

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA



RELATÓRIO FINAL DO TRABALHO DE CONCLUSÃO DE CURSO

Desenvolvimento de estratégias de segurança e monitoramento para sistemas embarcados provisionado em ambiente virtual

Autor: Victor Cintra Santos

Orientador: Prof. Dr. Euripedes Guilherme de Oliveira Nobrega

9 de dezembro de 2022

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

RELATÓRIO FINAL
TRABALHO DE CONCLUSÃO DE CURSO

DESENVOLVIMENTO DE ESTRATÉGIAS DE SEGURANÇA E MONITORAMENTO
PARA SISTEMAS EMBARCADOS PROVISIONADO EM AMBIENTE VIRTUAL

Autor: Victor Cintra Santos

Orientador: Prof. Dr. Euripedes Guilherme de Oliveira Nobrega

Curso: Engenharia de Controle e Automação

Trabalho de conclusão de curso apresentado à Comissão de Graduação da Faculdade de Engenharia Mecânica, como requisito para a obtenção do título de Engenheiro de Controle e Automação.

Campinas, 2022
São Paulo - Brasil

Dedico este trabalho aos meus pais e ao meu irmão, os alicerces e os grandes responsáveis por viabilizarem minha formação e as conquistas na minha vida; aos amigos que sempre estiveram ao meu lado, eterna gratidão.

Todos esses que aí estão atravancando meu
caminho, eles passarão...
Eu passarinho!

Eu Passarinho
Mario Quintana

Conteúdo

1	Introdução	2
2	Objetivos	2
3	Fundamentação teórica	3
3.1	ROS - Robot Operating System	3
3.2	Docker	4
3.3	Grafana	5
3.4	OWASP ZAP	5
4	Metodologia	6
4.1	Arquitetura Proposta	6
4.2	Virtualização do sistema embarcado	7
4.2.1	Arquitetura ROS	10
4.3	Instrumentação do sistema de monitoramento	12
4.3.1	Monitoramento do sistema operacional: node-exporter	12
4.3.2	Monitoramento da simulação ROS: Pushgateway e Node.js	14
4.3.3	Agregando as métricas e monitorando o sistema: Grafana e Prometheus	17
4.3.4	Construção das métricas e interpretação dos gráficos	20
4.3.5	Arquitetura e versionamento do projeto	21
4.4	Externalização do sistema	22
4.5	Segurança	24
4.6	Resultados finais e conclusões	25
5	Referências bibliográficas	27
	Apêndices	29
A	Arquivo index.js para a aplicação em Node.js	29
B	Arquivo config.js com as configurações necessárias para a aplicação Node.js	32
C	Arquivo constants.js com os marcadores para a construção das métricas na aplicação Node.js	33
D	Script para execução da aplicação	34
E	Script para o desligamento da aplicação	35

Lista de Figuras

1	Turtlesim - Aplicação de sistemas embarcados baseados em ROS	3
2	Turtlebot - Aplicação prática de sistemas embarcados baseados em ROS	4
3	Módulo Principal - Simulação Embarcada	6
4	Módulo Secundário - Validação de Segurança	7
5	Simulação ROS TurtleSim	9
6	Simulação ROS TurtleSim - trajetória alterada pelo nó <i>/teleop_turtle</i>	9
7	Saída da simulação ROS	10
8	Arquitetura Virtualizada no ROS	11
9	Arquitetura Virtualizada no ROS - Destaque aos Nós, Tópicos e Mensagens	11
10	Saída execução node_exporter	13
11	Porta local exposta com as métricas do node_exporter	14
12	Registro de logs do tópico /turtle1/pose indicando a posição e velocidade atual do robô simulado	15
13	Instância do Pushgateway com as métricas expostas localmente	16
14	Instância do Prometheus executando uma querie de pesquisa no banco de dados de métricas instrumentado	18
15	Instância Grafana em localhost	19
16	Grafana - Configuração do Prometheus como datasource no Grafana	19
17	Grafana - Dashboards das métricas do sistema	20
18	Grafana - Dashboard de posição do robô	21
19	Arquitetura Final - Integrações entre o ROS e as aplicações de monitoramento	22
20	NGROK - Tela inicial	23
21	NGROK - Logs quando a aplicação é acessada	23
22	NGROK - Grafana disponibilizado pela aplicação	24
23	Resultados Finais - Arquitetura Final - Integrações entre o ROS e as aplicações de monitoramento externalizadas em uma URL pública pelo NGROK	25
24	Resultados Finais - Grafana - Logs da simulação ROS	26
25	Resultados Finais - Grafana - Logs da performance do RaspberryPi	26

Resumo

Este relatório final de trabalho de conclusão de curso resume as atividades empreendidas no primeiro e segundo semestres de 2022, correspondendo ao período de atividades do aluno no seu Trabalho de Graduação das matérias "ES951 - Trabalho de Graduação I" e "ES952 - Trabalho de Graduação II". O projeto desenvolvido visa abordar o problema de segurança e monitoramento de sistemas embarcados em ambiente virtual, utilizando tecnologias modernas para a instrumentação do projeto aliadas ao sistema de simulação ROS2, aplicado à máquinas virtuais.

Keywords: ROS2, virtualizing embedded systems, Docker, monitoring virtualized systems, Grafana, Prometheus, node_exporter, systems monitoring instrumentation, ngrok.

1 Introdução

Sistemas embarcados da atualidade estão sujeitos a serem invadidos e utilizados de forma maliciosa, levando assim a uma preocupação que resultou em uma nova área de pesquisa que é constituída por técnicas de monitoramento e interpretação de dados para sua prevenção. Está sendo proposta aqui uma metodologia aplicada aos robôs industriais baseada na análise dos logs do sistema operacional associados ao comportamento apresentado pelos robôs respectivos.

2 Objetivos

O objetivo deste trabalho de graduação é o desenvolvimento de um sistema de monitoramento e segurança da operação de um sistema robótico, acompanhando sua integridade e funcionamento através de ferramentas de modelagem, análise e visualização do comportamento de sistemas embarcados. Aliado a isso, o desenvolvimento de um sistema de segurança à possíveis tentativas de invasão ao sistema.

3 Fundamentação teórica

3.1 ROS - Robot Operating System

O Sistema Operacional para Robôs (ROS) é uma plataforma de simulação e desenvolvimento de sistemas embarcados, contendo os módulos necessários para a simulação de um robô, com os dispositivos de entrada e saída necessários para a interação com o mesmo.

Tal sistema segue o protocolo Produtor/Consumidor ("*Publisher/Subscriber*") [18]. Nesse protocolo um cliente envia mensagens diretamente a um *broker* e outro cliente pode acessar essas informações se inscrevendo nesse intermediário. No ROS segue a mesma lógica: um módulo (chamado de "Nó") é responsável por publicar mensagens, informações, em um fluxo (chamado de "Tópico"). As mensagens no tópico podem ser acessadas por qualquer nó que tenha interesse por meio de uma subscrição. Portanto, se um tipo de mensagem for de interesse de algum nó, esse módulo em específico pode se inscrever nesse fluxo de mensagens. Esse protocolo difere do Cliente/Servidor, já que esse se vale da requisição da informação antes de ter a resposta com o conteúdo solicitado.

Nesse trabalho tomamos como base a simulação do TurtleSim: um robô tartaruga simplificado que consegue se movimentar num ambiente 2D recebendo comandos de posição e velocidade linear e angular, enviando mensagens do seu status de operação, posicionamento na tela e controles de colisão. Abaixo, exemplos da simulação do TurtleSim na Figura 1 que possui aplicações práticas como o TurtleBot, Figura 2, também executado sob plataforma ROS:

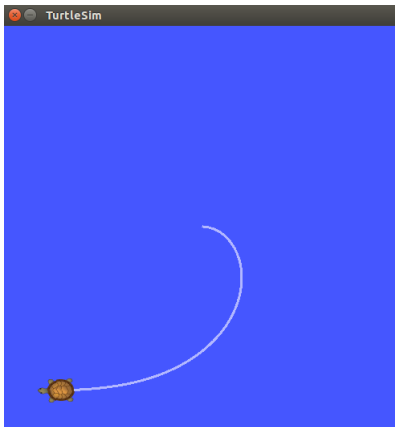


Figura 1: Turtlesim - Aplicação de sistemas embarcados baseados em ROS



Figura 2: Turtlebot - Aplicação prática de sistemas embarcados baseados em ROS

No desenvolvimento desse trabalho foram utilizados alguns conceitos básicos do sistema ROS, sendo eles:

- *Nó (Node)*: é um módulo base do sistema ROS, responsável por simular um sistema embarcado ou então de simular sensores ou comandos; basicamente é um processo que executa métodos enviando mensagens de saída sobre o seu estado;
- *Tópico (Topic)*: é o meio de comunicação entre os nós; é uma via de troca de mensagens entre nós distintos, podendo ser requisitado (assinado) pelos nós que tem interesse nessa via de comunicação;
- *Publicações (Publications)*: é a saída de um nó para um tópico; um nó pode publicar mensagens em um tópico, como por exemplo o seu estado de execução, sendo essas mensagens algum tipo de estrutura de dados;
- *Inscrições (Subscriptions)*: são os tópicos em que os nós se relacionam sem definir alguma tipagem de mensagem;
- *Serviços (Services)*: são os serviços expostos de um nó que podemos utilizar para interagir com o sistema embarcado, como por exemplo um serviço de controle de velocidade linear e angular, que pode ser utilizado para controlar o robô simulado.

Para o trabalho, o TurtleSim será o nó utilizado que está inscrito no tópico `/turtle1/cmd_vel` para receber mensagens de controle de posicionamento e velocidade linear e angular.

3.2 Docker

Como indicado na documentação oficial do sistema [5], Docker é um gerenciador de pacotes de produtos que se utiliza da virtualização do sistema operacional para provisionar todo o ferramental e dependências necessárias para uma aplicação. A virtualização criada pelo Docker é chamada de container. Tal provisionamento da plataforma como serviço é chamado de PaaS [23].

3.3 Grafana

Grafana [6] é um software *open source* de monitoramento de dados, podendo ser utilizado para gestão de logs de operações bem como visualização de gráficos e suporte a pesquisas por queries numa base de logs. Há diversos modelos disponíveis podendo ser customizado livremente de acordo com a aplicação requerida. Nesse trabalho será utilizado para centralizar o monitoramento do sistema virtualizado, compilando indicadores do hardware utilizado (RaspberryPi) e status do funcionamento do robô (posicionamento na tela e indicadores de colisão).

3.4 OWASP ZAP

O OWASP ZAP [7] também é um software *open source* utilizado para varrer endereços na web, conexões e comunicações entre sistemas buscando vulnerabilidades de serviços e falhas de segurança. Tal ferramenta gera status sobre o endereço e a aplicação submetida a testes, pontuando as falhas encontradas podendo ser utilizada para aprimorar a segurança da comunicação com o robô simulado.

4 Metodologia

Para o objetivo desse trabalho, a ideia base é de criar um ambiente virtualizado com a simulação de um sistema embarcado baseado no ROS e expor esse sistema a possíveis ataques, monitorando toda a aplicação utilizando o Grafana e criando estratégias de segurança e as validando por meio do OWASP ZAP. Sendo assim, analisamos na sequência a arquitetura geral do sistema.

4.1 Arquitetura Proposta

Para o desenvolvimento do trabalho será desenvolvida a seguinte arquitetura do projeto:

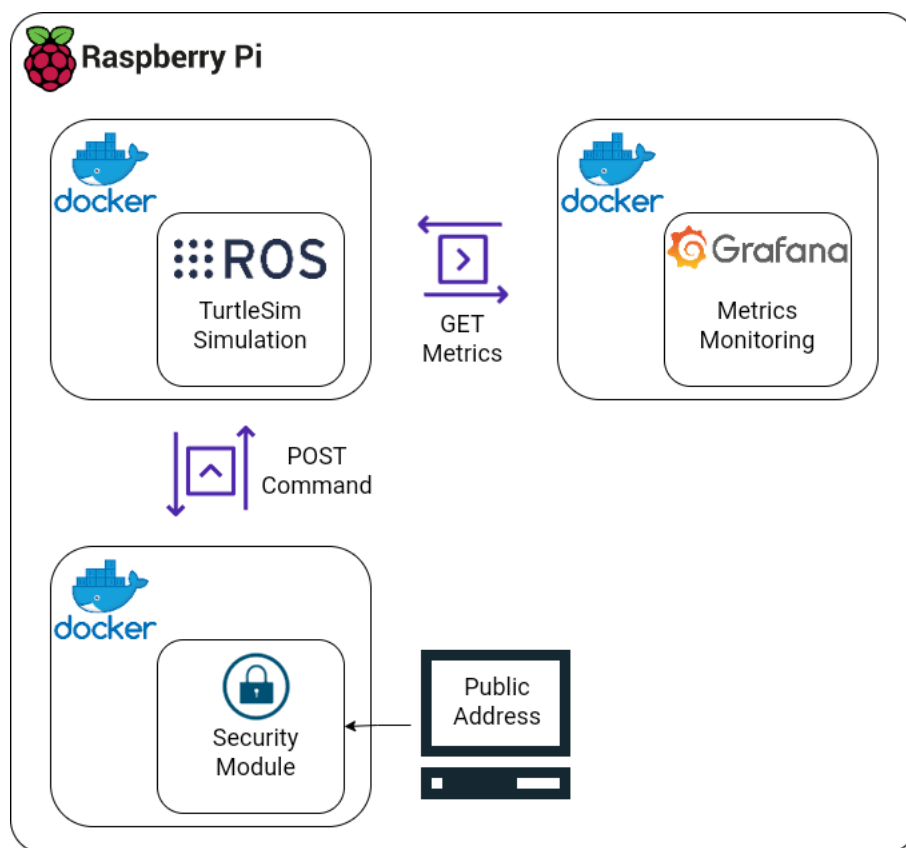


Figura 3: Módulo Principal - Simulação Embarcada

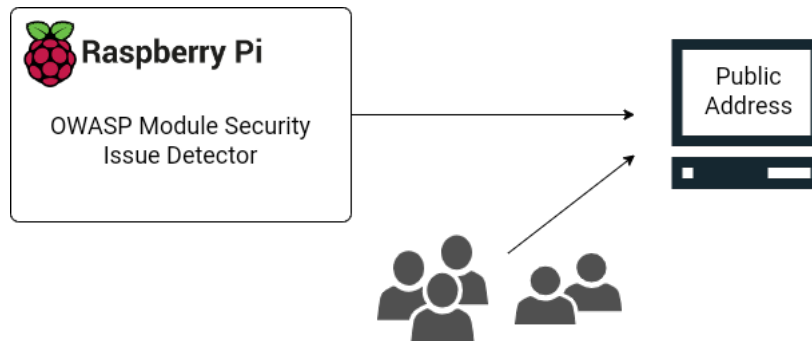


Figura 4: Módulo Secundário - Validação de Segurança

Os módulo principal, detalhado na Figura 3, contém as seguintes especificações:

- *Framework* ROS [11] com a simulação do Sistema Embarcado TurtleSim, simulando um robô responsivo a comandos de locomoção;
- Monitoramento de métricas através de API para averiguação do funcionamento do sistema embarcado simulado;
- Compilação e visualização das métricas pelo Grafana [6], como por exemplo a posição do robô e indicadores de colisão, bem como o status do hardware da simulação e os contêineres envolvidos;
- Módulo de Segurança acoplado à simulação do sistema embarcado para garantir a segurança de seu funcionamento;
- Endereço público de acesso à simulação do robô, permitindo interação com o mesmo para validação do Módulo de Segurança;

Todos os módulos descritos serão virtualizados e provisionados separadamente em instâncias Docker [5], para haver desacoplamento dos sistemas e comunicação entre os mesmo. Todo o sistema será provisionado na placa de desenvolvimento RaspberryPi [10].

Além do módulo principal, haverá um módulo secundário, indicado pela Figura 4, de tentativa de explorar vulnerabilidades do módulo de segurança e de interferir no funcionamento do robô, fornecendo um relatório do sistema OWASP ZAP[7] das vulnerabilidades encontradas. Tal módulo também será provisionado em RaspberryPi ou mesmo um computador secundário.

O sistema tem como saída as métricas retiradas pelo Grafana e o relatório do módulo de segurança. Podendo aferir o comportamento do robô com tentativas de ataque ao sistema, avaliando como o Módulo de Segurança protegeu o sistema embarcado.

4.2 Virtualização do sistema embarcado

Para a virtualização da simulação do nó TurtleSim do ROS, como indicado no repositório [12] sobre "*different ways to deal with ROS 2 node interconnectivity*", primeiramente é necessário instalar as dependências necessárias para o correto funcionamento da aplicação.

Para a virtualização, foi necessário criar um arquivo Docker com a receita para a criação do container com as dependências necessárias. Com tal arquivo para o provisionamento do ambiente, os passos a seguir foram executados no terminal:

1. Atualizar todas as dependências da máquina e instalar o sistema Docker:

```
sudo apt-get update; sudo apt-get upgrade;

sudo -E apt-get -y install apt-transport-https \
ca-certificates software-properties-common && \

curl -sL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - && \

arch=$(dpkg --print-architecture) && \

sudo -E add-apt-repository "deb [arch=${arch}] \
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" && \

sudo -E apt-get update && \
sudo -E apt-get -y install docker-ce docker-compose

sudo systemctl daemon-reload
sudo systemctl restart docker
```

2. Com as dependências atualizadas e instaladas, executamos o arquivo Docker para a criação do container com o sistema ROS:

```
sudo chmod +x eg1/ros_entrypoint.sh

sudo docker build -t turtle_demo -f eg1/Dockerfile .

xhost local:root
```

3. A partir desse momento, executamos os módulos do ROS para a nossa simulação do sistema embarcado com o comando abaixo:

```
sudo docker run --rm -it --env DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix:rw \
turtle_demo ros2 launch my_turtle_bringup turtlesim_demo.launch.py
```

A simulação abre uma janela em que mostra a execução do robô TurtleSim com a sua trajetória, como na Figura 5. Podemos também interagir com o robô através do nó */teleop.turtle* executado numa diferente janela do terminal, alterando a trajetória como na Figura 6 abaixo:



Figura 5: Simulação ROS TurtleSim

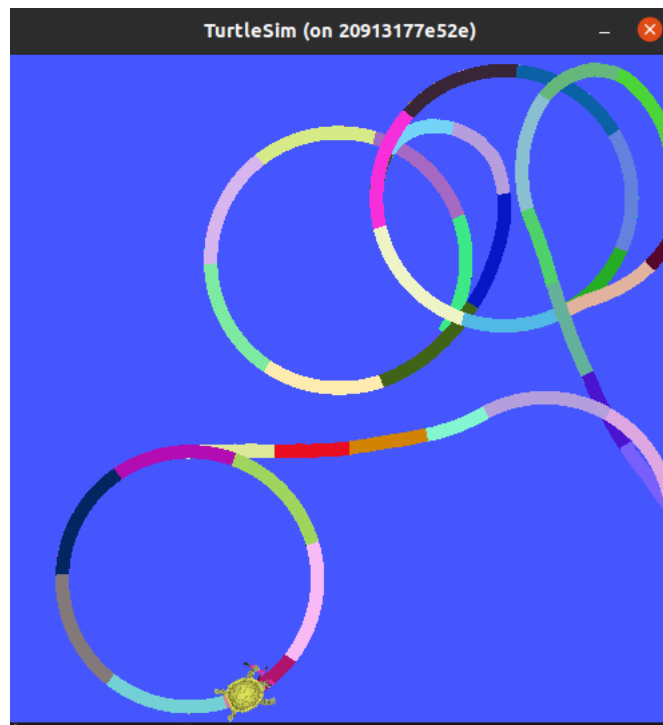


Figura 6: Simulação ROS TurtleSim - trajetória alterada pelo nó */teleop_turtle*

Com a execução da simulação acima, o robô fornece dados diretamente relacionados a sua execução e ao seu posicionamento na tela. Tais informações podem ser recolhidas inspecionando o tópico */turtle1/cmd_vel*, que fornece as informações de velocidades linear e angular do robô, ou então acompanhando a saída da execução do comando de inicialização da simulação, que mantém o processo na janela do terminal gerando mensagens de histórico da execução da simulação. Na Figura 7 abaixo, pode-se ver até que na saída indica quando o robô atingiu os limites virtuais da janela da simulação.

```
[color_controller-3] [INFO] [1657064072.563765796] [color_controller]: [247, 46, 218] SetPen srv response: turtlesim.srv.SetPen_Response()
[color_controller-3] [INFO] [1657064073.574140261] [color_controller]: [239, 244, 199] SetPen srv response: turtlesim.srv.SetPen_Response()
[color_controller-3] [INFO] [1657064074.643366706] [color_controller]: [82, 184, 229] SetPen srv response: turtlesim.srv.SetPen_Response()
[color_controller-3] [INFO] [1657064075.556337986] [color_controller]: [225, 178, 157] SetPen srv response: turtlesim.srv.SetPen_Response()
[color_controller-3] [INFO] [1657064076.563030385] [color_controller]: [86, 7, 44] SetPen srv response: turtlesim.srv.SetPen_Response()
[turtlesim_node-1] [WARN] [1657064076.815894508] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.098142, y=2.992025])
[turtlesim_node-1] [WARN] [1657064076.832634259] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.103717, y=2.963668])
[turtlesim_node-1] [WARN] [1657064076.848263369] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.103288, y=2.935091])
[turtlesim_node-1] [WARN] [1657064076.864085518] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.102855, y=2.906299])
[turtlesim_node-1] [WARN] [1657064076.880164974] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.102420, y=2.877301])
[turtlesim_node-1] [WARN] [1657064076.896074681] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.101981, y=2.848101])
[turtlesim_node-1] [WARN] [1657064076.912234598] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.101539, y=2.818708])
[turtlesim_node-1] [WARN] [1657064076.928400106] [turtlesim]: Oh no! I hit the wall! (Clamping from [x=11.101094, y=2.789127])
```

Figura 7: Saída da simulação ROS

4.2.1 Arquitetura ROS

Na virtualização do sistema ROS executamos os seguintes módulos para a nossa simulação do sistema embarcado:

1. Nó */turtlesim*: robô tartaruga com os tópicos e subscrições necessárias para a operação, inscrito no tópico */turtle1/cmd_vel* para receber comandos de velocidade linear e angular;
2. Nó */move_controler*: responsável por executar comandos de velocidade angular e linear constantes para o robô, enviando comandos para o tópico */turtle1/cmd_vel*, mantendo o robô num percurso constante desenhando um círculo na tela;
3. Nó */color_controler*: responsável apenas por aplicar cores no trajeto do robô para melhor visualização;
4. Nó */teleop_turtle*: responsável por criar uma interação com as setas do teclado para controlar a trajetória do robô, publicando mensagens diretamente no tópico */turtle1/cmd_vel*.

Tais módulos podem ser verificados graficamente numa arquitetura detalhada indicando a virtualização do sistema ROS, através do módulo *rqt_graph*. Executando o comando abaixo para construir a arquitetura da simulação, temos como saída o diagrama da Figura 8 que segue:

```
sudo docker run --rm -it --env DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix:rw \
turtle_demo ros2 run rqt_graph rqt_graph
```

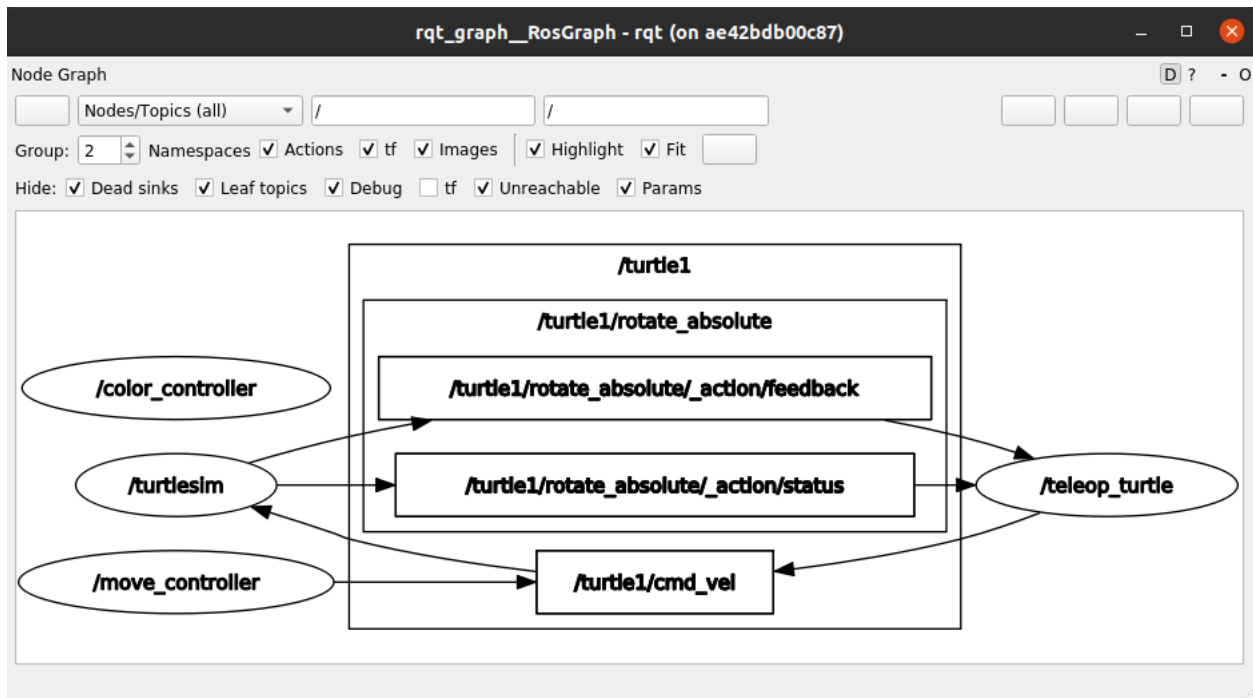


Figura 8: Arquitetura Virtualizada no ROS



Figura 9: Arquitetura Virtualizada no ROS - Destaque aos Nós, Tópicos e Mensagens

Na Figura 9 acima, podemos ver destacado somente a estrutura de nós e tópicos do sistema ROS

virtualizado. Nela, tanto o nó */move_controller* quanto o nó */telop_turtle* publicam mensagens no tópico */turtle1/cmd_vel* em que o nó principal */turtlesim* se inscreve. Esses dois nós são os responsáveis por movimentar o robô tartaruga (sendo o primeiro responsável pelo movimento circular constante e o segundo por passar comandos das setas do teclado para o robô). O nó */color_controller* não publica e nem se inscreve em nenhum tópico pois ele só é usado para colorir a trajetória na tela. O nó */turtlesim* publica mensagens referente a sua rotação na tela através dos tópicos */rotate_absolute*.

Na Figura 8, temos uma visão mais macro do processo em que o ponto de destaque é o escopo */turtle1/rotate_absolute* referente aos comandos constantes que o nó */move_controller* escreve no tópico */turtle1/cmd_vel*, que faz com que o robô mantenha uma trajetória circular. O escopo geral da simulação é nomeado */turtle1*.

4.3 Instrumentação do sistema de monitoramento

Para o sistema de monitoramento, dividiremos as informações a serem monitoradas em dois grupos distintos:

- Sistema operacional: dados como uso de processamento de CPU, memória, tempo de execução do sistema, instâncias dockers virtualizadas, dentre outros dados úteis da máquina e sistema operacional que suporta a simulação;
- Simulação do sistema embarcado em ROS: dados relacionados ao funcionamento do robô, trajetória, problemas na simulação;

Para a centralização dos monitoramento da simulação, utilizaremos o Grafana como um agregador de todas as fontes de informação do sistema. Para ingestão de dados ao Grafana, utilizaremos o Prometheus integrado ao *node-exporter* e o *Pushgateway*.

4.3.1 Monitoramento do sistema operacional: node-exporter

O *node-exporter* [26] é uma aplicação para expor métricas de hardware e do kernel do sistema operacional, como memória disponível do sistema, uso dos núcleos do processador, tempo de disponibilidade do sistema, entre outras. A vantagem dessa aplicação é de se integrar facilmente a uma das bases de dado do Grafana, o Prometheus.

Para instalarmos o *node-exporter*, seguimos a documentação oficial da integração com o Prometheus [19], com o pacote da aplicação disponibilizado no site:

```
wget https://\
github.com/prometheus/node_exporter/releases/download/v*/node_exporter-*. *-amd64.tar.gz

tar xvfz node_exporter-*. *-amd64.tar.gz
```

Entrando na pasta do *node-exporter*, executamos o binário:

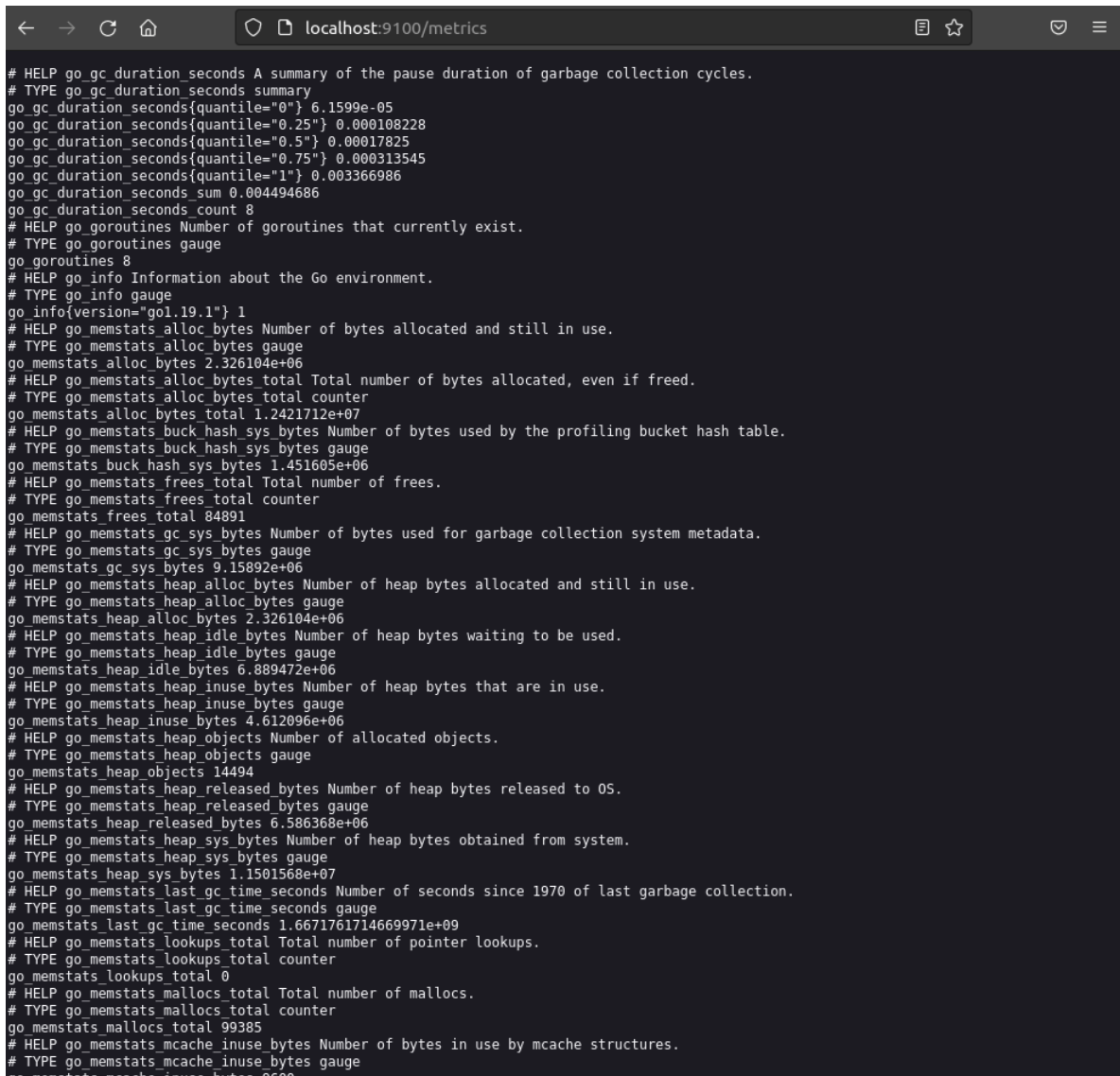
```
./node_exporter
```

Com a aplicação sendo executada, no terminal temos alguns logs das métricas sendo coletadas, como mostrado na Figura 10 abaixo:

```
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=rapl
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=schedstat
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=selinux
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=sockstat
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=softnet
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=stat
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=tapestats
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=textfile
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=thermal_zone
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=time
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=timex
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=udp_queues
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=uname
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=vmstat
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=xtfs
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:115 level=info collector=zfs
ts=2022-10-30T23:42:21.769Z caller=node_exporter.go:199 level=info msg="Listening on" address=:9100
ts=2022-10-30T23:42:21.769Z caller=tls_config.go:195 level=info msg="TLS is disabled." http2=false
```

Figura 10: Saída execução node_exporter

E ao acessarmos o endereço local na porta 9100 (<http://localhost:9100/metrics>), vemos as métricas do sistema expostas pelo *node_exporter*:



```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 6.1599e-05
go_gc_duration_seconds{quantile="0.25"} 0.000108228
go_gc_duration_seconds{quantile="0.5"} 0.00017825
go_gc_duration_seconds{quantile="0.75"} 0.000313545
go_gc_duration_seconds{quantile="1"} 0.003366986
go_gc_duration_seconds_sum 0.004494686
go_gc_duration_seconds_count 8
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 8
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.19.1"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 2.326104e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.2421712e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.451605e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 84891
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 9.15892e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 2.326104e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 6.889472e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 4.612096e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 14494
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 6.586368e+06
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 1.1501568e+07
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage collection.
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.6671761714669971e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 99385
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 9600
```

Figura 11: Porta local exposta com as métricas do node_exporter

4.3.2 Monitoramento da simulação ROS: Pushgateway e Node.js

Para o monitoramento da operação do robô simulado no ROS na instância Docker, como o sistema não possui uma integração simplificada com o Grafana, ou com outra visualização de integrações de métricas, ou até mesmo alguma porta exposta para requisições *GET* para obter o estado do sistema, temos que instrumentar uma aplicação baseada nos comandos já existentes do ROS.

Partindo da análise do tópico de comunicação entre o nó */move_controller* com o nó do robô */turtlesim*, podemos inspecionar os comandos de movimentação do robô, que indicam a sua atual posição. Rodando o comando abaixo no terminal a partir da instância Docker do *turtlesim*, obtemos sua atual posição e redirecionamos esse registro para um arquivo de log **position.txt** que trataremos posteriormente:

```
sudo docker run --rm -it --env DISPLAY \
--volume /tmp/.X11-unix:/tmp/.X11-unix:rw turtle_demo \
ros2 topic echo /turtle1/pose >> position.txt
```

Nos fornecendo os registros da posição do robô abaixo:

- x: posição do robô no eixo x;
- y: posição do robô no eixo y;
- theta: ângulo da trajetória do robô referente ao plano da simulação;
- linear_velocity: velocidade linear de operação constante;
- angular_velocity: velocidade angular de operação constante

```
x: 3.5103251934051514
y: 7.014804363250732
theta: -1.2662125825881958
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.520381212234497
y: 6.9844255447387695
theta: -1.2511329650878906
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.5308940410614014
y: 6.954201698303223
theta: -1.2360533475875854
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.5418612957000732
y: 6.924139976501465
theta: -1.2209737300872803
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.553280830383301
y: 6.894246578216553
theta: -1.205894112586975
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.5651497840881348
y: 6.864529132843018
theta: -1.19081449508667
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
x: 3.577465295791626
y: 6.834994316101074
theta: -1.1757348775863647
linear_velocity: 2.0
angular_velocity: 0.942477822303772
---
```

Figura 12: Registro de logs do tópico /turtle1/pose indicando a posição e velocidade atual do robô simulado

Além de buscarmos informações sobre a posição do robô, queremos identificar a saúde da sua operação, podendo identificar falhas ou então colisões nas limitações da simulação. Para isso, usaremos a mesma estratégia: na execução do comando de início da simulação, redirecionaremos os registros para um arquivo centralizado de logs **output.txt**, que listará a saída da simulação como indicado na Figura 7.

Com esses dois arquivos de texto nos fornecendo em tempo real os registros da saúde da operação do sistema do robô simulado, bem como a sua posição e velocidade, criamos uma aplicação local em Node.js [21] que constantemente ingere os dados desses arquivos e utiliza o Prometheus Pushgateway [25] para disponibilizar os dados para o Grafana.

O Pushgateway é uma aplicação nativa do Prometheus que serve para expor endpoints locais para qualquer aplicação poder inserir, deletar ou consultar métricas. Para o seu provisionamento no sistema, foi utilizado a documentação oficial citada acima, que cria uma instância Docker e disponibiliza o Pushgateway na porta local 9091 (<http://localhost:9091/>), como na Figura 13 abaixo:

```
docker pull prom/pushgateway
docker run -d -p 9091:9091 prom/pushgateway
```

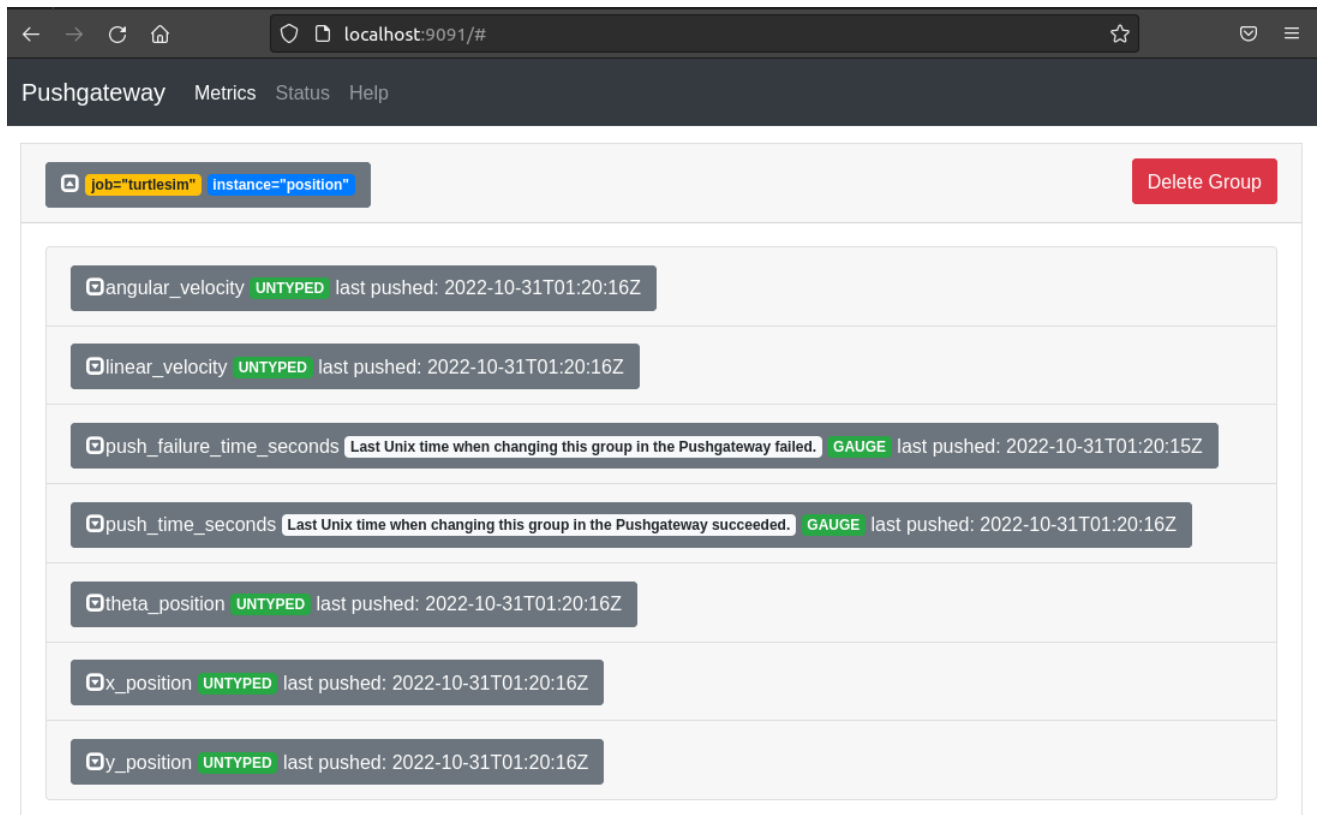


Figura 13: Instância do Pushgateway com as métricas expostas localmente

Assim, nosso sistema Node.js vasculha os arquivos **output.txt** e **position.txt** enviando esses dados via

requisição POST API, utilizando o pacote *npm* axios [3] para a requisição:

```
await axios({
  method: 'post',
  url: `${PUSHGATEWAY_INSTANCE}/job/${job}/instance/${instance}`,
  data,
});
```

Na requisição acima, comparada com a Figura 13, pode-se perceber que nas métricas inseridas no Pushgateway podemos definir um nome para o serviço/aplicação (*job*) e dividir em instâncias (*instances*), incluindo também marcadores nas métricas para serem mais facilmente identificadas.

Para a aplicação Node.js conseguir observar os arquivos de saída da simulação, foi usado o *watchFile* [13], uma funcionalidade do Node.js de criar um *listener* para arquivos e notificar sempre que o arquivo tiver alguma alteração, trazendo consigo os dados referentes a essa atualização. Essa funcionalidade foi utilizada de maneira simplificada através do módulo *npm Tail* [17], bastando apenas passarmos o caminho do arquivo que se deseja criar esse *observer* e como resposta obtemos as linhas que foram inseridas nesse arquivo.

4.3.3 Agregando as métricas e monitorando o sistema: Grafana e Prometheus

Com as duas fontes de métricas configuradas (*node_exporter* e Pushgateway), podemos instanciar o Prometheus para integrar essas métricas como um centralizador e as disponibilizar ao Grafana. Seguindo a documentação oficial para essa integração [19]:

```
wget \
https://github.com/prometheus/prometheus/releases/download/v*/prometheus-*.*-amd64.tar.gz

tar xvf prometheus-*.*-amd64.tar.gz
```

E executando o utilitário:

```
./prometheus --config.file=./prometheus.yml
```

Ao acessarmos localmente o endereço 9090 (<http://localhost:9090>), podemos ver a aplicação do Prometheus e já criar consultas (*queries*) para pesquisar as métricas no nosso banco de dados instrumentado:

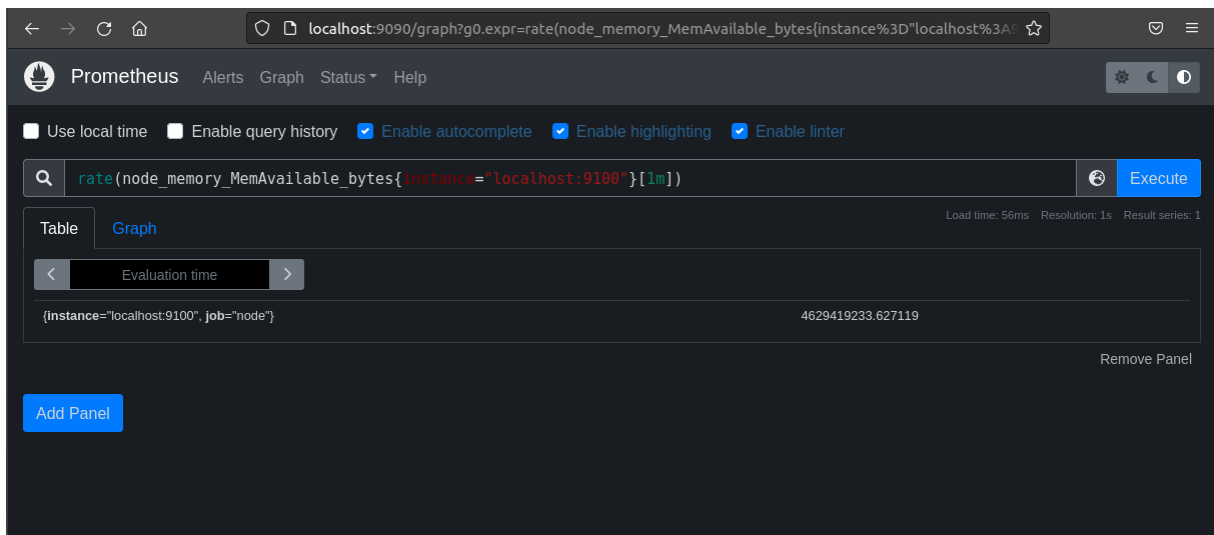


Figura 14: Instância do Prometheus executando uma querie de pesquisa no banco de dados de métricas instrumentado

Com nossas métricas e banco de dados funcionais, podemos criar agora o visualizador integrado do sistema, utilizando o Grafana. Com a documentação oficial do software [6], tomando como base o monitoramento de sistemas virtualizados, disponível no site oficial da plataforma [4], executamos uma instância com o comando abaixo:

```
docker run -d --name=grafana \
--restart always \
-p 3000:3000 \
-e "GF_SERVER_PROTOCOL=http" \
-e "GF_SERVER_HTTP_PORT=3000" \
-v /docker/grafana/data:/var/lib/grafana \
grafana/grafana
```

Tal comando inicializa uma instância container com o Grafana virtualizado na porta 3000 do dispositivo local, podendo ser acessado como na Figura 15 abaixo:

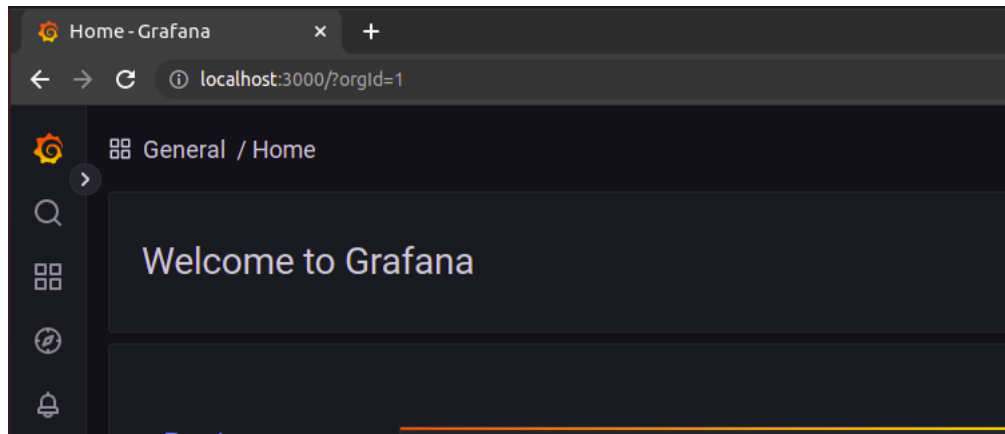


Figura 15: Instância Grafana em localhost

Configurando no Grafana a instância do Prometheus como fonte de dados como na Figura 16, podemos criar queries para criar os quadros de dashboard na aplicação, como abaixo:

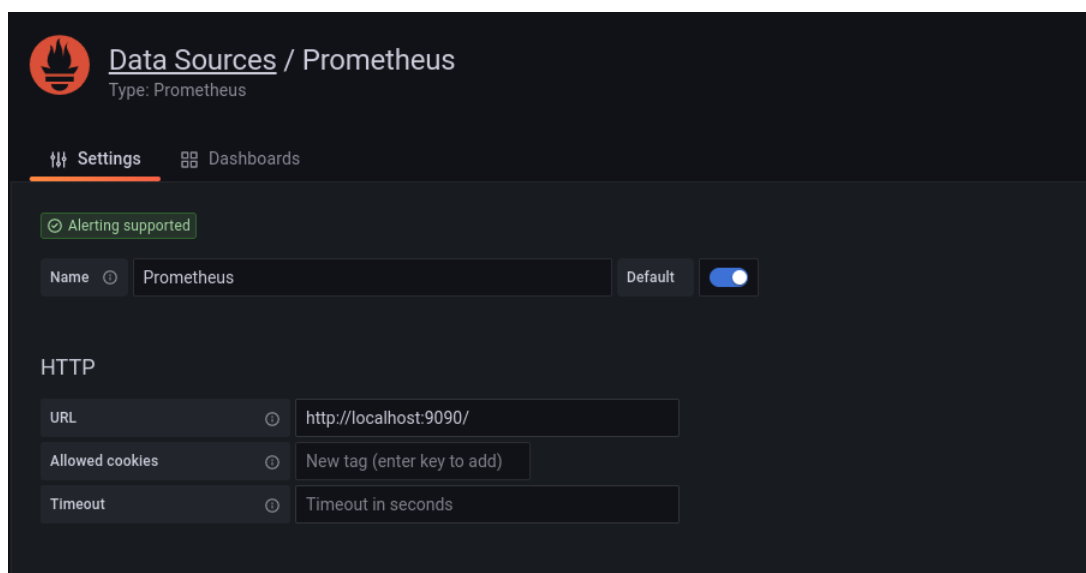


Figura 16: Grafana - Configuração do Prometheus como datasource no Granafa

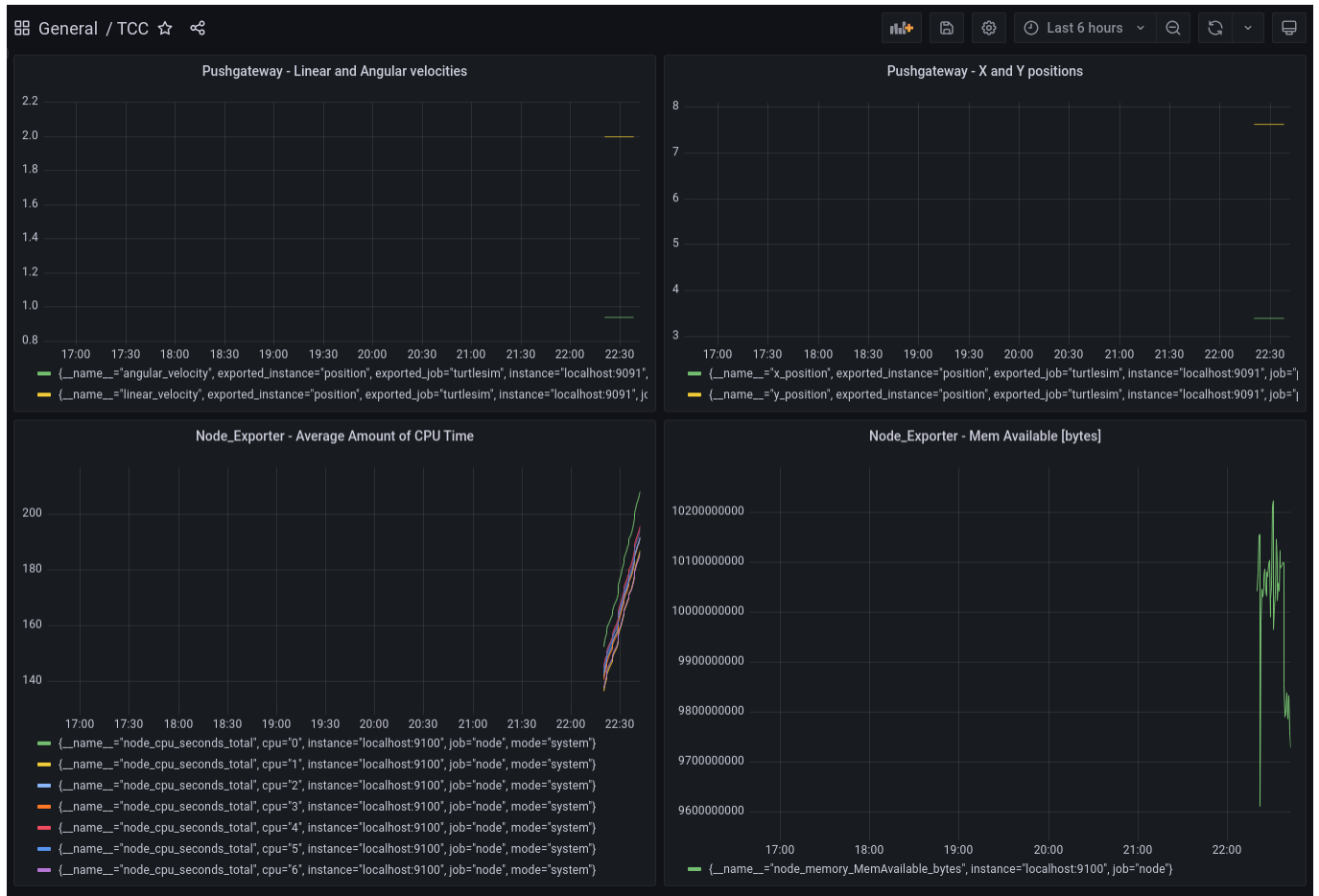


Figura 17: Grafana - Dashboards das métricas do sistema

4.3.4 Construção das métricas e interpretação dos gráficos

Para o monitoramento proposto, no Grafana olhamos para os seguintes grupos de métricas:

- `node_memory_MemAvailable_bytes`: métrica direta do `node_exporter` indicando a memória disponível do sistema em bytes;
- `node_cpu_seconds_total`: métrica direta do `node_exporter` que trás o processamento dos cores da CPU do sistema;
- `x_position` e `y_position`: métricas customizadas enviadas pela aplicação Node.js ao Pushgateway e Prometheus, indicando a posição x e y do robô;
- `angular_velocity` e `linear_velocity`: métricas customizadas enviadas pela aplicação Node.js ao Pushgateway e Prometheus, indicando a velocidade linear e angular do robô;
- `start{exported_job="turtlesim"}`: métrica customizada enviada pela aplicação Node.js ao Pushgateway e Prometheus, indicando o início da operação do robô;

- `colision{exported_job="turtlesim"}:` métrica customizada enviada pela aplicação Node.js ao Pushgateway e Prometheus, indicando quando o robô sofre alguma colisão na simulação;

Na Figura 18 abaixo, podemos ver o dashboard do Grafana das posições x e y do robô operando a um tempo na sua trajetória constante, sem interrupções:

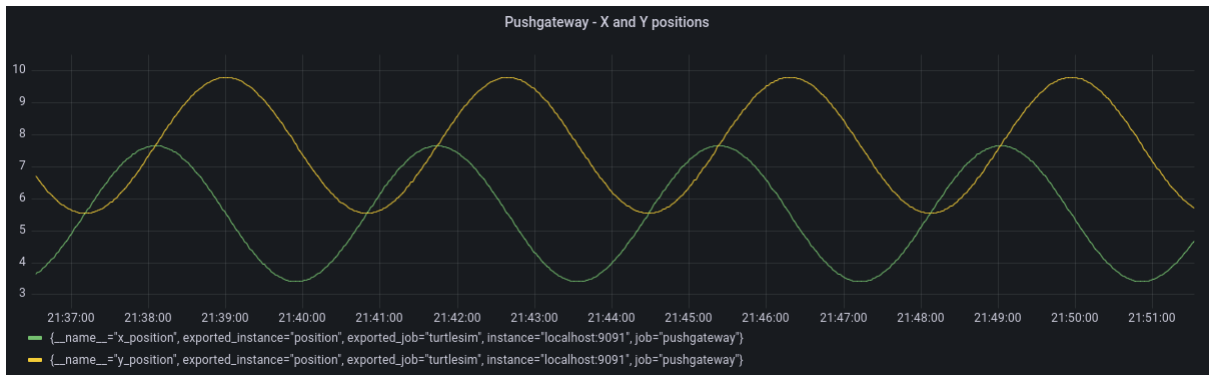


Figura 18: Grafana - Dashboard de posição do robô

4.3.5 Arquitetura e versionamento do projeto

Como resultado das aplicações descritas acima, temos a seguinte arquitetura do sistema:

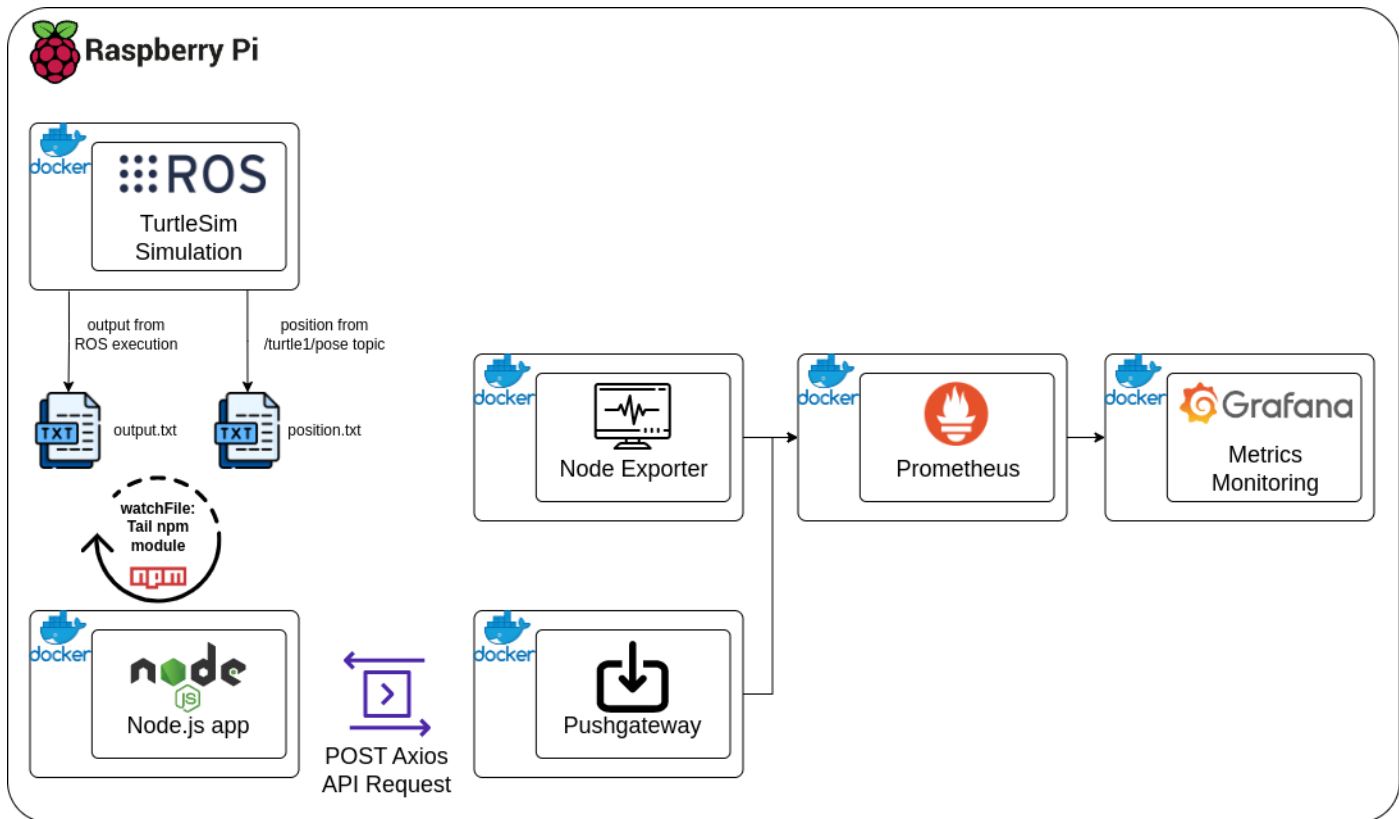


Figura 19: Arquitetura Final - Integrações entre o ROS e as aplicações de monitoramento

Todo o código desenvolvido para a aplicação, bem como o passo-a-passo para a execução do experimento está versionado no GitHub desse projeto [27]

4.4 Externalização do sistema

Para um sistema de monitoramento de um sistema embarcado é interessante tornar os resultados do monitoramento do sistema disponíveis externamente, sem serem acessados apenas nas instâncias locais (<http://localhost:3000/>) da rede que está sendo executado o experimento.

Para isso, é possível utilizar aplicações como o *ngrok* [20], que é capaz de externalizar suas instâncias localhost para um endereço público na internet, podendo configurar autenticação para o endereço ou então limitar por IP.

Alternativas como o *ngrok*, que já funcionam de forma simples, ou então instrumentando um endereço público para a internet utilizando uma instância *nginx* [1] com endereços privados de IP configurados direto no roteador, para expor o localhost. Tais possibilidades são viáveis para fazer com que um sistema embarcado e o seu monitoramento possam ser acessados de endereços web.

Dessa forma, instalando o *ngrok* globalmente para a nossa aplicação, com o comando abaixo, foi possível acessar o monitoramento do sistema a partir de qualquer rede, utilizando um login e senha configurado, como ilustrada na Figuras 22:

```
sudo snap install ngrok
```

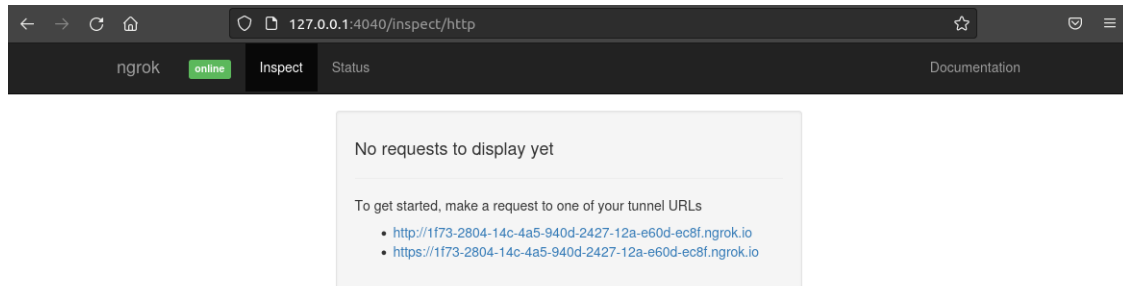


Figura 20: NGROK - Tela inicial

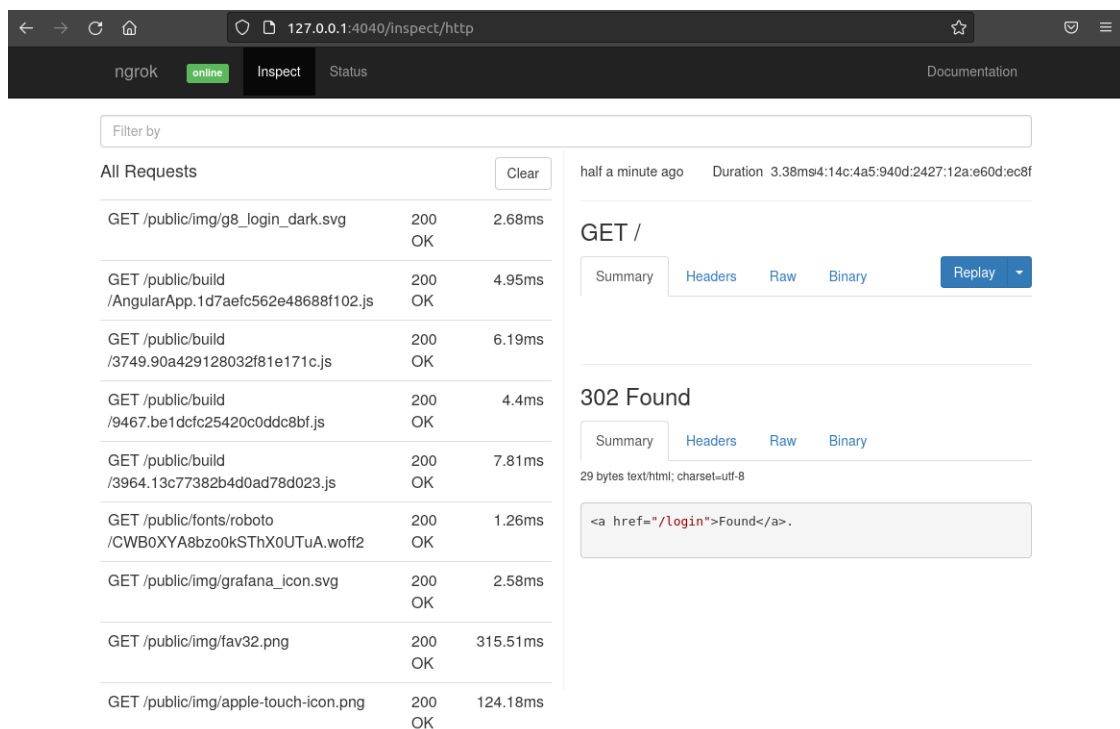


Figura 21: NGROK - Logs quando a aplicação é acessada

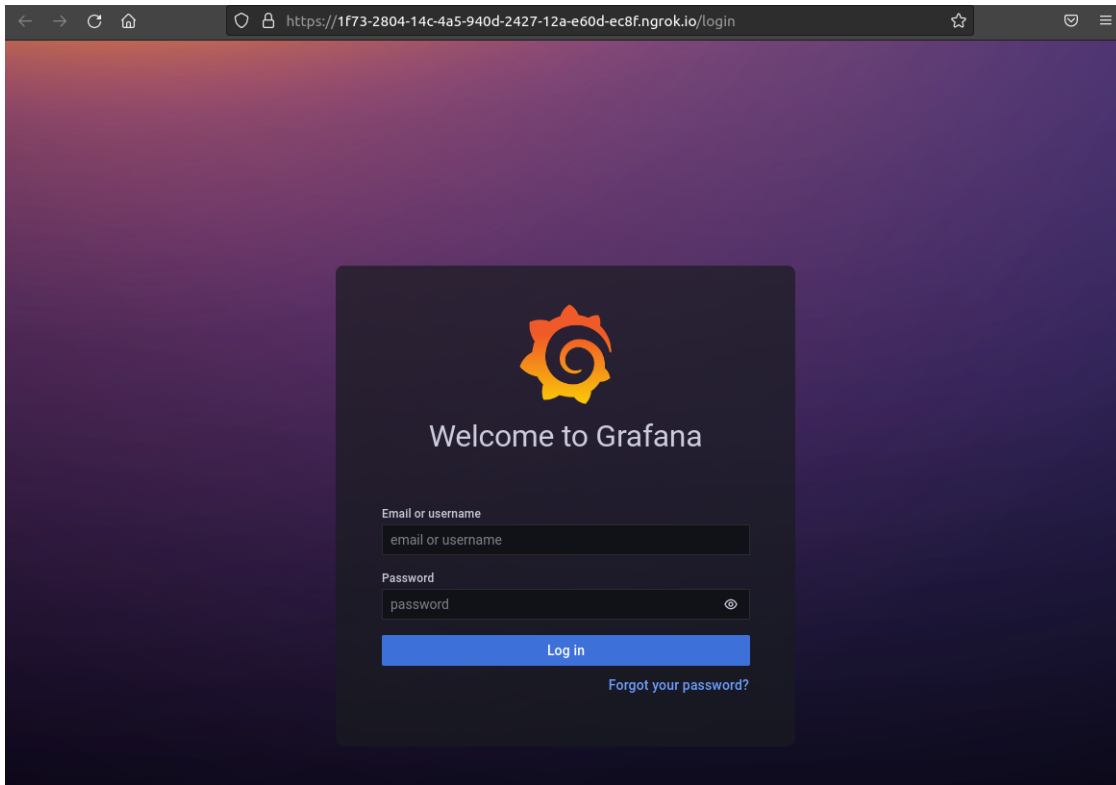


Figura 22: NGROK - Grafana disponibilizado pela aplicação

Na Figura 20 temos a primeira tela do NGROK, que fica disponível localmente no endereço `http://127.0.0.1:4040/inspect/`. No momento em que a aplicação é acessada, como na Figura 21 temos todo o histórico de logs de acesso ao sistema de monitoramento, nos fornecendo mais informações sobre o sistema bem como aumentando a segurança, já que através do *ngrok* é possível limitar o acesso por autenticação ou séries de endereços públicos (IPs).

4.5 Segurança

Além da segurança estabelecida no sistema de monitoramento pela segurança de login/senha e limitações de endereço de IP utilizando o *ngrok* ou como alternativa o *nginx*, podemos também avaliar a segurança do nosso código rodando testes no nível:

- Vulnerabilidades de código no projeto: com aplicações como o *Horussec* [14] é possível verificar problemas de segurança nos módulos de dependência utilizados no nosso sistema, bem como padrões de codificação que poderiam ter alguma possibilidade de causar uma falha de segurança no sistema. Como estamos expondo um endereço para a web, qualquer problema do código ou algum módulo que expõem alguma falha de segurança, pode causar problemas para o nosso sistema ou até ser uma porte de acesso a *malwares* para a nossa máquina;
- Vulnerabilidades nos endereços externalizados: tendo uma porta localhost exposta a um endereço

web pode criar acessos a possíveis ameaças a integridade da nossa aplicação ou até mesmo a máquina que está a executando, com aplicações como *OwaspZap* [7] podemos passar a URL externalizada pelo *ngrok* e fazer um diagnóstico da aplicação.

4.6 Resultados finais e conclusões

Como atividades futuras para dar continuidade a este trabalho pode ser feito um controlador aliado aos logs da trajetória para correção do trajeto do robô, bem como utilizar outros tópicos do sistema ROS para monitorar outros aspectos da simulação, incrementando o sistema de monitoramento pelo Grafana. Além disso, configurando alertas no sistema de monitoramento, pelo Grafana, podemos configurar ações para o sistema executar no momento em que é detectado uma colisão, por exemplo, desligando o movimento do robô. Expondo o controlador do robô para um endereço web e codificando uma aplicação para enviar dados de trajetórias ao robô podem ser um meio de simular um ataque ou mal funcionamento na simulação, testando assim o controlador e o sistema de monitoramento.

Portanto, o experimento se apresentou viável e escalável, podendo ser aplicado a outros contextos. Quanto ao sistema de monitoramento instrumentado, com a estrutura final indicada pela Figura 23 abaixo, foi possível acessar os dados até mesmo fora da rede da aplicação, se mostrando uma conexão segura, devido as camadas de autenticação envolvidas, e também confiável, pela atualização quase que instantânea dos logs do robô, graças ao uso de tecnologias novas no mercado aliado ao sistema já estabelecido no mercado como o ROS.

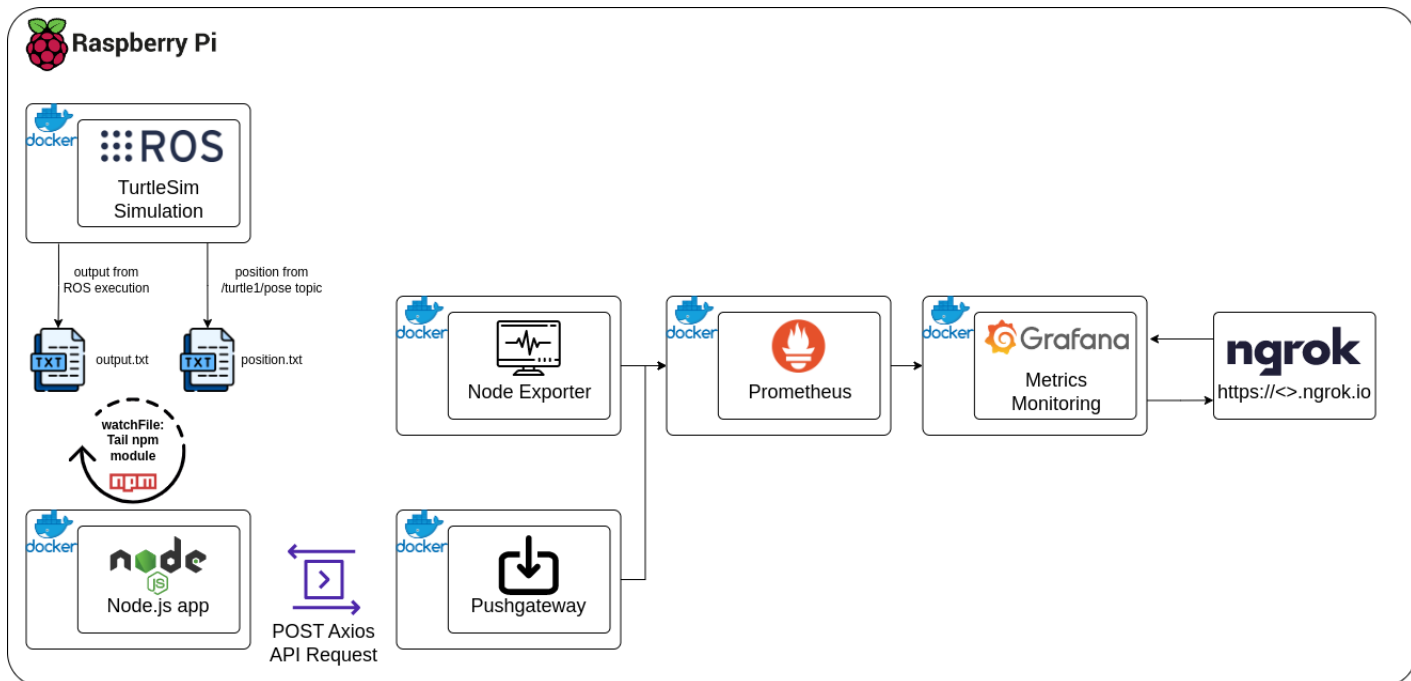


Figura 23: Resultados Finais - Arquitetura Final - Integrações entre o ROS e as aplicações de monitoramento externalizadas em uma URL pública pelo NGROK



Figura 24: Resultados Finais - Grafana - Logs da simulação ROS



Figura 25: Resultados Finais - Grafana - Logs da performance do RaspberryPi

Nos dashboards da Figura 24 podemos ver todos os pontos em que o robô atingiu o limite da simulação e a sua posição, bem como o início do sistema e também a velocidade do robô. Nos dashboards da Figura 25 temos o uso de memória e processamento do sistema.

5 Referências bibliográficas

Referências

- [1] *Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX*. <https://www.nginx.com/>. (Acessado em Dezembro 2022).
- [2] João Pedro Xavier Araújo. *Monitorização de um Sistema Publish-Subscribe ROS para Enumeração e Detecção de Intrusões*. <https://repositorio-aberto.up.pt/bitstream/10216/122484/2/354166.pdf>. (Acessado em Julho 2022). 2019.
- [3] *axios - npm*. <https://www.npmjs.com/package/axios>. (Acessado em Outubro 2022).
- [4] *Docker and system monitoring — Grafana Labs*. <https://grafana.com/grafana/dashboards/893-main/>. (Acessado em Julho 2022).
- [5] *Documentação Docker*. <https://docs.docker.com/>. (Acessado em Abril 2022).
- [6] *Documentação Grafana*. <https://grafana.com/>. (Acessado em Abril 2022).
- [7] *Documentação OWASP ZAP*. <https://www.zaproxy.org/>. (Acessado em Abril 2022).
- [8] *Documentação Podman*. <https://docs.podman.io/en/latest/>. (Acessado em Abril 2022).
- [9] *Documentação Prometheus*. <https://prometheus.io/>. (Acessado em Abril 2022).
- [10] *Documentação RaspberryPi*. <https://www.raspberrypi.com/documentation/>. (Acessado em Abril 2022).
- [11] *Documentação ROS*. <https://www.ros.org/>. (Acessado em Abril 2022).
- [12] DominikN/ros2_docker_examples: Running ROS 2 with docker. Different code samples and examples showing how to run Turtle bot example on one or across multiple hosts. https://github.com/DominikN/ros2_docker_examples. (Acessado em Julho 2022).
- [13] *File system — Node.js v19.2.0 Documentation*. (Acessado em Dezembro 2022). URL: %5Curl%7Bhttps://nodejs.org/docs/latest/api/fs.html#fs%5C_fs%5C_watchfile_filename%5C_options%5C_listener%7D.
- [14] *Home — Horusec*. <https://horusec.io/site/>. (Acessado em Dezembro 2022).
- [15] Raimarius e Byoung Wook Choi Jaeho Park. *Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study — IEEE Journals & Magazine — IEEE Xplore*. <https://ieeexplore.ieee.org/document/9172073>. (Acessado em Dezembro 2022).
- [16] Vasco Lopes. (PDF) *Detecting Robotic Anomalies using RobotChain*. https://www.researchgate.net/publication/334031521_Detecting_Robotic_Anomalies_using_RobotChain. (Acessado em Dezembro 2022).
- [17] *lucagrulla.tail - npm*. <https://www.npmjs.com/package/tail>. (Acessado em Dezembro 2022).
- [18] Ismael Rodrigues Martins e José Luis Zem. *Repositório Institucional do Conhecimento do Centro Paula Souza: Estudo dos protocolos de comunicação MQTT e COaP para aplicações machine-to-machine e Internet das coisas*. <http://ric.cps.sp.gov.br/handle/123456789/107>. (Acessado em Julho 2022). 2015.

- [19] *Monitoring Linux host metrics with the Node Exporter — Prometheus*. <https://prometheus.io/docs/guides/node-exporter/>. (Acessado em Outubro 2022).
- [20] *ngrok - Online in One Line*. <https://ngrok.com/>. (Acessado em Dezembro 2022).
- [21] *Node.js*. <https://nodejs.org/en/>. (Acessado em Outubro 2022).
- [22] Carlos Cambra e Álvaro Herrero Nuño Basurto. *A visual tool for monitoring and detecting anomalies in robot performance — SpringerLink*. <https://link.springer.com/article/10.1007/s10044-021-01053-0>. (Acessado em Dezembro 2022).
- [23] *O que é PaaS? Plataforma como Serviço — Microsoft Azure*. <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-paas/>. (Acessado em Julho 2022).
- [24] University of Virginia Prof. Dr. Madhur Behl. *Curso de Simulação de Sistemas Embarcados*. <https://linklab-uva.github.io/autonomousracing/>. (Acessado em Setembro 2022).
- [25] *Prometheus Pushgateway*. <https://github.com/prometheus/pushgateway/blob/master/README.md>. (Acessado em Outubro 2022).
- [26] *prometheus/node_exporter: Exporter for machine metrics*. https://github.com/prometheus/node_exporter. (Acessado em Outubro 2022).
- [27] *VitaoCs/unicamp-final-paper: This is my final paper for my graduation on Control and Automation Engineering at UNICAMP*. <https://github.com/VitaoCs/unicamp-final-paper>. (Acessado em Novembro 2022).

Apêndices

Todos os scripts da aplicação, incluindo os scripts abaixo para executar e desligar o experimento e o repositório para desenvolver esse trabalho estão versionados no GitHub do projeto *unicamp-final-paper* [27]

A Arquivo index.js para a aplicação em Node.js

```
const ngrok = require('ngrok')
const Tail = require('tail').Tail
const axios = require('axios')
const {
  METRICS_TO_PUSHGATEWAY,
  POSITION_FILE_PATH,
  ROS_OUTPUT_FILE_PATH,
  PUSHGATEWAY_INSTANCE
} = require('./lib/constants')
const { loadConfig } = require('./lib/config')
const {
  ngrokToken,
  ngrokBaseAuth
} = loadConfig()

let collisionWasDetected = false

/*
  Since Grafana has a scrape interval to search the metrics in Prometheus,
  we need to wait this interval to update the metrics values
*/
const sendToWithDelay = async ({
  job,
  instance,
  data,
  delay
}) => {
  return new Promise((resolve, reject) => {
    setTimeout(async () => {
      try {
        await axios({
          method: 'post',
```

```

        url:
        ↪ `_${PUSHGATEWAY_INSTANCE}/job/${job}/instance/${instance}`,
        data,
    })
  } catch (error) {
    console.error('Erros with axios request.')
  }
  resolve()
}, delay);
})
}

const sendToPushgateway = async ({
  identifier,
  value,
}) => {
  const {
    job,
    instance,
    metric,
    delay
  } = METRICS_TO_PUSHGATEWAY[identifier]

  const data = `${metric} ${value}\n`

  await sendToWithDelay({
    job,
    instance,
    data,
    delay
  })
}

const treatPositionLine = async (line) => {
  const [positionIdentifier, value] = line.split(':')
  if (value && METRICS_TO_PUSHGATEWAY[positionIdentifier]) {
    await sendToPushgateway({
      identifier: positionIdentifier,
      value: value.trim(),
    })
  }
}

```

```

}

const treatOperationLine = async (line) => {
  const message = line.split(': ')[1]
  const START_MESSAGE = 'Starting turtlesim'
  const COLLISION_MESSAGE = 'I hit the wall'
  if (message && message.includes(START_MESSAGE)) {
    await sendToPushgateway({
      identifier: 'start',
      value: '1',
    })
    // indicates the beginning of the simulation so we need to shut down this
    ↪ flag after sending it
    await sendToPushgateway({
      identifier: 'start',
      value: '0',
    })
  } else if (message && message.includes(COLLISION_MESSAGE)) {
    collisionWasDetected = true
    await sendToPushgateway({
      identifier: 'collision',
      value: '1',
    })
  } else if (collisionWasDetected) {
    // since it is no longer colliding, we can bring down this flag
    collisionWasDetected = false
    await sendToPushgateway({
      identifier: 'collision',
      value: '0',
    })
  }
}

const createNgrokTunnel = async () => {
  const url = await ngrok.connect({
    authToken: ngrokToken
  })
  console.log(`The ngrok tunnel url is: ${url}`)
}

const createTailsForROS = async () => {

```

```

const outputTail = new Tail(ROS_OUTPUT_FILE_PATH);
const positionTail = new Tail(POSITION_FILE_PATH);

// Reads output file from ROS execution for health checker
outputTail.on('line', async function (data) {
    await treatOperationLine(data)
});
outputTail.on('error', function (error) {
    console.error('[outputOperationStream] Error while reading file: ',
        ↪ error)
});

// Reads position file to check ROS screen location
positionTail.on('line', async function (data) {
    await treatPositionLine(data)
});
positionTail.on('error', function (error) {
    console.error('[positionStream] Error while reading file: ', error)
});

console.log('Tails created!')
}

const main = async () => {
    createTailsForROS()
    createNgrokTunnel()
}

main()

```

B Arquivo config.js com as configurações necessárias para a aplicação Node.js

```

const { PATH_TO_SECRET_KEYS_FILE } = require('./constants')

const requireConfigFile = (module) => {
    delete require.cache[require.resolve(module)]
    return require(module)
}

const loadConfig = () => {

```

```

    const {
      NGROK_TOKEN,
      NGROK_BASE_AUTH
    } = requireConfigFile(PATH_TO_SECRET_KEYS_FILE)

    return {
      ngrokToken: NGROK_TOKEN,
      ngrokBaseAuth: NGROK_BASE_AUTH
    }
  }

  module.exports = {
    loadConfig
  }
}

```

C Arquivo constants.js com os marcadores para a construção das métricas na aplicação Node.js

```

const PATH_TO_SECRET_KEYS_FILE = '../..keys.json'
const POSITION_FILE_PATH = `${process.env.PWD}/apps/ros2_docker_examples/position.txt`
const ROS_OUTPUT_FILE_PATH = `${process.env.PWD}/apps/ros2_docker_examples/output.txt`
const PUSHGATEWAY_INSTANCE = 'http://localhost:9091/metrics'
const GRAFANA_SCRAPE_INTERVAL_POSITION = 100
const GRAFANA_SCRAPE_INTERVAL_OUTPUT = 1000

const METRICS_TO_PUSHGATEWAY = {
  x: {
    job: 'turtlesim',
    instance: 'position',
    metric: 'x_position',
    delay: GRAFANA_SCRAPE_INTERVAL_POSITION
  },
  y: {
    job: 'turtlesim',
    instance: 'position',
    metric: 'y_position',
    delay: GRAFANA_SCRAPE_INTERVAL_POSITION
  },
  theta: {
    job: 'turtlesim',
    instance: 'position',

```



```

        metric: 'theta_position',
        delay: GRAFANA_SCRAPE_INTERVAL_POSITION
    },
    linear_velocity: {
        job: 'turtlesim',
        instance: 'position',
        metric: 'linear_velocity',
        delay: GRAFANA_SCRAPE_INTERVAL_POSITION
    },
    angular_velocity: {
        job: 'turtlesim',
        instance: 'position',
        metric: 'angular_velocity',
        delay: GRAFANA_SCRAPE_INTERVAL_POSITION
    },
    start: {
        job: 'turtlesim',
        instance: 'operation',
        metric: 'start',
        delay: GRAFANA_SCRAPE_INTERVAL_OUTPUT
    },
    collision: {
        job: 'turtlesim',
        instance: 'operation',
        metric: 'collision',
        delay: GRAFANA_SCRAPE_INTERVAL_OUTPUT
    }
}

module.exports = {
    METRICS_TO_PUSHGATEWAY,
    POSITION_FILE_PATH,
    ROS_OUTPUT_FILE_PATH,
    PUSHGATEWAY_INSTANCE,
    PATH_TO_SECRET_KEYS_FILE
}

```

D Script para execução da aplicação

```
#!/bin/bash
```

```

# Cleaning ros2_docker_examples repository
cd apps/ros2_docker_examples/
rm output.txt
rm position.txt
touch output.txt
touch position.txt

# Execute node_exporter
cd ../node_exporter/
nohup ./node_exporter &

# Execute Prometheus
cd ../prometheus/
nohup ./prometheus --config.file=./prometheus.yml &

# Execute Pushgateway
docker run -d --name pushgateway -p 9091:9091 prom/pushgateway

# Init node app with file tails and ngrok
cd ../../
node index.js &

echo -e "[Start Script] Start script done!"

```

E Script para o desligamento da aplicação

```

#!/bin/bash

# Saerching for PID process
PROCESS=($(ps -ef | grep -i index.js | awk '{print $2}'))
PROCESS+=($(ps -ef | grep -i prometheus | awk '{print $2}'))
PROCESS+=($(ps -ef | grep -i node_exporter | awk '{print $2}'))

for processPID in "${PROCESS[@]}"
do
    echo -e "[Stop Script] Killing process: $processPID"
    kill -9 $processPID
done

DOCKER=($(docker ps -aqf "name=pushgateway"))
for containerId in "${DOCKER[@]}"

```

```
do
    echo -e "[Stop Script] Stopping docker container id: $containerId"
    docker stop $containerId
done

docker rm pushgateway

echo -e "[Stop Script] Stop script done!"
```