Análise de desempenho da comunicação entre o ROS 2 e o MicroROS

Gabriel Toffanetto França da Rocha [*] **Giovani Bernardes Vitor** [*]
Rafael Francisco dos Santos [*] **Willian Gomes de Almeida** [*]

[*] *Laboratório de Robótica, Sistemas Inteligentes e Complexos RobSIC, Universidade Federal de Itajubá – Campus Itabira, MG, E-mails: gabriel.toffanetto@unifei.edu.br; giovanibernardes@unifei.edu.br; rsantos@unifei.edu.br; will.almeida@unifei.edu.br*

Abstract: In mobile robotics environment, has been essentially necessary the implementation of distributed systems, compound of any modules, like sensors, computers, embedded systems and IoT devices. Once this demand exists, the second version of Robot Operation System framework, the ROS 2, have communication protocols, libraries and tools for help the implementation of this systems for robotics, given support for real-time applications. However, looking for a wider application of ROS, the MicroROS gives the possibility to embed ROS 2 in a microcontroller and making possible to use the system in low and high levels, connected by an agent. Since this integration exists, the knowledge about the real efficiency and strengths and weaknesses of this communication is little, considering the application in a robotic system. Thus, for evaluate the communication benchmarking, latency, jitter, data loss and determinism characteristics will be analyzed, testing the growth of callbacks processing and rise of data input frequency. Thereby, the results show that the MicroROS susceptible to two situations: (i) the quality of service influence on system response and (ii) for estable operation frequencies, the system shows deterministic and real-time processing. Lastly, in the comparison of SingleThread and MultiThread tests, was prove that MicroROS have a critical point for latencies and topic loss registered.

Resumo: No âmbito da robótica móvel, a necessidade da implementação de sistemas distribuídos, compostos de módulos como sensores, computadores, sistemas embarcados e dispositivos de IoT têm se tornado essencial. Posto isso, a segunda versão do *framework Robot Operation System*, o ROS 2, detém de protocolos de comunicação, bibliotecas e ferramentas para o auxílio da implementação de tais sistemas na robótica, trazendo ainda suporte para aplicações de tempo real. Porém, visando a aplicação ainda mais ampla do ROS, o MicroROS traz a possibilidade de embarcar o ROS 2 à um microcontrolador, em um sistema embarcado de tempo real, possibilitando a aplicação do sistema em baixo e alto nível, interligados por um agente. Uma vez que existe essa integração, pouco se sabe sobre a real eficiência dessa comunicação, seus pontos fortes e fracos, considerando a sua aplicação em um sistema robótico. Logo, o presente estudo busca analisar o desempenho da comunicação entre os dois, analisando a latência, o *jitter*, a perda e pacotes e o determinismo da comunicação, ensaiando o aumento de processamento das *callbacks* e da frequência de envio de dados. Como resultado, observa-se que o MicroROS é sensível a duas situações: (i) o comportamento do sistema é influenciado pela qualidade de serviço e (ii) para frequências onde a comunicação se mostra estável, o sistema se mostrou determinístico e de tempo real. Por fim, com a comparação de ensaios *SingleThread* e *MultiThread*, foi constatado que o MicroROS possui um ponto crítico para as latências e perdas de pacotes registradas.

Keywords: Robotics; Autonomous robotics; Mobile robotics; Real-time; Embedded systems; Distributed systems; Communication;

Palavras-chaves: Robótica; Robótica autônoma; Robótica móvel; Tempo real; Sistemas embarcados; Sistemas distribuídos; Comunicação;

1. INTRODUÇÃO

Na robótica móvel é cada vez mais presente o uso de sistemas distribuídos para a construção de veículos autônomos que contam com sensoriamento, por exemplo, câmeras estéreo, LiDARs, GPS, IMU e encoders, e com controladores, como sistemas embarcados e *drivers*, para poderem identificar o ambiente e tomar suas próprias decisões. Para que toda essa a integração e comunicação entre os módulos seja facilitada, o *Robot Operation Sysyem* (ROS) (Open Robotics, 2021b) é um *framework* que conta com um conjunto de bibliotecas e ferramentas para a conexão entre diferentes dispositivos do robô, que utiliza de uma comunicação simples e limpa, o que facilita a implementação e análise dos sistema robótico, como feito previamente no veículo autônomo desenvolvido em de Almeida et al. (2019).

Mostrado por Rojas-Rueda et al. (2020), o uso de veículos autônomos tem uma relação direta com a segurança pública da população, principalmente em relação à ocorrência de acidentes de trânsito. Tais ocorrências são geradas principalmente por conta de excesso de velocidade, distração do motorista e também por manobras irregulares. Tendo em vista essas causas, pesquisadores e entidades de vários setores automobilísticos estimam que os veículos autônomos podem levar a uma redução drástica desses acidentes, salvando inúmeras vidas. Porém, para que um veículo possa oferecer a segurança desejada, seu sistema deve ser confiável para que não haja falhas e as decisões sejam tomadas em tempo real, de forma que a navegação se torne fluida e segura.

Já em sua segunda versão, o ROS 2 (Open Robotics, 2022b) traz fortemente o conceito de sistemas distribuído (Woodall, 2019), operando em módulos e oferecendo também suporte para o funcionamento em tempo real, o que não era possível em sua primeira versão. Abrangendo ainda mais o ROS, foi desenvolvido o MicroROS, *framework* que possibilita a implementação do ROS 2 em microcontroladores (MicroROS, 2022b), sendo utilizado sobre um sistema operacional de tempo real (RTOS). Com a união de tais sistemas, pode-se estabelecer a comunicação entre o ROS 2 em um computador e o MicroROS em um sistema embarcado. Esse feito, que é de grande interesse para a robótica autônoma, possibilita que o ROS seja executado nas camadas de alto e baixo nível.

Dada a ótica da execução em tempo real, é importante garantir o determinismo do sistema, ou seja, que o robô irá operar de uma forma conhecida, respeitando os parâmetros impostos. Isso traz confiabilidade e segurança para um veículo autônomo, garantindo que poderá responder aos comandos enviados em um tempo conhecido e constante. A perda de informações ou um alto *delay* para a execução de uma tarefa em um robô autônomo podem causar falhas na operação, resultando em possíveis erros que poderão causar acidentes com vítimas. Isso reforça a necessidade de uma comunicação confiável entre alto e baixo nível, garantindo que os comandos enviados para o robô sejam executados de forma íntegra, assegurando seu processamento em tempo real.

O artigo de Yang and Azumi (2020) realiza a análise de desempenho da comunicação *Push* entre dois nós do ROS 2 utilizando o executor *Callback-level-group*, desenvolvido no MicroROS para a execução em tempo real. Focando na robótica móvel autônoma e na modularização do sistema, alternada entre sistemas RTOS e de propósito geral (GPOS).

O presente trabalho busca realizar a análise de desempenho da comunicação *Push* entre dois nós. Diferentemente do trabalho apresentado por Yang and Azumi (2020), este trabalho foca na execução desses dois nós em sistemas distintos, sendo um processado no ROS 2 Foxy e outro embarcado ao MicroROS, interconectados através de um agente serial, comparando os resultados dos testes de desempenho utilizando diferentes executores.

Este trabalho foi organizado como se segue: A Seção 2 apresenta o embasamento teórico para a realização dos ensaios e mostra o nicho de pesquisa aplicado ao assunto nos últimos anos. Já na Seção 3, é explanado de forma detalhada os testes que foram realizados e quais seus objetivos. Seguindo, na Seção 4, os dados coletados na seção anterior são exibidos e discutidos e, por fim, as conclusões tiradas sobre o posto trabalho estão descritas na Seção 5.

2. FUNDAMENTOS TEÓRICOS

O conceito de tempo real pode ser interpretado de várias formas, mas sempre está relacionado à ideia de rapidez. A definição dada por Laffey et al. (1988) dita que um sistema de tempo real é um sistema que responde aos dados recebidos em uma taxa mais rápida do que ele recebe dados, não provocando *delay* entre a entrada e saída causado pelo fluxo de dados. Com isso, pode-se dizer que um sistema de tempo real é um sistema que tem a capacidade de garantir que após um certo período de tempo constante, a resposta à uma entrada será dada, independente da entrada e do estado atual do sistema. Trazendo esses conceitos à robótica móvel, pode-se garantir com um sistema de tempo real que ao enviar um comando para o robô, ele irá responder à esse em um tempo conhecido, independente da ação que ele estiver executando, ou seja, trazendo confiabilidade na execução dos processos. Além de um sistema ser de tempo real, pode-se classificar o seu determinismo, ou seja, se o mesmo se comporta de forma constante em caso de acontecimentos constantes. Como mostrado por Adomat et al. (1996), a distância entre a melhor e pior resposta de um sistema pode ser utilizada para mensurar o determinismo, sendo que, quanto menor a variação entre o melhor e pior caso, mais determinístico é o sistema.

Como uma forma de dar um perfil de funcionamento à comunicação em sistemas distribuídos, a qualidade de serviço (QoS) da comunicação, dada por Tournier and Babau (2004), é utilizada. A QoS é baseada na garantia de alguns serviços, sendo um deles é a latência, tempo que uma mensagem, ação ou evento leva para chegar ao seu destino final. Essa propriedade é relevante para a robótica autônoma, uma vez que o tempo de reação do sistema à uma entrada de controle faz toda a diferença

ISSN: 2525-8311

DOI: 10.20906/CBA2022/3659

na sua navegação, o envio desse comando pode influenciar diretamente no controle do robô. Outro atributo é a taxa de tráfego de dados, que representa a quantidade de dados que são enviados ou recebidos em um determinado período de tempo, importante para que, posto o tamanho de uma mensagem que deve ser transmitida, saber quanto tempo ela irá tardar para ser totalmente enviada. Por fim, o último requisito é o tempo para detecção de erro, intervalo máximo que o sistema demora para perceber que houve uma falha na comunicação. Além disso, outro conceito de suma importância que deriva da latência é o *jitter*, uma variável estatística que representa a variação da latência durante o envio de vários pacotes, obtido por meio do desvio padrão das várias tomadas de latência.

A comunicação *Push* ocorre quando as informações são enviadas para o seu destinatário sem a necessidade de uma requisição, sendo implementada no ROS 2 por meio de tópicos. O QoS aplicado à ela traz requisitos mais rebuscados e funciona a partir do pressuposto que o perfil da qualidade de serviço entre *publisher* e *subscriber* são compatíveis (Knapp, 2019). Dentro desse perfil, a política de QoS está baseada no *History*, que rege quantas mensagens devem ser mantidas de forma local, *Durability*, se o *publisher* consegue fornecer mensagens antigas do histórico, *Reliability*, que garante que as mensagens enviadas precisam chegar ao seu destinatário e *Lifespan*, métrica para definir o tempo de validade de mensagens ainda não enviadas. Trazendo um novo conceito ao meta sistema operacional ROS, a qualidade de serviço pode conter funções *callback* baseada em eventos, o que traz para si a política de *Deadline*, a frequência miníma que um *publisher* deve enviar mensagens e também a de *Liveliness*, que realiza a verificação se o *publisher* ou *subscriber* estão *online* e funcionando e decide qual tipo de *flag* os mesmos devem fornecer para isso (Open Robotics, 2021a).

Para que todas as funcionalidades do ROS possam operar, um importante componente são os executores, responsáveis por gerenciar a execução das tarefas. No ROS 1, o gerenciamento era realizado por meio de uma *thread* que recebia e organizava todas as mensagens recebidas em uma fila, sem a existência de nenhuma manipulação na ordem de execução, sendo ela na forma FIFO (*First Input First Output*) (Open Robotics, 2022a). Essa configuração de execução não possui nenhum tipo de otimização ou prioridade para a execução das tarefas, o que dificulta inteiramente a execução em tempo real do sistema. Nesse sentido, o ROS 2 incorpora o conceito e ferramentas de DDS (*Data Distribution Service*), onde as mensagens ficam armazenadas para serem executadas. Quando é identificado que o tempo de processamento das funções de *callback* é menor que o intervalo de tempo de ocorrência de eventos no sistema, o executor processa as tarefas em uma fila sequencial, uma vez que não trará atrasos na fila de execução. Porém, uma vez que o processamento pode levar mais tempo que o período da chegada de mensagens, as tarefas são organizadas a partir de um algoritmo *round-robin*, contudo, priorizando sempre o processamento dos *timers* acima das outras *callbacks*.

Os executores da biblioteca rclcpp, na segunda versão do ROS, podem trabalhar de forma *SingleThread*, onde cada executor terá apenas uma *thread* para executar suas tarefas ou *MultiThread*, podendo executar tarefas em paralelo. Para o segundo caso, além da *thread* principal, as funções de *callback* são organizadas em grupos e cada grupo é adicionado a uma nova *thread*, onde, as tarefas podem ser executadas de forma sequencial ou em paralelo. Porém, os executores padrão do ROS 2 não são adequados para a execução em tempo real, o que ocorre devido a possibilidade de *callbacks* de baixa prioridade impedirem a execução de *callbacks* de alta prioridade. Além disso, não há um controle explícito sobre a ordem em que as tarefas serão executadas, nem um escalonamento de tarefas preparado para a análise no tempo. A falta também de um controle próprio dos gatilhos para tópicos específicos é mais um motivo para não ser possível garantir o processamento em tempo real utilizando os executores nativos. De outra forma, o executor rclcpp padrão do ROS 2 provê de bons aspectos, como latência ponto-a-ponto limitada, *jitter* baixo para respostas de causa e efeito, processamento determinístico e tempo de resposta reduzido em situações de sobrecarga (MicroROS, 2022a). Também é importante observar que a configuração de QoS influência diretamente na forma como o executor irá controlar a execução das tarefas, podendo levar a o aumento de tempo das mesmas ou até da repetição da execução de certas tarefas, alterando propriedades importantes da comunicação como a latência e a perca de tópicos. Foi desenvolvido recentemente outro executor para a biblioteca rclcpp, o *Callback-Group-Level*, que traz para o ROS 2 o conceito de prioridades das tarefas, possuindo níveis de prioridade para os executores, sendo eles em ordem decrescente de prioridade: RT-CRITICAL, onde a tarefa irá ser executada em tempo real, e BEST-EFFORT, onde a tarefa é executada no melhor tempo possível, possibilitando a garantia que as tarefas com maior prioridade realmente sejam executadas em tempo real, como utilizado por Yang and Azumi (2020).

Com foco na atuação em sistemas RTOS, como o FreeRTOS (FreeRTOS, 2021), NuttX (The Apache Software Foundation, 2019) e Zephyr (Linux Foundation, 2021), o MicroROS tem um novo executor, o rclc executor, baseado no executor rclcpp do ROS 2, mas com foco em garantir a execução em tempo real e o determinismo, integrando atividades de tempo real ou não. Dentre suas funcionalidades, o executor pode organizar o processamento das tarefas de diferentes formas. Uma delas é a execução sequencial, onde a ordem como os *callbacks* serão executadas é definida por configuração prévia, e as funções podem ser configuradas para serem executadas sempre ou somente quando houver novas mensagens. Além disso, uma sequência de tarefas pode também ser executada de acordo com um gatilho, de um *timer* ou evento, executando todas as *callbacks* ou somente as que tiverem entradas de dados naquele momento. Também é implementado no executor rclc, a execução em tempo real, que por meio de um gatilho acionado por *timer* e da organização da tarefas por meio do LET (*Logical Executor*), as tarefas são divididas por períodos de execução, garantindo que o sistema possa ser executado em tempo real (MicroROS, 2022a).

3. MATERIAIS E MÉTODOS

3.1 Hardware

O trabalho lida com dois níveis de *hardware*, o computador, que engloba o sistema do ROS 2, e o microcontrolador

que sustenta o sistema embarcado com o MicroROS. O computador utilizado possui um processador Intel Core i7-4790, Quad Core com *clock* de $3,60$ GHz, 16 GB de memória RAM, uma placa de vídeo AMD RDNHD R5 240 de 1 GB e utiliza o sistema operacional Ubuntu 18.04 LTS. Já o sistema embarcado é dotado de um microcontrolador ARM STM32F103RCT6 com *clock* externo de 8 kHz. Como programador e *debugger* do microcontrolador utilizado, foi empregada a placa STM32F4 Discovery com ST-LINK.

A comunicação entre PC e sistema embarcado foi feita através da comunicação serial assíncrona (UART). Foi utilizado o protocolo RS232 entre o sistema embarcado e o computador, com uma taxa de transferência de dados de 115200 b/s. Essa conexão é abstraída por meio do protocolo Micro XRCE-DDS , que através de um agente, decodifica as mensagens enviadas via serial pelo MicroROS para o protocolo de RTPS do ROS 2 e vice-versa (Micro-ROS, 2022c). Dessa forma, é realizada uma ponte entre os dois sistemas, como mostrado na Figura 1.

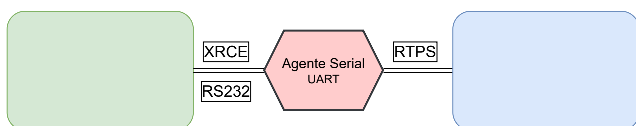

Figura 1. Estrutura de *hardware* para a comunicação entre ROS 2 e MicroROS.

3.2 Software

A parte de *software* é constituída pelo *framework* ROS 2 Foxy, executado no computador, e pelo MicroROS embarcado sobre o FreeRTOS no microcontrolador. O ROS é um sistema formado por diversos blocos modulares chamados de nós, que são interligados por protocolos internos de comunicação, constituindo um sistema distribuído. É possível por meio da ferramenta *rqt_graph*, traçar uma visão por meio de um grafo da área de trabalho do ROS, mostrando seus ambientes, nós, tópicos e como eles interagem entre si por meio de setas.

O sistema no geral é formado de dois nós, um no *Ping*, presente no ROS 2 e um nó *Pong*, embarcado no MicroROS. O nó de *Ping* tem a função de enviar o tópico de *Ping* e armazenar o *timestamp* no qual essa informação foi enviada. Além disso, ele recebe a resposta do *Pong* com a informação indexada para poder comparar o tempo de envio e de chegada de cada tópico, obtendo assim sua latência. O nó de *Pong* tem como única função receber o *Ping*, simular a execução de alguma tarefa por meio de um *delay* chamado de *Busy-loop*, e publicá-lo no tópico de resposta, como feito por Lange (2018).

Para um ensaio de Ping-Pong simples, contendo apenas um tópico, a configuração do sistema é dada pelo grafo apresentado na Figura 2. Pode-se observar que a integração da área de trabalho do ROS 2 com o MicroROS somente através dos tópicos. Observa-se ainda que o agente que interliga as duas esferas do sistema não aparece no grafo, ou seja, ele faz somente uma ponte entre o computador e o sistema embarcado, possibilitando o envio e recebimento de informações na forma padrão de comunicação do ROS, sendo transparente para a camada de aplicação.

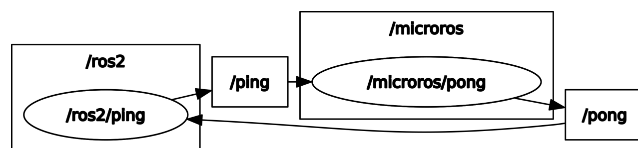

Figura 2. Grafo RQT da topologia do sistema ROS para um tópico de Ping-Pong.

Já considerando um ensaio mais complexo e mais exaustivo para o sistema, onde três tópicos de *Ping* enviam mensagens para o microcontrolador em sequência, vê-se que a estrutura é similar, com os tópicos atuando em paralelo entre os dois nós, como mostrado na Figura 3.

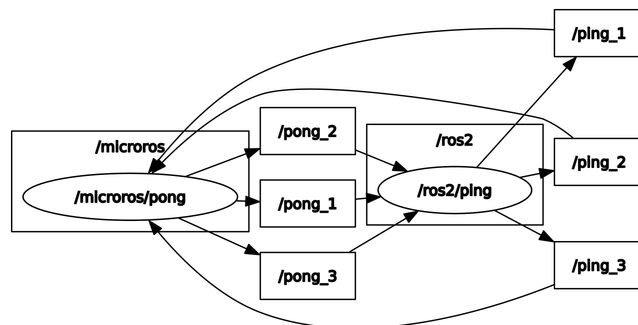

Figura 3. Grafo RQT da topologia do sistema ROS para três tópicos de Ping-Pong.

3.3 Contextualização dos ensaios

Para a validação e obtenção da comparação dos resultados das diferentes possíveis performances da comunicação entre o ROS 2 e o MicroROS, foram projetados três ensaios, sendo eles: Ping-Pong com apenas um tópico e executor *SingleThread*, Ping-Pong com três tópicos e executor *SingleThread* e Ping-Pong com três tópicos e executor *MultiThread*, todos desenvolvidos na linguagem de programação C++ [1] (Lange, 2021). Com os ensaios, a principal variável de interesse é a latência, ou seja, período de tempo entre o envio do *Ping* e o recebimento do *Pong*, que são armazenadas em um arquivo para processamento posterior.

São enviados pacotes de dados de 16 bytes, compostos de 4 bytes de *payload* (informação) e 12 bytes de *headers* do protocolo XRCE, e para cada ponto de operação do ensaio, são coletadas 20 amostras de 200 tópicos transmitidos, com intuito de amenizar variações causadas pela ocupação do CPU. Para o ensaio de simulação do carregamento do sistema, o tempo de *Busy-loop* é incrementado para analisar o impacto desta ação no funcionamento da comunicação. Inicialmente é considerado o período entre o envio de *Ping* de 50 ms, possuindo então uma frequência dos tópicos de 20 Hz, ponto de operação onde o MicroROS apresenta respostas satisfatórias. Com o intuito também de verificar o impacto das diferentes qualidades de serviço, serão feitos os testes para os tópicos com QoS *reliable* e *best-effort*. Após a obtenção das latências realizando os ensaios, os dados coletados foram tratados por meio do uso do *software* MATLAB, para extração de novas informações,

[1] Disponível em: `github.com/Robsic/uros_benchmarking`

como o *jitter* e a perda de tópicos na comunicação, além da representação gráfica desses valores encontrados.

A topologia de ensaio para o Ping-Pong com executor *SingleThread* e apenas um tópico é mostrada na Figura 4, onde existem dois nós, em dispositivos diferentes, conversando através de um agente serial. Já para o teste utilizando três tópicos e mantendo executor *SingleThread*, têm-se agora uma maior quantidade de *callbacks* sendo executadas na mesma *thread*, o que já traz ao sistema uma maior carga de processamento e sua topologia é mostrada na Figura 5. Para o ensaio com a qualidade de serviço *best-effort*, o ROS 2 tem seus *publishers* e *subscribers* configurados com o *SystemDefautsQoS* e o MicroROS inicia os tópicos na configuração *best-effort*. Já ao utilizar o QoS configurado como *reliable*, no ROS 2 é utilizado o *ParametersQoS* como parâmetro dos tópicos e no MicroROS os *publishers* e *subscribers* são declarados como *default*. O mesmo é feito para o uso do executor *MultiThread*, explicitado na Figura 6, onde cada *callback* do ROS 2 está adicionada a uma *thread* diferente, possibilitando que as mesmas possam atuar em paralelo, e da mesma forma os ensaios são repetidos com diferentes qualidades de serviço.

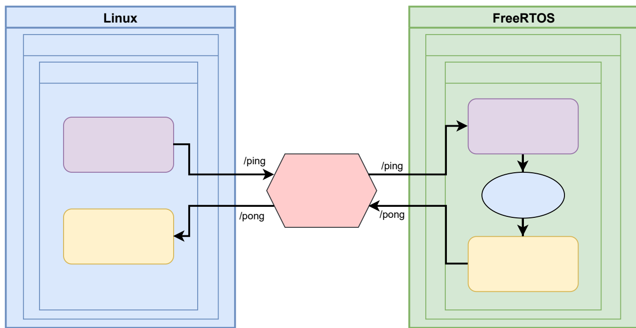

Figura 4. Diagrama da comunicação *Push* entre computador e sistema embarcado através de um agente serial.

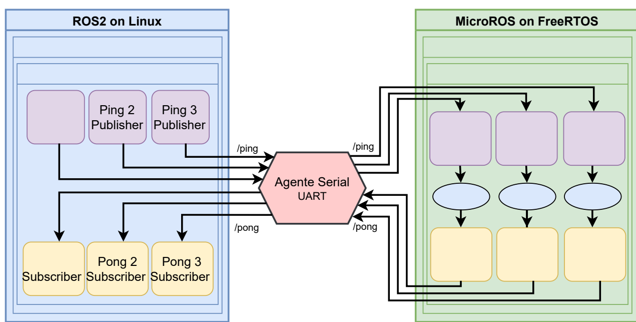

Figura 5. Diagrama do ensaio de Ping-Pong para três tópicos e executor *SingleThread*.

Por fim, é importante ter o conhecimento da quantidade de dados que podem ser enviados pelo sistema, e com a intenção de traçar os pontos críticos de frequência máxima de envio de dados para o MicroROS, será reutilizada a topologia de ensaio *SingleThread* com Ping-Pong triplo para validar o comportamento de latência e perca de tópicos para diferentes frequências de *Ping*. Com essa topologia que representa melhor um cenário real, onde existem vários tópicos, será possível observar o comportamento das respostas mediante o aumento do fluxo de entrada de dados, em diferentes qualidades de serviço.

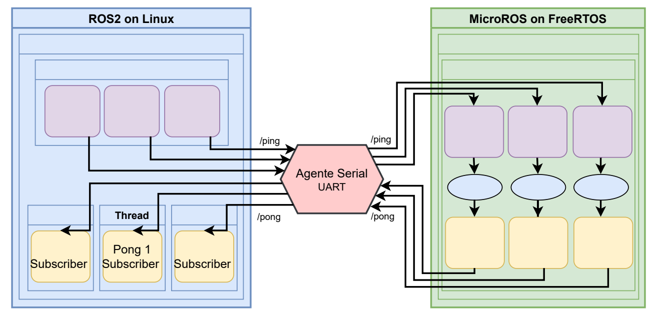

Figura 6. Diagrama do ensaio de Ping-Pong para três tópicos e executor *MultiThread*.

4. RESULTADOS

4.1 Ping-Pong simples SingleThread

Ao realizar o ensaio de Ping-Pong simples com o executor *SingleThread*, têm-se uma noção de referência do funcionamento do sistema, uma vez que a execução é feita de forma isolada. É visto na Figura 7 os resultados das médias de latência, *jitter* e perca de tópicos para cada ponto de *Busy-loop*, variando-o de 0.3 ms até 3 ms com passo de 0.3 ms, considerando a qualidade de serviço *best-effort* e *reliable*. Para a primeira qualidade de serviço percebe-se que a latência aumenta levemente com o aumento do tempo de execução da *callback*, resultado esperado devido ao aumento do tempo entre a chamada da função e a publicação do tópico de *Pong*. Percebe-se também que o *jitter* é baixo para cada amostra, mostrando que há pouca variação de latência entre os tópicos enviados. A perda de tópicos também cresce com o aumento do *Busy-loop*, representando uma perda máxima de 2% dos tópicos enviados.

Comparando análise de desempenho com diferentes qualidades de serviço, percebe-se que a latência para os tópicos *reliable* é maior, assim como seu *jitter*, devido ao fato da necessidade de confirmação de recebimento da mensagem, o que faz com que o tráfego de dados para enviar uma mensagem seja maior, aumentando assim o tempo para enviá-lo. Apesar disso, é notável a redução da perda de tópicos, se mantendo abaixo dos 0.5% de perda, valor que se mantém relativamente estável com o aumento do tempo de processamento da *callback*.

Uma vez que o período em que os *Pings* são publicados é de 50 ms, a latência máxima registrada está bem abaixo desse valor, podendo julgar assim que o sistema está operando na forma de tempo real. Devido à baixa variação da latência com a mudança do *Busy-loop*, o sistema apresentou o comportamento esperado para o determinismo aplicando o QoS *best-effort*.

4.2 Ping-Pong triplo SingleThread

Simulando uma situação mais realística, durante o funcionamento de um robô autônomo, diversos tópicos podem trafegar entre o ROS 2 e o MicroROS. Nesse caso, ao executar o ensaio do Ping-Pong triplo com execução em apenas uma *thread*, variando o *Busy-loop* da mesma forma do experimento anterior, têm-se um aumento considerável do estresse tanto de processamento do microcontrolador

ISSN: 2525-8311

DOI: 10.20906/CBA2022/3659

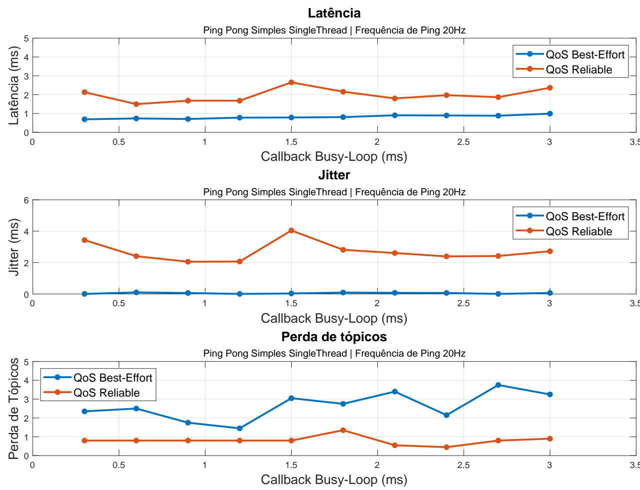

Figura 7. Resultados ensaio Ping-Pong simples com executor *SingleThread* para qualidade de serviço *reliable* e *best-effort*.

como também do protocolo de comunicação. Esse carregamento é perceptível ao analisar os gráficos de latência e perda de tópicos, na existência de um deslocamento vertical entre as curvas dos tópicos 1, 2 e 3, como mostrado nas Figuras 8 e 9. O tópico 1 que é enviado primeiro, apresenta uma menor latência e uma menor perda de tópicos. Já os tópicos 2 e 3 apresentam valores crescentes em relação ao seu anterior. Isso se dá pelo processamento em apenas uma *thread*, onde a chamada da *callback* do tópico 1 atrasa a *callback* do tópico 2 e assim por diante. Como ambas as três curvas de latência apresentam o mesmo perfil, o atraso causado na execução não altera seu tempo de execução, adicionando apenas um *delay* para que a mesma seja processada. Esse atraso, para poder ser processada, leva ao aumento da perda de tópicos, uma vez que podem chegar novos tópicos enquanto a *thread* está bloqueada em uma *callback* e os mesmos serem perdidos.

Para a qualidade de serviço *best-effort*, as latências foram próximas às dos ensaios com apenas um tópico de Ping-Pong, sendo um pouco maiores apenas para os tópicos 2 e 3. A variação da latência entre os tempos de *Busy-loop* também se mantiveram baixas, porém apresentando um alto *jitter* entre amostras para o tempo de processamento maior que 1.8 ms. Já considerando os tópicos com perfil *reliable*, a latência aumentou consideravelmente, com menos variação e um *jitter* menor entre as amostras de cada ponto de operação. Apesar disso, devido ao escalonamento e atraso de execução das tarefas ainda mais acentuado devido à confirmação de recebimento, a perda de tópicos aumentou, principalmente para o tópico 3, devido aos atrasos para o processamento dos tópicos recebidos periodicamente.

Como houve pouca variação de latência, a resposta continua sendo consideravelmente determinística isoladamente, apesar disso esse determinismo decaí ao comparar os três tópicos, uma vez que mesmo executando a mesma tarefa e enviados em sequência, suas latências são diferentes. Entretanto, o tempo de resposta prosseguiu sendo consideravelmente menor que o tempo de envio de dados, apresentando uma boa execução em tempo real. O maior problema visto é a perca de tópicos, que alcançou os 30% para o QoS *reliable* e os 10% para o *best-effort*, deixando a desejar na confiabilidade do recebimento das mensagens.

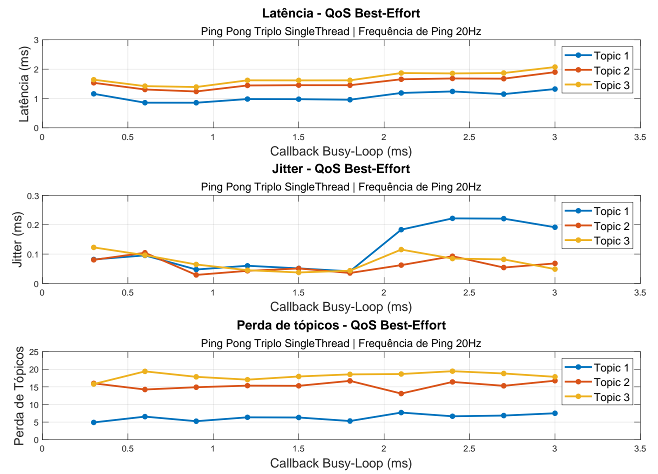

Figura 8. Resultados ensaio Ping-Pong triplo com executor *SingleThread* para qualidade de serviço *best-effort*.

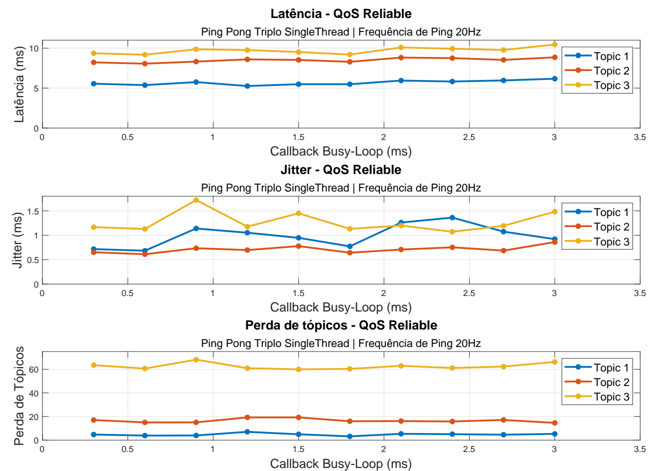

Figura 9. Resultados ensaio Ping-Pong triplo com executor *SingleThread* para qualidade de serviço *reliable*.

4.3 Ping-Pong triplo MultiThread

Executando o teste com mais de um tópico em *MultiThread*, têm-se uma melhor versatilidade na execução devido a possibilidade das *threads* serem processadas em paralelo. Para isso o ROS 2 cria uma *thread* para cada *callback*, porém devido à arquitetura utilizada para implementação do MicroROS no FreeRTOS, no sistema embarcado continua existindo apenas uma *thread* para processar todas as funções que recebem os *Pings*. Dessa forma, pode se observar que tanto para a qualidade de serviço *best-effort*, presente na Figura 10, não houve muita diferença na latência da comunicação. O mesmo ocorre comparando as latências para a QoS *reliable*, mostrada na Figura 11. Com o ensaio feito utilizando o executor *SingleThread*, o tempo de resposta obtido é minimamente menor, se equiparando ao experimento com apenas uma *thread*. O mesmo ocorre também para a perda de tópicos, que para os dois casos não apresenta muitas diferenças em comparação ao experimento anterior.

ISSN: 2525-8311

DOI: 10.20906/CBA2022/3659

Visto isso, era esperado que, com o aumento das *threads*, o sistema opere de forma mais rápida devido ao processamento em paralelo. Como os resultados não demonstram tal expectativa, pode-se dizer que o gargalo para a latência e perda de tópicos está sendo dado pelo MicroROS, que apresenta somente uma *thread* para processar as tarefas está gerando *delays* e vindo a perder dados. De toda forma, observando as curvas de latência, o sistema obteve uma menor variação de latência, se mostrando mais determinístico com a variação do *Bush-loop*, e mantendo a ideia de execução em tempo real, uma vez que a latência é menor que o período com que os dados são publicados nos tópicos.

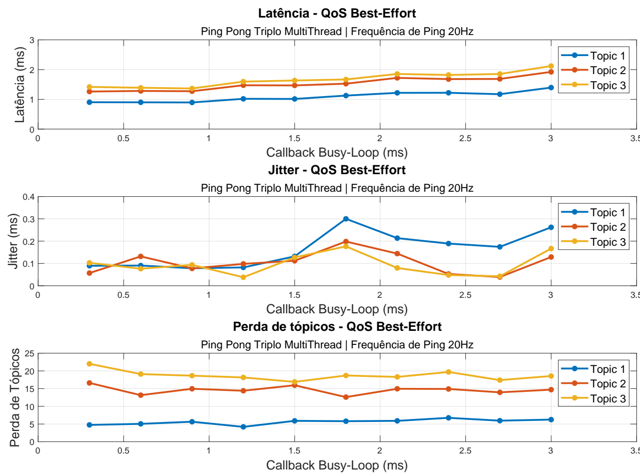

Figura 10. Resultados ensaio Ping-Pong triplo com executor *MultiThread* para qualidade de serviço *best-effort*.

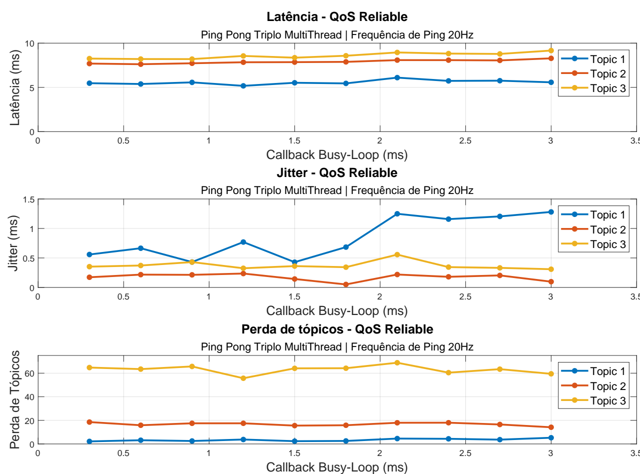

Figura 11. Resultados ensaio Ping-Pong triplo com executor *MultiThread* para qualidade de serviço *reliable*.

4.4 Comportamento com diferentes frequências

Para analisar o comportamento com diferentes frequências, foi realizado o ensaio Ping-Pong utilizando três tópicos, o executor *SingleThread*, o *Busy-loop* de 0.3 ms e a frequência de publicação dos *publishers* no nó de Ping variando entre 10 e 100 Hz com um passo de 10 Hz, com o intuito de avaliar o desempenho do sistema. A primeira situação utilizando a qualidade de serviço *best-effort* tem seus resultados expostos na Figura 12. Nela, as medidas de latência se mantêm estáveis até os 60 Hz e os de perda de tópicos até os 70 Hz, levando após isso a grandes aumentos de ambas. Esse aumento da latência acima de 70 Hz faz com que o sistema deixe de operar em tempo real, pois o período que as mensagens chegam é de aproximadamente 14 ms, valor que é extrapolado para a operação nas frequências acima de 80 Hz. Para essa QoS, os tópicos conseguem ser transmitidos com velocidade próxima à nominal do protocolo de comunicação utilizado, uma vez que não há dados de *feedback* trafegando pelo serial, o que também influência para que a comunicação suporte uma maior frequência.

Utilizando da QoS *reliable*, que devido à confirmação de envio das mensagens necessita de mais banda e mais tempo para se comunicar, para frequências acima de 30 Hz a perda de tópicos começa a aumentar significativamente, alcançando mais de 85% de perda de dados e inviabilizando a comunicação. Com a redução do período entre o envio das mensagens, a latência também passou à aumentar, ultrapassando já em 50 Hz o limite para que a comunicação seja em tempo real. Com tal variação, vê-se também que com a mudança da frequência de *Ping* para valores críticos o sistema deixa de ser determinístico, de tempo real e chega a inviabilização da comunicação. Dessa forma, além do tempo de processamento da *callback*, a comunicação entre ROS 2 e MicroROS também é sensível à variação do período com que as mensagens são enviadas pelos tópicos.

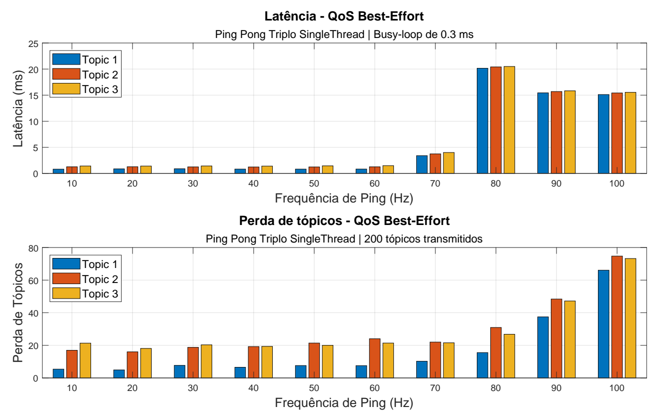

Figura 12. Resultados ensaio de variação de frequência para qualidade de serviço *best-effort*.

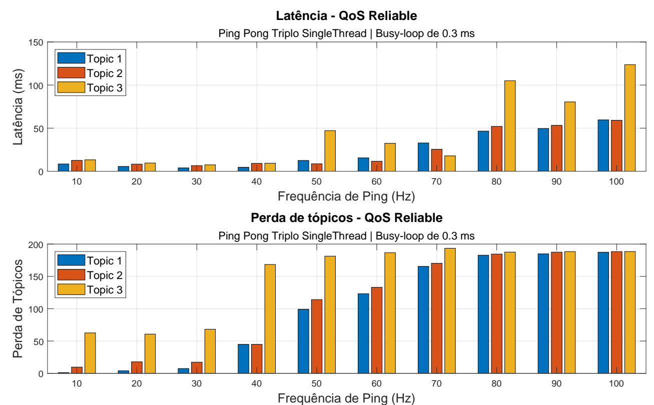

Figura 13. Resultados ensaio de variação de frequência para qualidade de serviço *reliable*.

5. CONCLUSÃO

A implementação da comunicação do ROS 2 com o MicroROS se mostra como uma boa arquitetura para sistemas distribuídos que interligam computadores e sistemas embarcados, mantendo o mesmo protocolo e topologia. Como visto, é de suma importância que essa comunicação seja confiável para garantir um funcionamento seguro dos robôs que venham a utilizá-la. Para isso, além da análise de alguns parâmetros, como latência e perda de pacotes, para garantir determinismo e operação em tempo real, podem ser aplicados perfis específicos de qualidade de serviço para garantir a performance desejada.

Realizando a análise de desempenho de tal comunicação, comparando a execução em cenários, executores e QoS diferentes, pôde-se traçar um perfil de operação para o MicroROS, ajudando posteriormente a identificar possíveis problemas em uma aplicação real na robótica. O *framework* embarcado tem grande sensibilidade para a alteração da frequência com que os dados são enviados à ele, apresentando um bom desempenho para frequências menores que 60 Hz utilizando qualidade de serviço *best-effort* e para QoS *reliable*, frequências menores que 30 Hz. Além disso, para execução de apenas um tópico no MicroROS, houve uma perda de tópicos pequena, de aproximadamente 0.5%, porém sempre presente. Tal perda cresceu com a presença de mais tópicos e deve ser levada em consideração durante sua aplicação.

Quanto ao carregamento do sistema, com o aumento do *Bush-loop*, a latência e a perda de tópicos para cada tópico mostrou pouca variação, provando que o sistema tem características determinísticas, apresentando tempo de resposta próximo, mesmo com alterações no tempo de processamento de cada *callback*. Com um ponto de operação de período de envio de dados em que o MicroROS tem um bom comportamento, o intervalo de tempo entre envio e recebimento foi menor que tal período de *Ping*, mostrando que a execução se mantém em tempo real. Todavia, na existência de vários tópicos comunicando, houve uma perda de dados considerável, podendo vir a gerar problemas para o funcionamento de um sistema real.

Contudo, é relembrado que as configurações de *hardware* do microcontrolador, como *clock*, número de *cores* e *threads* podem alterar o comportamento dos ensaios realizados. Assim como também a velocidade e a topologia da comunicação serial utilizada. Além de que, como a análise foi feita com foco em sistemas de tempo real, o fato do computador utilizado possuir um sistema que não é de tempo real, pode ser um agravante para seu resultado.

A análise de desempenho entre o ROS 2 e o MicroROS em um ambiente controlado e dedicado é apenas o primeiro passo para traçar o perfil de funcionamento, conseguindo mapear os pontos fortes e fracos desse *framework*. Dessa forma, abre-se novas portas para testes em robôs reais, analisando latência e perda de pacotes para sistemas reais, onde além do MicroROS, o sistema embarcado ainda realize leitura de sensores, processamento de dados e envio de comandos para a plataforma robótica. Além disso, novas pesquisas podem permitir um estudo mais aprofundado do sistema, principalmente focando em testes utilizando diferentes topologias e *hardwares*, possibilitando o mapeamento do verdadeiro obstáculo no desempenho do *framework* e permitindo assim, além de possibilitar a melhor escolha de arquitetura à ser utilizada, sanar os problemas detectados nesse trabalho e em futuros, aumentando a aplicabilidade do MicroROS.

REFERÊNCIAS

Adomat, J., Furunäs, J., Lindh, L., and Stärner, J. (1996). Real-time kernel in hardware rtu: a step towards deterministic and high-performance real-time systems. *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, 164–168.

de Almeida, W.G., de Almeida Monte-Mor, J., dos Santos, R.F., da Silveira, E.E.P., Izidoro, S.C., de Brito, T.G., Batista, N.C., and Vitor, G.B. (2019). Conception of an electric vehicle's robotic platform developed for applications on cts. In Y. Iano, R. Arthur, O. Saotome, V. Vieira Estrela, and H.J. Loschi (eds.), *Proceedings of the 3rd Brazilian Technology Symposium*, 123–129. Springer International Publishing, Cham.

FreeRTOS (2021). Real-time operating system for microcontrollers. URL `www.freertos.org`.

Knapp, E. (2019). Quality of Service Policies for ROS2 Communications. In *ROScon 2019*.

Laffey, T.J., Cox, P.A., Schmidt, J.L., Kao, S.M., and Readk, J.Y. (1988). Real-time knowledge-based systems. *AI magazine*, 9(1), 27–27.

Lange, R. (2018). Callback-group-level Executor for ROS 2. In *ROSCon 2018*. Madrid, Espanha.

Lange, R. (2021). Examples rclcpp cbg executor. URL `github.com/ros2/examples/tree/master/rclcpp/executors/cbg_executor`.

Linux Foundation (2021). Zephyr project. URL `www.zephyrproject.org`.

MicroROS (2022a). Execution management. URL `https://micro.ros.org/docs/concepts/client_library/execution_management/`.

MicroROS (2022b). Features and architecture. URL `micro.ros.org/docs/overview/features/`.

MicroROS (2022c). Micro xrce-dds. URL `micro.ros.org/docs/concepts/middleware/Micro_XRCE-DDS/`.

Open Robotics (2021a). About quality of service settings. URL `docs.ros.org/en/foxy/Concepts/About-Quality-of-Service-Settings.html`.

Open Robotics (2021b). ROS. URL `www.ros.org`.

Open Robotics (2022a). Executors. URL `https://docs.ros.org/en/foxy/Concepts/About-Executors.html`.

Open Robotics (2022b). ROS 2. URL `github.com/ros2`.

Rojas-Rueda, D., Nieuwenhuijsen, M.J., Khreis, H., and Frumkin, H. (2020). Autonomous vehicles and public health. *Annual Review of Public Health*, 41(1), 329–345. doi:10.1146/annurev-publhealth-040119-094035.

The Apache Software Foundation (2019). Apache nuttx. URL `nuttx.apache.org`.

Tournier, J.C. and Babau, J.P. (2004). Communication protocol evaluation for embedded systems. volume 2, 1006 – 1011 Vol.2. doi:10.1109/ICIT.2003.1290799.

Woodall, W. (2019). ROS on DDS. URL `design.ros2.org/articles/ros_on_dds.html`.

Yang, Y. and Azumi, T. (2020). Exploring real-time executor on ros 2. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, 1–8. doi:10.1109/ICESS49830.2020.9301530.