 127.0.1.1:44869
06/05/2019-13:23:14.537621  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40094 -> 127.0.0.1:11311
06/05/2019-13:23:29.397524  [**] [1:4:0] registerPublisher [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40122 -> 127.0.0.1:11311
06/05/2019-13:23:29.497634  [**] [1:4:0] registerPublisher [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40128 -> 127.0.0.1:11311
06/05/2019-13:23:29.502727  [**] [1:2:0] registerService [**] [Classification: (null)] [Priority: 3] {TCP} 127.0.0.1:40134 -> 127.0.0.1:11311
```

Figura 4.4: Exemplo do ficheiro fast.log

4.2 Programa Desenvolvido

Terminada a execução da ferramenta Suricata, a informação essencial pode ser analisada pelo pequeno programa desenvolvido. Este programa começa por abrir o ficheiro com o *output* do Suricata, inicia um contador 'counter' para contagem das linhas e inicia várias listas para memorizar os dados que serão extraídos do ficheiro. A função de cada lista é a seguinte:

- **timeStampList** (string)**:** guarda a data e hora de cada evento
- **timeInSecList** (float)**:** guarda a data e hora de cada evento em segundos
- **srcList** (string)**:** guarda o endereço IP e porto origem de cada evento

- **destList** (string)**:** guarda o endereço IP e porto destino de cada evento
- **contentList** (string)**:** guarda o tipo de evento de cada evento

Listing 4.1: Inicialização de Variáveis

```
f = open("/var/log/suricata/fast.log")
counter = 0
timeStampList = list()
timeInSecList = list()
srcList = list()
destList = list()
contentList = list()
```

O programa pode ser dividido em 3 partes, representadas na figura 4.5. A primeira parte foca-se na leitura e ordenação de informação, a segunda tem como função obter os endereços IP e portos de entidades relevantes ao problema, sendo estas, o ROS MASTER, os publicadores e subscritores e a terceira parte interpreta cada evento e cria uma cronologia dos eventos numa linguagem mais compreensível.

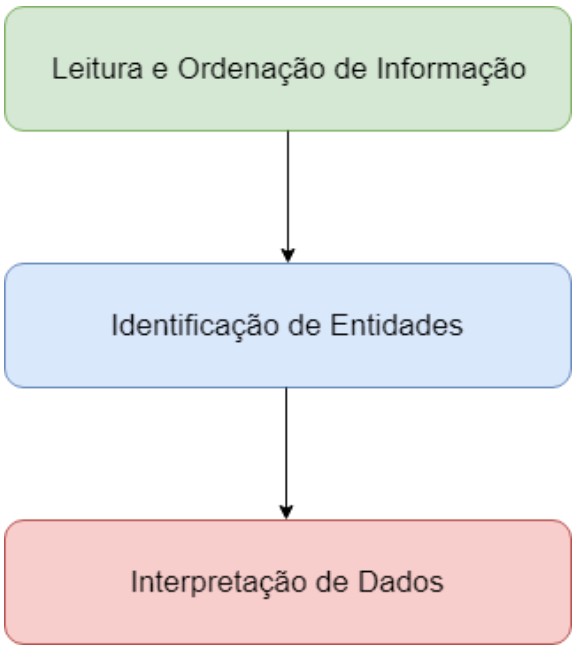

Figura 4.5: Diagrama das Partes do Programa

4.2.1 Leitura e Ordenação de Informação

A primeira parte do programa consiste num ciclo de leitura do ficheiro aberto anteriormente. Este ciclo tem duas funções. A função de leitura, realizada na parte inicial do ciclo e a função de ordenação, executada posteriormente à leitura.

O ciclo inicia com a leitura de uma linha do ficheiro, na parte de leitura, lista 4.2, essa linha é dividida com a função 'split()', que permite dividir a linha em secções separadas pelo caractere

que é introduzido como argumento da função, e guardar cada secção relevante em variáveis temporárias. Nestas variáveis é guardado informação como o instante temporal, o tipo de evento, o endereço de origem e o de destino.

Listing 4.2: Leitura do Ficheiro

```
for line in f:
    counter += 1          # Counter
    timeStamp = line.split(' ')[0]        # TimeStamp
    year = line.split('-')[0].split('/')[2]       # Year
    month = line.split('-')[0].split('/')[1]      # Month
    day = line.split('-')[0].split('/')[0]       # Day
    hour = line.split('-')[1].split(':')[0]       # Hour
    minute = line.split(':')[1]        # Minute
    second = (line.split(':')[2].split(' ')[0])      # Second
    content = ("{:<20}".format(line.split(' ')[4])).replace("'","").
        strip()  # Method Name
    srcDest = line.split('{TCP}')[1]
    src = srcDest.split('->')[0].replace("'","").strip()     # Source
    dst = srcDest.split('->')[1].replace("'","").strip()     #
        Destination
```

O ano, mês, dia, hora, minuto e segundo são também armazenados em variáveis temporárias para facilitar o calculo do tempo em segundos, que é calculado logo a seguir à leitura e armazenamento de dados, e guardado noutra variável temporária, lista 4.3.

Listing 4.3: Calculo do Instante Temporal em Segundos

```
timeCalc = float(year)*366.0*32.0*24.0*60.0*60.0 +
          float(month)*32.0*24.0*60.0*60.0 +
          float(day)*24.0*60.0*60.0 +
          float(hour)*60.0*60.0 +
          float(minute)*60.0 +
          float(second)
```

Cada ciclo termina com a transferência dos dados das variáveis temporárias para as listas criadas anteriormente. Se as listas estiverem vazias, os dados são armazenados através da função 'append()' na primeira condição da lista 4.4. Caso esta já tenha conteúdo, o programa entra num ciclo de ordenação por ordem temporal no qual itera por cada índice da lista 'timeInSecList' e insere, com a função 'insert()', os dados na primeira posição da lista onde o seu espaço temporal em segundos for menor que o dos dados que ocupam essa posição no momento, incrementando o índice de todos os eventos que se seguem. Se a lista for percorrida na sua totalidade sem a inserção dos dados, estes são adicionados ao fim da lista.

Listing 4.4: Ordenação dos Eventos

```
if len(timeInSecList) == 0:
          timeStampList.append(timeStamp)
```

```
            timeInSecList.append(timeCalc)
            srcList.append(src)
            destList.append(dst)
            contentList.append(content)
      else:
            for i in range(0, len(timeInSecList)):
                  if timeCalc < timeInSecList[i]:
                        timeStampList.insert(i, timeStamp)
                        timeInSecList.insert(i, timeCalc)
                        srcList.insert(i, src)
                        destList.insert(i, dst)
                        contentList.insert(i, content)
                        break
                  elif i == len(timeInSecList) - 1:
                        timeStampList.append(timeStamp)
                        timeInSecList.append(timeCalc)
                        srcList.append(src)
                        destList.append(dst)
                        contentList.append(content)
                        break
```

4.2.2 Identificação de Entidades

Antes de iniciar a interpretação dos eventos, é crucial identificar as entidades envolvidas nas comunicações capturadas. Apesar da ferramenta Suricata restringir a quantidade de informação que pode ser transferida no processo de filtragem dos pacotes de comunicação, é possível realizar a identificação de destas entidades por reconhecimento de padrões nas comunicações através da deteção de conteúdo especifico e espectável, não permitindo a extração de informação mais dinâmica, neste caso, como nomes de serviços ou tópicos.

Esta parte é composta por um ciclo percorre a lista de eventos na sua totalidade e que recorre a alguns padrões temporais e ao nome de alguns métodos para identificar cada entidade. Os padrões utilizados são os seguintes:

- **ROS MASTER**: qualquer registo ou publicação é feito para o ROS MASTER, portanto, pedidos com métodos como 'registerPublisher', 'registerSubscriber' ou 'registerService' são sempre feitos para o endereço de destino do ROS MASTER, este apresenta dois endereços, um endereço principal para atividades de registo e outro que é utilizado para subscrever a informação dos publicadores e subscritores.

- **Publicadores**: A identificação de publicadores é um pouco mais complexa pois implica algum conhecimento de como o registo destas entidades é feito. Cada operação de registo de um publicador contém mensagens de registo do tópico, registo do próprio como nó da rede, registo de serviços e pedidos ou envios de actualização de informação dos seus tópicos

ou serviços e pedidos. Este padrão apresenta sempre dois 'registerPublish', um para o tópico que irá fornecer e outro para o nó.

- **Subscritores**: A identificação de subscritores é um pouco semelhante à dos publicadores. A diferença é que um subscritor regista-se como um nó da rede, utilizando o método 'registerPublish' mas subscreve a tópicos com o método 'registerSubscriber', permitindo a distinção entre os eventos de registo de um publicador e de um subscritor.

Antes de iniciar o ciclo são iniciadas as variáveis, observável na lista 4.5:

- **'MasterIP'**: endereço principal do ROS MASTER
- **'MasterSubToLogInfoIP'**: endereço de subscrição a informação dos nós
- **'SubscriberIP'**: lista de endereços de subscritores
- **'PublisherIP'**: lista de endereços de publicadores
- **'currentRegistration'**: memoriza qual tipo de registo atual
- **'currentTime'**: memoriza o espaço temporal do evento inicial de um registo em segundos
- **'eventPeriod'**: constante de tempo máximo da duração de um registo em segundos

Listing 4.5: Inicialização de Variáveis

```
MasterIP = "ROS Master IP Not Found"
MasterSubToLogInfoIP = "Sub To Log Info IP Not Found"
SubscriberIP = list()
PublisherIP = list()

currentRegistration = "No registration"
publisherTimeInSecList = list()
subscriberTimeInSecList = list()
currentTime = 0.0
eventPeriod = 0.5
```

Este ciclo percorre a lista de eventos e procura os eventos 'registerPublisher' ou 'getPid', 'publisherUpdate' e 'requesTopic', apresentado na lista 4.6.

Listing 4.6: Visualização do Ciclo de Identificação de Entidades

```
for i in range(0, counter):

      if timeInSecList[i] - currentTime > eventPeriod:
            currentRegistration = "No registration"
```

```
if contentList[i] == "registerPublisher" or contentList[i] == "
    getPid":
      if MasterIP == "ROS Master IP Not Found":
            MasterIP = destList[i]

      if currentRegistration == "registerPublisher":
            currentRegistration = "registerPublisher2"
      else:
            currentTime = timeInSecList[i]
            for j in range(1, 10):
                  if i + j >= counter:
                        break

                  if contentList[i + j] == "registerPublisher" and
                      timeInSecList[i + j] - currentTime <
                      eventPeriod:
                        publisherTimeInSecList.append(timeInSecList[
                            i + j])
                        currentRegistration = "registerPublisher"
                        break
                  elif contentList[i + j] == "registerSubscriber"
                      and timeInSecList[i + j] - currentTime <
                      eventPeriod:
                        subscriberTimeInSecList.append(timeInSecList
                            [i + j])
                        currentRegistration = "registerSubscriber"
                        break

elif contentList[i] == "publisherUpdate":
      if destList[i] not in SubscriberIP and MasterSubToLogInfoIP ==
           "Sub To Log Info IP Not Found":
            MasterSubToLogInfoIP = destList[i]

elif contentList[i] == "requestTopic":
      IP = destList[i]
      if IP not in SubscriberIP and IP not in PublisherIP:
            if currentRegistration == "registerSubscriber":
                  SubscriberIP.append(IP)
            elif currentRegistration == "registerPublisher" or
                currentRegistration == "registerPublisher2":
                  PublisherIP.append(IP)
```

4.2.2.1 Identificação de Inicio de um Registo e ROS MASTER

Se for encontrado 'registerPublisher' ou 'getPid' existe a possibilidade de registo de uma entidade. Para confirmar que se trata de um registo de um publicador ou subscritor, é iniciada uma pequena pesquisa nos eventos seguintes pelo método 'registerSubscriber' ou por um novo 'registerPublisher'. Caso encontrado um destes dentro do intervalo de tempo da duração máxima de um registo, significa que existe um registo de um subscritor ou de um publicador, respetivamente. Como o registo é feito para o ROS MASTER e caso o seu endereço ainda não tenha sido encontrado, o endereço de destino deste evento é armazenado como o endereço de ROS MASTER.

4.2.2.2 Identificação do Endereço Subscritor ROS MASTER

O endereço pelo qual o ROS MASTER obtém informação sobre o estado dos nós, pode ser encontrado no endereço de destino do pedido da ligação com o método 'publishUpdate'. Este método serve para atualizar informação de serviços ou tópicos para subscritores, pelo que o primeiro subscritor a ser informado é o ROS MASTER.

4.2.2.3 Identificação de Publicadores e Subscritores

O método 'requesTopic' é um pedido de informação sobre o tópicos ou serviços direcionado ao publicador destes. No momento de registo de um nó, os nós subscritores, ROS MASTER inclusive, são informados e enviam um pedido através deste método. Caso este tenha sido detetado, é possível encontrar o endereço do nó em processo de se registar na rede. Este apenas é adicionado a uma das listas de subscritores ou publicadores caso ainda não tenha sido encontrado.

4.2.3 Interpretação de Dados

Por fim, depois de concretizadas a leitura de dados, ordenação e identificação de entidades, o programa pode finalmente interpretar as comunicações capturadas. Este secção do programa volta a percorrer a lista de eventos e imprime uma interpretação de cada evento. O programa interpreta os seguintes métodos:

- **'registerPublisher'**: Registo de um nó publicador ou subscritor. Pode também ser um registo de tópico pertencente a um publicador.
- **'registerSubscriber'**: Registo de subscrição a um tópico.
- **'getPid'**: Pedido da identificação de um processo.
- **'registerService'**: Registo de um serviço de um nó publicador ou subscritor.
- **'publisherUpdate'**: Atualização de informação de tópicos ou serviços direcionada aos subscritores.
- **'requestTopic'**: Pedido de informação de subscritores a publicadores sobre tópicos.

Esta parte começa por iniciar algumas variáveis auxiliares e inicia o ciclo que percorre a lista de eventos. No inicio deste, é verificado se existe algum tipo de registo. Isto é feito com uma série de condições semelhantes às condições de identificação de publicadores e subscritores da parte anterior do programa. Como podemos observar em 4.7, se o método atual for 'registerPublisher' ou 'getPid', inicia-se um processo de procura nos seguintes eventos de um segundo 'registerPublisher' ou 'registerSubscriber' dentro do intervalo máximo de registo, que se verificado, assinala a presença de um registo. Se o intervalo máximo expirar os eventos atuais não pertencem a um registo de um publicador ou subscritor. Logo de seguida, é iniciado o processo de interpretação de cada método resumido anteriormente.

Listing 4.7: Identificação de Registo Atual

```
for i in range(0, counter):

      if timeInSecList[i] - currentTime > eventPeriod:
            currentRegistration = "No registration"

      if contentList[i] == "registerPublisher" or contentList[i] == "
          getPid":
            if currentRegistration != "registerPublisher2":
                  currentTime = timeInSecList[i]

            for j in range(1, 10):
                  if i + j >= counter:
                        break
                  if contentList[i + j] == "registerPublisher" and
                      timeInSecList[i + j] - currentTime < eventPeriod:
                        if contentList[i] == "getPid":
                              currentRegistration = "registerPublisherPid"
                        elif contentList[i] == "registerPublisher":
                              currentRegistration = "registerPublisher"
                        break
                  elif contentList[i + j] == "registerSubscriber" and
                      timeInSecList[i + j] - currentTime < eventPeriod:
                        if contentList[i] == "getPid":
                              currentRegistration = "registerSubscriberPid
                                  "
                        currentRegistration = "registerSubscriber"
                        break
```

4.2.3.1 Interpretação de 'registerPublisher'

Na presença deste método, ou por vezes 'getPid' em casos onde processos já estejam iniciados, a sua interpretação pode ser um pouco complexa devido à sua versatilidade. Em caso de registo de um publicador, este apresenta dois métodos 'registerPublisher', dos quais o primeiro publica

o tópico e o segundo regista-se como um nó publicador. Se houver um 'getPid' seguido de um 'registerPublisher', significa que o publicador já estava registado e apenas publicou o tópico. Se este método aparecer seguido de um 'registerSubscriber', houve uma publicação de um nó foi feita pelo subscritor.

Em 4.8, pode-se observar as vários funcionalidades que este método pode tomar, incluindo a publicação anónima.

Listing 4.8: Código de 'registerPublisher'

```
if contentList[i] == "registerPublisher":
            if currentRegistration == "registerSubscriber":
                  print("")
                  subscriberCounter += 1
                  currentRegistration = "registerSubscriber"
                  print(timeStampList[i] + ": Subscriber"),
                  print(subscriberCounter + 1),
                  print("registered as a node to MASTER")
            elif currentRegistration == "registerPublisher":
                  print("")
                  publisherCounter += 1
                  currentRegistration = "registerPublisher"
                  print(timeStampList[i] + ": Publisher"),
                  print(publisherCounter + 1),
                  print("published a topic to MASTER")
                  currentRegistration = "registerPublisher2"
            elif currentRegistration == "registerPublisher2":
                  print(timeStampList[i] + ": Publisher"),
                  print(publisherCounter + 1),
                  print("registered as a node to MASTER")
            elif currentRegistration == "registerPublisherPid":
                  print(timeStampList[i] + ": Publisher"),
                  print(publisherCounter + 1),
                  print("published a topic to MASTER")
            elif currentRegistration == "No registration":
                  print(timeStampList[i] + ": Publisher was registered to
                      MASTER")
```

4.2.3.2 Interpretação de 'getPid'

Esta função é utilizada para pedidos de identificação de processos ou entidades. Se alguma entidade já esteja registada, é feito um pedido de identificação do seu processo. Na lista 4.9 é apresentado o código equivalente a esta interpretação.

Listing 4.9: Código de 'getPid'

```
elif contentList[i] == "getPid":
```

```
            if currentRegistration == "registerPublisherPid":
                  print("")
                  publisherCounter += 1
                  print(timeStampList[i] + ": Publisher"),
                  print(publisherCounter + 1),
                  print("requested a process ID to MASTER")
            elif currentRegistration == "registerSubscriberPid":
                  print("")
                  subscriberCounter += 1
                  print(timeStampList[i] + ": Subscriber"),
                  print(subscriberCounter + 1),
                  print("requested a process ID to MASTER")
            elif currentRegistration == "No registration":
                  print(timeStampList[i] + ": A process ID was requested
                     to MASTER")
```

4.2.3.3 Interpretação de 'registerSubscriber'

Este método é o mais simples de interpretar, como se pode visualizar em 4.10, pois a sua invocação apenas é feita quando um nó subscreve a um tópico.

Listing 4.10: Código de 'registerSubscriber'

```
elif contentList[i] == "registerSubscriber":
            print(timeStampList[i] + ": Subscriber"),
            print(subscriberCounter + 1),
            print("published a topic to MASTER to MASTER")
```

4.2.3.4 Interpretação de 'registerService'

Sempre que um serviço é registado, esta função é utilizada para o fazer. Nós publicadores e subscritores registam os seus serviços através desse método quando são registados na rede. A lista 4.11 apresenta as condições para cada um dos tipos de nós e uma terceira para quando um serviço é registado fora de padrões de registo de nós.

Listing 4.11: Código de 'registerService

```
elif contentList[i] == "registerService":
            if currentRegistration == "registerSubscriber" or
               currentRegistration == "registerSubscriberPid":
                  print(timeStampList[i] + ": Subscriber"),
                  print(subscriberCounter + 1),
                  print("registered a service to MASTER")
            elif currentRegistration == "registerPublisher" or
               currentRegistration == "registerPublisher2" or
               currentRegistration == "registerPublisherPid":
```

```
                print(timeStampList[i] + ": Publisher"),
                print(publisherCounter + 1),
                print("registered a service to MASTER")
        else:
                print("")
                print(timeStampList[i] + ": Service registered to MASTER
                    ")
```

4.2.3.5 Interpretação de 'publisherUpdate'

Pedidos com este método tem como objetivo informar a existência de informação atualizada de serviços ou tópicos. Este método é sempre utilizado por fornecedores de serviços para manter os subscritores informados do estado de cada serviço. Este é utilizado por publicadores após ter feito atualizações nos seus tópicos. Como mencionado anteriormente, cada nó fornece serviços de informação do seu estado ao ROS MASTER pelo que também é utilizado por subscritores a tópicos para informar o ROS MASTER do seu estado. O código utilizado para esta interpretação é observável em 4.12.

Listing 4.12: Código de 'publisherUpdate'

```
elif contentList[i] == "publisherUpdate":
        if currentRegistration == "registerSubscriber":
            if destList[i] == MasterSubToLogInfoIP:
                print(timeStampList[i] + ": Subscriber"),
                print(subscriberCounter + 1),
                print("updated information of a service to ROS
                    MASTER")
        elif currentRegistration == "registerPublisher" or
            currentRegistration == "registerPublisher2" or
            currentRegistration == "registerPublisherPid":
            if destList[i] == MasterSubToLogInfoIP:
                print(timeStampList[i] + ": Publisher"),
                print(publisherCounter + 1),
                print("updated information of a service to ROS
                    MASTER")
            if destList[i] in SubscriberIP:
                for j in range(0, len(SubscriberIP)):
                    if destList[i] == SubscriberIP[j]:
                        print(timeStampList[i] + ": Publisher"
                            ),
                        print(publisherCounter + 1),
                        print("updated topic information to
                            subscriber"),
                        print(j + 1)
        else:
```

```
                print(timeStampList[i] + ": Updated information of a
                    service to ROS MASTER")
```

4.2.3.6 Interpretação de 'requestTopic'

O 'requestTopic' é um pedido de informação de tópicos ou serviços realizado aos publicadores por parte dos subscritores. Como se pode observar em 4.13, este método pode ter subscritores como destino pois estes também se comportam como publicadores ao fornecer serviços ao ROS MASTER, pelo que este pode realizar pedidos de informação estes.

Listing 4.13: Código de 'requestTopic'

```
elif contentList[i] == "requestTopic":
            if destList[i] in SubscriberIP:
                  for j in range(0, len(SubscriberIP)):
                        if destList[i] == SubscriberIP[j]:
                              print(timeStampList[i] + ": ROS MASTER
                                  requested a topic to subscriber"),
                              print(j + 1)
            elif destList[i] in PublisherIP:
                  for j in range(0, len(PublisherIP)):
                        if destList[i] == PublisherIP[j]:
                              print(timeStampList[i] + ": A subscriber or
                                  ROS MASTER requested a topic to publisher
                                  "),
                              print(j + 1)
```

Capítulo 5

Resultados

Neste capítulo irão ser apresentados os resultados dos testes realizados ao programa desenvolvido. De forma a diversificar os testes, foram utilizados dois simuladores e um conjunto de programas que interagem entre si. Nas seguintes secções serão apresentados de forma sucinta a funcionalidade dos programas e simuladores e as suas comunicações processadas pelo programa desenvolvido.

Para obter os resultados, as comunicações de cada teste são capturadas pela ferramenta Wireshark e guardadas para um ficheiro. Esse ficheiro é filtrado com a ferramenta Suricata usando o seguinte comando num terminal:

Listing 5.1: Código de 'requestTopic'

```
suricata -r 'pcap-file-path'
```

Este comando executa a filtragem e guarda a informação filtrada que é utilizada pelo programa desenvolvido para analisar as comunicações, bastando executar o programa com o seguinte comando:

Listing 5.2: Código de 'requestTopic'

```
python ros-comms-interpret.py
```

5.1 Publicador e Subscritor

Para realizar os testes, foram criados dois programas que comunicam entre si. Estes foram desenvolvidos de forma a que um se comporte como um publicador e outro como um subscritor. O publicador tem como função enviar mensagens através da publicação de um tópico ao qual o subscritor subscreve e recebe as mensagens do publicador. O publicador 'talker' e outro que subscrição a esse tópico e receção das mensagens chamado 'listener'.

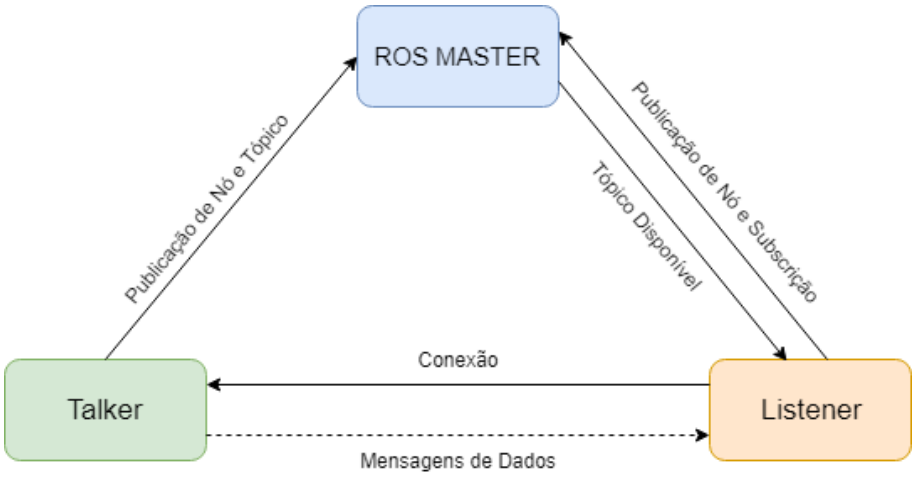

Figura 5.1: Diagrama Simplificado do ‘talker’ e ‘listener’

5.1.1 Código Talker e Listener

Como se pode observar em 5.3, o executável ‘talker’ apenas possui uma uma função simples chamada ‘talker()’. Esta função publica um tópico com o nome ‘chatter’ na primeira linha, inicia um nó na segunda linha com o nome da função que se torna o publicador do tópico anteriormente publicado e inicia um ciclo, com a frequência atribuída na função ‘Rate()’, onde publica mensagens para quem esteja subscrito ao tópico através da função ‘publish()’ e informa o *broker* pela função ‘loginfo()’.

O executável ‘listener’ possui duas funções, verificável em 5.4. Uma com o nome ‘listener()’ que publica um nó com o nome da função e subscreve esse nó ao mesmo tópico publicado pelo nó ‘talker’. A segunda função é uma função ‘calkback’. Esta função é invocada quando recebe alguma informação. Se o nó publicador envia uma mensagem e o nó subscritor recebe, esta função apenas recebe essa informação e informa o ‘broker’ que a recebeu e o que recebeu.

Listing 5.3: Código do publicador Talker

```
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(1) # 10hz -> default
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
```

```
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Listing 5.4: Código do subscritor Listener

```
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + 'I heard %s', data.data)

def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber('chatter', String, callback)

    rospy.spin()

if __name__ == '__main__':
    listener()
```

5.1.2 Resultados de Talker e Listener

Foram realizados vários testes diferentes com os ficheiros executáveis *talker* e *listener*, onde o número e a ordem de execução foram trocados de forma a obter diferentes resultados.

5.1.2.1 Um Listener Seguido de um Talker

Neste exemplo, um subscritor *listener* e um publicador *talker* foram registados pela ordem respetiva. Após o procedimento de análise, programa recebeu as ligações apresentadas na figura 5.2.

```
127.0.0.1:39556 called registerPublisher        to      127.0.0.1:11311 at      06/03/2019-13:50:45.427479
127.0.0.1:39560 called registerService          to      127.0.0.1:11311 at      06/03/2019-13:50:45.431560
127.0.0.1:39562 called registerService          to      127.0.0.1:11311 at      06/03/2019-13:50:45.435411
127.0.0.1:39566 called registerSubscriber       to      127.0.0.1:11311 at      06/03/2019-13:50:45.438749
127.0.0.1:37296 called publisherUpdate          to      127.0.1.1:42625 at      06/03/2019-13:50:45.470498
127.0.0.1:35900 called requestTopic             to      127.0.1.1:43869 at      06/03/2019-13:50:45.572417
127.0.0.1:39572 called registerPublisher        to      127.0.0.1:11311 at      06/03/2019-13:50:49.482709
127.0.0.1:35906 called publisherUpdate          to      127.0.1.1:43869 at      06/03/2019-13:50:49.517790
127.0.0.1:47870 called requestTopic             to      127.0.1.1:36165 at      06/03/2019-13:50:49.565992
127.0.0.1:39582 called registerPublisher        to      127.0.0.1:11311 at      06/03/2019-13:50:49.582673
127.0.0.1:39586 called registerService          to      127.0.0.1:11311 at      06/03/2019-13:50:49.588019
127.0.0.1:39588 called registerService          to      127.0.0.1:11311 at      06/03/2019-13:50:49.589541
127.0.0.1:37296 called publisherUpdate          to      127.0.1.1:42625 at      06/03/2019-13:50:49.619078
127.0.0.1:47884 called requestTopic             to      127.0.1.1:36165 at      06/03/2019-13:50:49.721168
```

Figura 5.2: Comunicações Capturadas

Cada linha representa uma ligação onde foi invocada uma função com origem no endereço da esquerda e destino no o endereço da direita, no espaço temporal encontrado no final de cada

linha. Nessa figura é possível verificar que existem dois momentos temporais onde houve um agregado de mensagens trocadas, um cerca do segundo 45 e outro por volta do segundo 49. É possível distinguir cada um dos momentos como o registo de cada um dos nós. O momento inicial apresenta um 'registerPublisher' e um 'registerSubscriber' e representa a publicação do nó e a subscrição a um tópico. No momento seguinte apresenta dois 'registerPublisher' pelo que pode ser atribuído à atividade do publicador *talker*.

```
ROS MASTER listens on IP address and port 127.0.0.1:11311
ROS MASTER subscribes to log information from the nodes and listens on IP address and port 127.0.1.1:42625
Publisher 1 listens on IP address and port 127.0.1.1:36165
Subscriber 1 listens on IP address and port 127.0.1.1:43869
```

Figura 5.3: Entidades Encontradas

A partir desta lista de eventos é possível identificar os endereços das entidades envolvidas, encontrados na figura 5.3, tendo em conta os métodos utilizados e o endereço de destino. A entidade que inicia a ligação pode apresentar um endereço diferente para cada ligação. Mas o endereço de destino é obrigatoriamente estático para cada entidade que recebe um pedido.

Todo o tipo de registos é feito para o ROS MASTER pelo que é fácil de identificar o seu endereço. Após o registo de serviços ou tópicos, cada nó informa os seus subscritores da existência de informação atualizada através da função 'publisherUpdate'. Como o ROS MASTER subscreve a informação 'log' de cada nó, o endereço do processo que subscreve a estes serviços pode ser encontrado nos endereços de destino quando a entidade fornecedora do serviço atualiza informação deste para o ROS MASTER.

Os subscritores podem pedir informação sobre serviços ou tópicos aos publicadores através do método 'requesTopic'. Desta forma é possível identificar publicadores e subscritores, pois durante o seu registo o ROS MASTER pede informação a cada nó. Se no momento estiver a ser registado uma das entidades, é possível identificar o seu endereço a partir desta função chamada pelo ROS MASTER.

```
06/03/2019-13:50:45.427479: Subscriber 1 registered as a node to MASTER
06/03/2019-13:50:45.431560: Subscriber 1 registered a service to MASTER
06/03/2019-13:50:45.435411: Subscriber 1 registered a service to MASTER
06/03/2019-13:50:45.438749: Subscriber 1 published a topic to MASTER to MASTER
06/03/2019-13:50:45.470498: Subscriber 1 updated information of a service to ROS MASTER
06/03/2019-13:50:45.572417: ROS MASTER requested a topic to subscriber 1

06/03/2019-13:50:49.482709: Publisher 1 published a topic to MASTER
06/03/2019-13:50:49.517790: Publisher 1 updated topic information to subscriber 1
06/03/2019-13:50:49.565992: A subscriber or ROS MASTER requested a topic to publisher 1
06/03/2019-13:50:49.582673: Publisher 1 registered as a node to MASTER
06/03/2019-13:50:49.588019: Publisher 1 registered a service to MASTER
06/03/2019-13:50:49.589541: Publisher 1 registered a service to MASTER
06/03/2019-13:50:49.619078: Publisher 1 updated information of a service to ROS MASTER
06/03/2019-13:50:49.721168: A subscriber or ROS MASTER requested a topic to publisher 1
```

Figura 5.4: Interpretação dos Eventos

Finalmente, depois da identificação das entidades, torna-se simples identificar cada evento. O programa analisa esta informação e imprime uma interpretação compreensível dos acontecimentos, mostrado na figura 5.4. Esta mostra por ordem cronológica o registo do subscritor seguido do publicador e todas as interações entre cada identidade. Também é possível observar a interação entre o publicador e o subscritor após o registo do primeiro, onde o publicador informa o subscritor sobre o tópico no segundo 49,517 e este subscreve a este no segundo 49,721.

Através da ferramenta Wireshark é possível validar a interpretação feita pelo programa. Tirando a primeira linha como exemplo, é possível validar a sua interpretação com base no pedido do subscritor e resposta do ROS MASTER encontradas nas figuras 5.5 e 5.6, respetivamente. A primeira figura mostra o envio de informação por parte do 'listener' para o ROS MASTER. Esta contém o seu nome, para onde é publicado, o tópico a publicar, e o próprio endereço da entidade. A segunda figura apresenta a reposta ao pedido da primeira visualizado na primeira figura e contém a confirmação do registo e o o endereço da entidade à qual o pedido foi direcionado.

Figura 5.5: Exemplo de um pedido de registo

Figura 5.6: Exemplo de uma resposta a um pedido de registo

5.1.2.2 Um Talker Seguido de um Listener

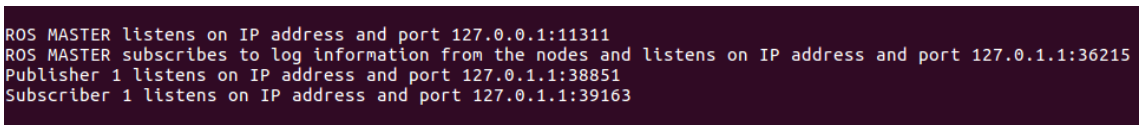

Figura 5.7: Entidades Encontradas

Este teste é semelhante ao anterior mas a ordem de registo do *listener* e *talker* é invertida, significando que o publicador foi o primeiro a ser iniciado.

Na identificação das entidades, o tipo e o número de entidades é semelhante ao do teste anterior, mas comparando as figuras 5.7 e 5.3, apenas o endereço do ROS MASTER é o mesmo e o resto são diferentes. O ROS MASTER apresenta um endereço estático pois, caso contrário, se este fosse dinâmico, cada nó não conseguiria iniciar comunicações com este.

```
06/03/2019-17:52:17.796092: Publisher 1 published a topic to MASTER
06/03/2019-17:52:17.894793: Publisher 1 registered as a node to MASTER
06/03/2019-17:52:17.898476: Publisher 1 registered a service to MASTER
06/03/2019-17:52:17.900142: Publisher 1 registered a service to MASTER
06/03/2019-17:52:17.975462: Publisher 1 updated information of a service to ROS MASTER
06/03/2019-17:52:18.076922: A subscriber or ROS MASTER requested a topic to publisher 1

06/03/2019-17:52:28.964408: Subscriber 1 registered as a node to MASTER
06/03/2019-17:52:28.969823: Subscriber 1 registered a service to MASTER
06/03/2019-17:52:28.972407: Subscriber 1 registered a service to MASTER
06/03/2019-17:52:28.975756: Subscriber 1 published a topic to MASTER to MASTER
06/03/2019-17:52:28.993668: A subscriber or ROS MASTER requested a topic to publisher 1
06/03/2019-17:52:29.064787: Subscriber 1 updated information of a service to ROS MASTER
06/03/2019-17:52:29.165603: ROS MASTER requested a topic to subscriber 1
```

Figura 5.8: Interpretação dos Eventos

Como se pode verificar na figura 5.8, o publicador foi iniciado primeiro e registando o tópico, serviços e como nó da rede. Como os endereços de origem são dinâmicos, é complicado distinguir qual a entidade que iniciou a ligação, mas observando a figura anterior é possível determinar que a ultima ligação na iniciação do publicador é iniciada pelo ROS MASTER, pois o publicador só é iniciado cerca de 10 segundos depois.

Após inicialização do subscritor, o também apresenta uma ligação onde um subscritor ou o ROS MASTER interage com o publicador. Nesta situação é também possível identificar qual das entidades iniciou a ligação. Esta é feita pelo subscritor pois é o único que ainda não tem informação atualizada sobre o tópico do publicador.

5.1.2.3 Dois Listener's Seguido de um Talker

```
ROS MASTER listens on IP address and port 127.0.0.1:11311
ROS MASTER subscribes to log information from the nodes and listens on IP address and port 127.0.1.1:36215
Publisher 1 listens on IP address and port 127.0.1.1:44869
Subscriber 1 listens on IP address and port 127.0.1.1:41363
Subscriber 2 listens on IP address and port 127.0.1.1:39923
```

Figura 5.9: Entidades Encontradas

Para aumentar um pouco mais a complexidade, foram iniciados dois subscritores *listener* seguidos de um publicador *talker*. A incrementação do número de subscritores pode ser observada na figura 5.9 pelo aparecimento de um segundo subscritor na lista de entidades descobertas.

Este exemplo é parecido com o primeiro teste 5.1.2.1 realizado, onde um publicador foi iniciado depois de um subscritor, com a diferença da ser iniciado um segundo subscritor, verificável na figura 5.10.

```
06/05/2019-13:23:14.534026: Subscriber 1 registered as a node to MASTER
06/05/2019-13:23:14.537621: Subscriber 1 registered a service to MASTER
06/05/2019-13:23:14.539812: Subscriber 1 registered a service to MASTER
06/05/2019-13:23:14.542876: Subscriber 1 published a topic to MASTER to MASTER
06/05/2019-13:23:14.599258: Subscriber 1 updated information of a service to ROS MASTER
06/05/2019-13:23:14.701151: ROS MASTER requested a topic to subscriber 1

06/05/2019-13:23:22.155705: Subscriber 2 registered as a node to MASTER
06/05/2019-13:23:22.159737: Subscriber 2 registered a service to MASTER
06/05/2019-13:23:22.161576: Subscriber 2 registered a service to MASTER
06/05/2019-13:23:22.164726: Subscriber 2 published a topic to MASTER to MASTER
06/05/2019-13:23:22.211742: Subscriber 2 updated information of a service to ROS MASTER
06/05/2019-13:23:22.313073: ROS MASTER requested a topic to subscriber 2

06/05/2019-13:23:29.397524: Publisher 1 published a topic to MASTER
06/05/2019-13:23:29.495355: Publisher 1 updated topic information to subscriber 1
06/05/2019-13:23:29.496196: Publisher 1 updated topic information to subscriber 2
06/05/2019-13:23:29.497634: Publisher 1 registered as a node to MASTER
06/05/2019-13:23:29.500965: Publisher 1 registered a service to MASTER
06/05/2019-13:23:29.502727: Publisher 1 registered a service to MASTER
06/05/2019-13:23:29.504432: A subscriber or ROS MASTER requested a topic to publisher 1
06/05/2019-13:23:29.505607: A subscriber or ROS MASTER requested a topic to publisher 1
06/05/2019-13:23:29.595973: Publisher 1 updated information of a service to ROS MASTER
06/05/2019-13:23:29.697400: A subscriber or ROS MASTER requested a topic to publisher 1
```

Figura 5.10: Interpretação dos Eventos

O processo de inicialização e registo de cada um dos subscritores é muito semelhante entre estes e entre o do primeiro exemplo. A diferença surge após o registo publicador. Comparando com o primeiro exemplo, o publicador informa cada um dos subscritores, verificável na segunda e terceira ligação a partir da sua iniciação e após informados sobre a existência de um publicador ao qual estes estão subscritos, estes interagem novamente com o seu publicador nos segundos 29.504 e 29.505. Como no exemplo inicial só existe um subscritor, o número de cada uma dessas interações entre elementos da rede diminui para metade.

5.1.2.4 Dois Talker's Seguido de um Listener

```
ROS MASTER listens on IP address and port 127.0.0.1:11311
ROS MASTER subscribes to log information from the nodes and listens on IP address and port 127.0.1.1:34025
Publisher 1 listens on IP address and port 127.0.1.1:33737
Publisher 2 listens on IP address and port 127.0.1.1:45541
Subscriber 1 listens on IP address and port 127.0.1.1:39281
```

Figura 5.11: Entidades Encontradas

Para finalizar os testes aos publicadores e subscritores criados pelos ficheiros executáveis anteriormente mencionados, foram iniciados dois publicadores e um subscritor, figura 5.11, mas desta vez o subscritor foi iniciado entre a iniciação dos dois publicadores, figura 5.12.

```
06/18/2019-15:31:21.310363: Publisher 1 published a topic to MASTER
06/18/2019-15:31:21.410167: Publisher 1 registered as a node to MASTER
06/18/2019-15:31:21.414330: Publisher 1 registered a service to MASTER
06/18/2019-15:31:21.416369: Publisher 1 registered a service to MASTER
06/18/2019-15:31:21.418161: Publisher 1 updated information of a service to ROS MASTER
06/18/2019-15:31:21.518741: A subscriber or ROS MASTER requested a topic to publisher 1

06/18/2019-15:31:30.531504: Subscriber 1 registered as a node to MASTER
06/18/2019-15:31:30.535494: Subscriber 1 registered a service to MASTER
06/18/2019-15:31:30.536969: Subscriber 1 registered a service to MASTER
06/18/2019-15:31:30.539813: Subscriber 1 published a topic to MASTER to MASTER
06/18/2019-15:31:30.547993: A subscriber or ROS MASTER requested a topic to publisher 1
06/18/2019-15:31:30.561036: Subscriber 1 updated information of a service to ROS MASTER
06/18/2019-15:31:30.666296: ROS MASTER requested a topic to subscriber 1

06/18/2019-15:31:38.468746: Publisher 2 published a topic to MASTER
06/18/2019-15:31:38.523574: Publisher 2 updated topic information to subscriber 1
06/18/2019-15:31:38.537720: A subscriber or ROS MASTER requested a topic to publisher 2
06/18/2019-15:31:38.570040: Publisher 2 registered as a node to MASTER
06/18/2019-15:31:38.574344: Publisher 2 registered a service to MASTER
06/18/2019-15:31:38.575557: Publisher 2 registered a service to MASTER
06/18/2019-15:31:38.624285: Publisher 2 updated information of a service to ROS MASTER
06/18/2019-15:31:38.725363: A subscriber or ROS MASTER requested a topic to publisher 2
```

Figura 5.12: Interpretação dos Eventos

Neste exemplo há uma mistura de comportamentos de entidades de vários dos tentes anteriores. Por exemplo, o padrão de comunicações quando o primeiro publicador e o subscritor são iniciados são equivalentes aos padrões do publicador e do subscritor iniciado no exemplo 5.1.2.2, respetivamente. Já o segundo publicador é semelhante ao publicador do exemplo 5.1.2.1.

5.2 Simulador

De forma a variar os testes efetuados, foi também utilizado um simulador para testar o programa desenvolvido. Este simulador está incorporado na ferramenta ROS e consiste num objeto contido no simulador que subscreve a um tópico chamado ‘cmd_vel’ pelo qual é possível publicar uma velocidade linear e uma angular, o que faz movimentar o objeto com a velocidade introduzida no tópico.

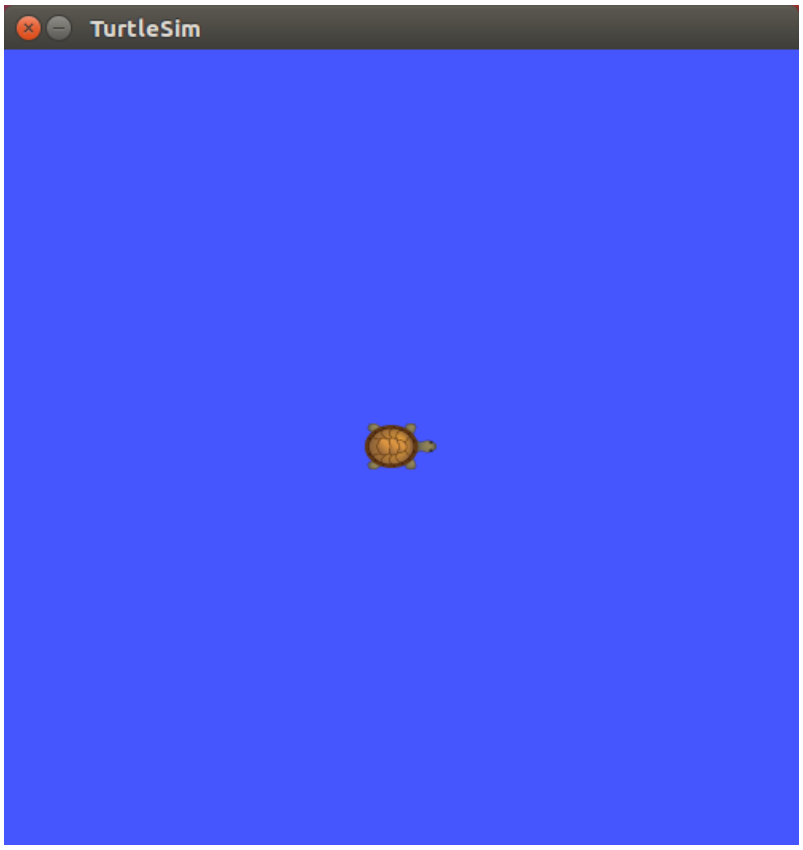

Figura 5.13: Simulador TurtleSim

Para a inicialização deste simulador basta apenas iniciar o *roscore* num terminal e executar o seguinte comando noutro terminal:

Listing 5.5: Código do subscritor Listener

```
rosrun turtlesim turtlesim_node
```

Este comando abre uma janela com um ambiente simulado e com o objeto incluído, como se pode ver na figura 5.13. O objeto inicialmente encontra-se parado mas pode ser movido através do tópico 'cmd_vel' ao qual é subscritor. A sua posição pode ser alterada publicando informação para esse tópico. Esta publicação pode ser feita publicando o seguinte comando num terceiro terminal:

Listing 5.6: Código do subscritor Listener

```
rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]'
    '[0.0, 0.0, 1.8]'
```

O comando anterior significa que houve uma publicação rostopic para o tópico '/turtle1/cmd_vel' com o tipo de mensagem 'geometry_msgs/Twist'. No fim do comando são incluídos dois vetores espaciais. O primeiro vetor indica a velocidade linear pretendida e o segundo a velocidade angular.

Com a publicação do tópico, o objeto subscritor recebe a informação e inicia o movimento com a velocidade linear e angular indicada, como se pode observar na figura 5.14.

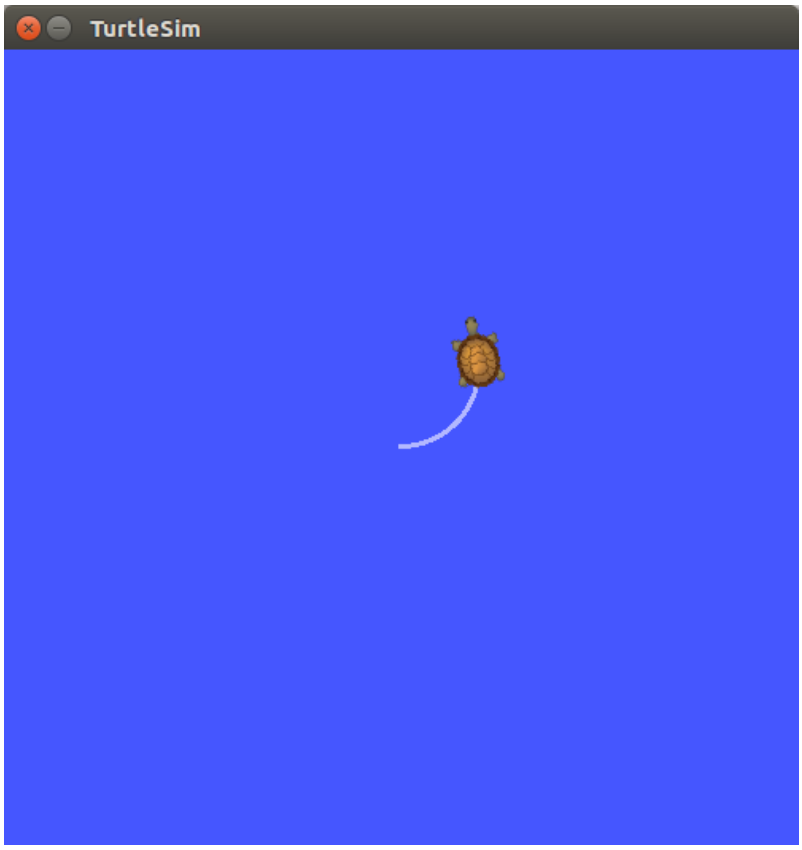

Figura 5.14: Movimentação do Objeto Após Publicação

5.2.1 Resultados Simulador

Efetuada analise do programa às comunicações de uma publicação, figura 5.15, à qual um componente do simulador está subscrito, os resultados revelam algumas diferenças com os exemplos anteriores.

```
127.0.0.1:51574 called getPid                to      127.0.0.1:11311 at     06/18/2019-17:57:15.219005
127.0.0.1:51576 called registerService       to      127.0.0.1:11311 at     06/18/2019-17:57:15.445009
127.0.0.1:51578 called registerService       to      127.0.0.1:11311 at     06/18/2019-17:57:15.446520
127.0.0.1:51580 called registerPublisher     to      127.0.0.1:11311 at     06/18/2019-17:57:15.448242
127.0.0.1:54132 called publisherUpdate       to      127.0.1.1:34989 at     06/18/2019-17:57:15.473673
127.0.0.1:53362 called requestTopic          to      127.0.1.1:34417 at     06/18/2019-17:57:15.576558
```

Figura 5.15: Comunicações Capturadas

Este exemplo o programa apenas detetou um publicador e nenhum subscritor, como se pode observar na figura 5.16, pois apenas foi realizada uma publicação. O subscritor foi iniciado na iniciação do simulador.

```
ROS MASTER listens on IP address and port 127.0.0.1:11311
ROS MASTER subscribes to log information from the nodes and listens on IP address and port 127.0.1.1:34989
Publisher 1 listens on IP address and port 127.0.1.1:34417
```

Figura 5.16: Entidades Encontradas

Analisando a interpretação do programa na figura 5.17, a publicação é semelhante à publicação no exemplo 5.1.2.2 e à primeira publicação no exemplo 5.1.2.4, mas com um pequena diferença logo na ligação inicial entre o publicador e o ROS MASTER. O método 'getPid' foi invocado na primeira comunicação entre estas entidades ao contrário do método 'registerPublisher' encontrado nos exemplos anteriores. Isto deve-se ao simples facto que o comando não publica um nó na rede, apenas publica informação de um tópico que já foi publicado pelo simulador. Esta é a razão pela utilização de um método diferente na ligação inicial e pela falta de um segundo 'registerPublisher' que normalmente aparece em publicações de nós, caso estes ainda não pertençam à rede.

```
06/18/2019-17:57:15.219005: Publisher 1 requested a process ID to MASTER
06/18/2019-17:57:15.445009: Publisher 1 registered a service to MASTER
06/18/2019-17:57:15.446520: Publisher 1 registered a service to MASTER
06/18/2019-17:57:15.448242: Publisher 1 published a topic to MASTER
06/18/2019-17:57:15.473673: Publisher 1 updated information of a service to ROS MASTER
06/18/2019-17:57:15.576558: A subscriber or ROS MASTER requested a topic to publisher 1
```

Figura 5.17: Interpretação dos Eventos

Capítulo 6

Conclusões e Trabalho Futuro

Esta dissertação tinha como objetivo a implementação de uma estrutura de captura, filtragem, identificação e enumeração de todas as comunicações relevantes numa rede implementada na plataforma Robot Operating System. A filtragem foi realizada pela ferramenta Suricata e a identificação e enumeração foi feita por uma ferramenta desenvolvida em linguagem *Python*. Apesar da captura poder ser feita pelo Suricata, foi utilizada a ferramenta Wireshark para realizar esta tarefa por uma questão de simplicidade e auxílio na análise dos pacotes capturados.

A ferramenta desenvolvida é capaz de solucionar o problema proposto através da captura das comunicações entre os vários elementos da rede. Esta consegue detetar o registo de publicadores, subscritores, serviços e tópicos através do padrão de comunicações, conseguindo também encontrar os provenientes das comunicações.

Porém, a sua habilidade de enumeração e identificação está consideravelmente restringida pela capacidade do Suricata, a ferramenta utilizada para filtrar as comunicações relevantes. Esta é condicionada pelas suas regras de filtragem que não permitem uma extração de informação de cada ligação mais detalhada, como a diferenciação dos diferentes serviços ou tópicos publicados ou subscritos, o que torna as funcionalidades da ferramenta menos objetiva. Esta ferramenta introduz um impedimento de extração de excertos de informação de forma dinâmica dos pacotes de dados capturados e filtrados. Um exemplo desta condicionante é o conhecimento de qual dos subscritores enviou um pedido ao publicador sobre informação a um tópico ou serviço ao qual é subscrito.

Um dos primeiros passos a realizar como trabalho futuro seria a substituição da ferramenta Suricata, por uma que possibilite uma identificação e enumeração mais detalhada, de forma a tornar a interpretação mais clara e concisa.

Outro passo mais avançado seria o teste da ferramenta com a introdução de elementos intrusos numa rede com uma estrutura bem definida e procurar detetar estas ou até preveni-las se possível e informar o utilizador da sua ocorrência.

Anexo A

Instalação e Configuração de Ferramentas Utilizadas

Neste capitulo serão apresentadas as ferramentas utilizadas, os procedimentos de instalação e configurações. As seguintes ferramentas foram instaladas numa maquina virtual com o sistema operativo Ubuntu com a versão 16.04.6.

A.1 Robot Operating System

Foi instalada e versão Kinetic da ferramenta Robot Operating System para uma maior compatibilidade com a versão do sistema operativo utilizado. Os próximos passos foram introduzidos no terminal com *shell bash*.

A.1.1 Instalação de Dependências

Antes de iniciar qualquer instalação, deve-se certificar que o sistema operativo aceite *software* de terceiros, neste caso, de packages.ros.org. Isto pode ser feito com a seguinte linha no terminal *bash*:

Listing A.1: Código do subscritor Listener

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc
    ) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Deve-se também adicionar a *serverkey* com o seguinte comando *bash*:

Listing A.2: Código do subscritor Listener

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-
    key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Se houver problemas na conexão com o servidor 'hkp://ha.pool.sks-keyservers.net:80', este pode ser substituido por 'hkp://pgp.mit.edu:80' ou 'hkp://keyserver.ubuntu.com:80'.

Por ultimo, é recomendado atualizar o sistema para prevenir erros de dependências que possam estar desatualizadas e outras falhas que possam surgir:

Listing A.3: Código do subscritor Listener

```
sudo apt-get update -y
```

A.1.2 Instalação do Robot Operating System

Após os passos preparativos, pode iniciar a instalação de uma das versões da ferramenta. Existem vários pacotes em cada versão que podem ser encontrados com a seguinte linha, apenas alterando '<VERSION>' para a versão que se pretende utilizar:

Listing A.4: Código do subscritor Listener

```
apt-cache search ros-<VERSION>
```

Neste caso, foi instalado o pacote completo da versão Kinetic:

Listing A.5: Código do subscritor Listener

```
sudo apt-get install ros-kinetic-desktop-full
```

A.1.3 Configuração da Workspace

Depois de acabada a instalação, é necessário iniciar o 'rosdep' e atualizá-lo, que facilita a instalação de dependências essenciais e é indispensavel a componetes *core* do ROS:

Listing A.6: Código do subscritor Listener

```
sudo rosdep init
rosdep update
```

Para que as variáveis de ambiente ROS sejam automaticamente adicionadas a qualquer sessão *bash* iniciada, é conveniente utilizar os seguites comandos:

Listing A.7: Código do subscritor Listener

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

A instalação do ROS Kinetic atual permite correr pacotes básicos mas para criar e gerir ROS *workspaces* personalizadas requer correr a seguinte linha:

Listing A.8: Código do subscritor Listener

```
sudo apt install python-rosinstall python-rosinstall-generator python-
    wstool build-essential
```

Após completar a instalação do sistema, deve-se criar um *workspace*.

Listing A.9: Código do subscritor Listener

```
mkdir -p catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

O diretório 'src' é o local onde o utilizador deve inserir os seus projetos. A linha 'catkin_make' cria os diretórios 'build', 'devel'. No segundo diretório encontra-se o ficheiro 'setup.sh' onde se encontram os vários comandos ROS que podem ser invocados. A seguinte linha torna este ficheiro como a fonte destes comandos:

Listing A.10: Código do subscritor Listener

```
source devel/setup.bash
```

A.2 Suricata

A.2.1 Instalação de Dependências

Antes de iniciar a instalação da ferramenta deve-se atualizar o software do sistema, correndo as seguintes linhas no terminal:

Listing A.11: Código do subscritor Listener

```
sudo apt-get update -y
sudo apt-get upgrade -y
```

O Suricata requer também a instalação de dependências prévia à própria instalação:

Listing A.12: Código do subscritor Listener

```
apt-get install libpcre3-dbg libpcre3-dev autoconf automake libtool
    libpcap-dev libnet1-dev libyaml-dev libjansson4 libcap-ng-dev libmagic-
    dev libjansson-dev zlib1g-dev
```

Esta ferramenta funciona como um sistema de deteção de intrusões mas pode funcionar também como sistema de prevenção de intrusões, instalando todas as dependências com o comando seguinte:

Listing A.13: Código do subscritor Listener

```
apt-get install libnetfilter-queue-dev libnetfilter-queue1 libnfnetlink-
    dev
```

Porém, apenas será preciso a ferramenta em modo sistema de deteção de intrusões pelo que apenas será necessário o primeiro comando.

Após instadas as dependências, a ferramenta pode ser instalada de duas formas diferentes:

- Instalação pela Fonte
- Instalação por um Ubuntu PPA

A.2.2 Instalação pela Fonte

Neste tipo de instalação, o utilizador tem de descarregar uma versão do Suricata do repositório oficial *online*, por exemplo:

Listing A.14: Código do subscritor Listener

```
wget https://www.openinfosecfoundation.org/download/suricata-3.2.tar.gz
```

Depois de descarregado, terá de ser extraído, podendo faze-lo com o comando seguinte:

Listing A.15: Código do subscritor Listener

```
tar -xvzf suricata-3.2.tar.gz
```

De seguida acede-se ao diretório criado pela extração com o primeiro comando. A segunda linha pode ser utilizada caso se pretenda a funcionalidade IPS:

Listing A.16: Código do subscritor Listener

```
cd suricata-3.2
./configure --enable-nfqueue --prefix=/usr --sysconfdir=/etc --
    localstatedir=/var
```

Finalmente, a instalação é feita com os dois primeiros comandos. O terceiro comando instala a configuração padrão do suricata:

Listing A.17: Código do subscritor Listener

```
make
make install
make install-conf
```

A.2.3 Instalação por um Ubuntu PPA

Outra forma de instalar o Suricata de forma mais simplificada é através de *Personal Package Archives* ou PPA, que são repositórios online de software exclusivo destinado a software e atualizações não padrão.

Primeiro o PPA do Suricata tem de ser adicionado ao repositório Ubuntu:

Listing A.18: Código do subscritor Listener

```
add-apt-repository ppa:oisf/suricata-stable
```

Depois de adicionado, o repositório deve ser atualizado com o primeiro comando e está pronto a instalar a ultima versão do Suricata fornecida pelo repositório:

Listing A.19: Código do subscritor Listener

```
apt-get update -y
apt-get install suricata suricata-dbg -y
```

A.2.4 Configuração do Suricata

Esta ferramenta foi utilizada como um filtro de informação das comunicações relevantes ao trabalho. De forma a facilitar o trabalho esta foi usada em modo de leitura de captura de pacotes, pelo que a sua configuração é básica.

A configuração do Suricata pode ser encontrada em:

Listing A.20: Código do subscritor Listener

```
/etc/suricata/suricata.yaml
```

Neste ficheiro apenas serão alterados alguns aspetos como a interface em que o Suricata irá capturar, que será atribuída à rede *localhost*, que por definição utiliza o endereço 127.0.0.1 e definida como *lo*, como podemos observar na figura A.1.

Outro aspeto que foi alterado foi o modo de *output* da ferramenta. Sempre que a informação é capturada e filtrada pelas regras é guardada num ficheiro com o nome ‘fast.log’, localizado em:

```
# Cross platform libpcap capture support
pcap:
  - interface: lo
    # On Linux, pcap will try to use mmaped capture and will use buffer-size
    # as total of memory used by the ring. So set this to something bigger
    # than 1% of your bandwidth.
```

Figura A.1: Interface da funcionalidade pcap

Listing A.21: Código do subscritor Listener

```
/var/log/suricata/fast.log
```

Nesta secção apenas foi alterado o modo como imprime para o ficheiro de maneira a apagar o que estiver neste para facilitar a leitura e analise. Esta funcionalidade pode ser ativa apenas atribuindo 'no' à variável 'append', como podemos verificar na figura A.2.

Por último, foi criado um ficheiro chamado 'rules.rules', onde foram definidas as regras utilizadas. Para que o Suricata tenha conhecimento da existência deste ficheiro e das suas regras, é necessário introduzir o seu nome, verificado na figura A.3, e coloca-lo no diretório de regras do suricata com a seguinte localização padrão:

Listing A.22: Código do subscritor Listener

```
/etc/suricata/rules
```

A.3 Wireshark

A ferramenta Suricata foi utilizada como ferramenta de captura de pacotes e analise da rede e dos protocolos utilizados pelos elementos nelas presentes. Para efetuar a sua instalação, primeiro é necessário adicionar um repositório de onde esta ferramenta pode ser descarregada. Esta pode ser descarregada de um PPA oficial seguido de uma atualização de repositórios, introduzindo os seguintes comandos no terminal:

```
# Configure the type of alert (and other) logging you would like.
outputs:
  # a line based alerts log similar to Snort's fast.log
  - fast:
      enabled: yes
      filename: fast.log
      append: no
      #filetype: regular # 'regular', 'unix_stream' or 'unix_dgram'
```

Figura A.2: Modo de escrita da ferramenta

```
##
## Step 2: select the rules to enable or disable
##

default-rule-path: /etc/suricata/rules
rule-files:
 - rules.rules
```

Figura A.3: Regras ativas

Listing A.23: Código do subscritor Listener

```
sudo add-apt-repository ppa:wireshark-dev/stable
sudo apt-get update
```

Após terminada a atualização de repositórios, a ferramenta está pronta a descarregar e instalar utilizando o seguinte comandos no terminal:

Listing A.24: Código do subscritor Listener

```
sudo apt-get install wireshark
```

Para iniciar o programa basta executar a seguinte linha no terminal:

Listing A.25: Código do subscritor Listener

```
sudo wireshark
```

Em caso de permissão negada na execução da ferramenta, pode ser executada a seguinte linha:

Listing A.26: Código do subscritor Listener

```
sudo dpkg-reconfigure wireshark-common
```

Este comando cria um grupo de utilizadores Wireshark que podem ser adicionados com o proximo comando, onde $USER é o nome do utilizador pretendido:

Listing A.27: Código do subscritor Listener

```
sudo adduser $USER wireshark
```

Referências

[1] Node-RED. URL: https://nodered.org/.

[2] Michael Blackstock e Rodger Lea. Fred: a hosted data flow platform for the iot built using node-red. *Proceedings of MoTA*, 2016.

[3] ROS.org | powering the world's robots. URL: http://www.ros.org/.

[4] Nicholas DeMarinis, Stefanie Tellex, Vasileios Kemerlis, George Konidaris, e Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. *arXiv preprint arXiv:1808.03322*, 2018.

[5] CoAP — constrained application protocol | overview. URL: http://coap.technology/.

[6] MQTT. URL: http://mqtt.org/.

[7] Juan Enrique Rubio, Cristina Alcaraz, Rodrigo Roman, e Javier Lopez. Analysis of intrusion detection systems in industrial ecosystems. Em *14th International Conference on Security and Cryptography (SECRYPT 2017)*, 2017.

[8] Syaiful Andy, Budi Rahardjo, e Bagus Hanindhito. Attack scenarios and security analysis of mqtt communication protocol in iot system. Em *Electrical Engineering, Computer Science and Informatics (EECSI), 2017 4th International Conference on*, páginas 1–6. IEEE, 2017.

[9] P. Pongle e G. Chavan. A survey: Attacks on RPL and 6lowpan in IoT. doi:10.1109/PERVASIVE.2015.7087034.

[10] M. Nawir, A. Amir, N. Yaakob, e O. B. Lynn. Internet of things (IoT): Taxonomy of security attacks. doi:10.1109/ICED.2016.7804660.

[11] Bruno Bogaz Zarpelao, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, e Sean Carlisto de Alvarenga. A survey of intrusion detection in internet of things. *Journal of Network and Computer Applications*, 84:25–37, 2017.

[12] Suricata. URL: https://suricata-ids.org/.

[13] Tom DeCanio. Contribute to decanio/suricata-IoT development by creating an account on GitHub. original-date: 2016-07-07T02:13:42Z. URL: https://github.com/decanio/suricata-IoT.

[14] Snort - network intrusion detection & prevention system. URL: https://www.snort.org/.