Mateus Amarante Araújo

Aplicação do ROS no Controle de Movimento de Robôs Equipados com Motores Dynamixel em Linux de Tempo Real

Brasil

7 de agosto de 2017

Mateus Amarante Araújo

Aplicação do ROS no Controle de Movimento de Robôs Equipados com Motores Dynamixel em Linux de Tempo Real

Projeto de Fim de Curso submetido ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia, requisito parcial para a obtenção do Título de Bacharel em Engenharia Mecatrônica

Universidade Federal de Uberlândia – UFU

Faculdade de Engenharia Mecânica – FEMEC

Graduação em Engenharia Mecatrônica

Orientador: Prof. Dr. Roberto de Souza Martins

Brasil

7 de agosto de 2017

Mateus Amarante Araújo

Aplicação do ROS no Controle de Movimento de Robôs Equipados com Motores Dynamixel em Linux de Tempo Real

Projeto de Fim de Curso submetido ao Curso de Engenharia Mecatrônica da Universidade Federal de Uberlândia, requisito parcial para a obtenção do Título de Bacharel em Engenharia Mecatrônica

---

Prof. Dr. Roberto de Souza Martins
Orientador

Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares
Convidado 1

Msc. Lucas Antônio Oliveira Rodrigues
Convidado 2

Brasil
7 de agosto de 2017

Agradecimentos

Primeiramente, agradeço à Equipe de Desenvolvimento em Robótica Móvel (EDROM), que me despertou o interesse pela robótica e computação, e proporcionou experiências essenciais para a minha formação pessoal e profissional.

À Universidade Federal de Uberlândia, que proporciou estrutura adequada e ambiente agradável no preparo para me tornar engenheiro mecatrônico.

Ao professor Rogério Sales Gonçalves, pela confiança no meu trabalho e pelo exemplo de comprometimento como tutor da equipe.

Ao professor orientador Roberto Souza Martins, que me deu liberdade para desenvolver minhas ideias e apoio decisivo na construção do texto.

Aos amigos da faculdade, em especial aos da 15ª turma de engenharia mecatrônica e aos que passaram pela EDROM, com quem aprendi e ri bastante.

Aos familiares e amigos extra-classe, que mesmo sem conhecer de perto o meu trabalho, o reconhecem e confiam em mim.

Enfim, agradeço aos meus pais, Antônio e Cleide, à minha namorada, Camilla, e ao meu falecido irmão, Filipe, pelo apoio incondicional. Vocês me ajudaram em todos os momentos, principalmente mais difíceis, me dando força e incentivo para poder acreditar nos meus sonhos e superar os desafios encontrados.

Resumo

A EDROM é um grupo de pesquisa da Universidade Federal de Uberlandia que participa de diversas competições de robótica ao redor do mundo, já tendo ganhado diversos títulos. Apesar do sucesso, a equipe enfrenta dificuldades em realizar o controle de movimento dos seus robôs humanoides. Uma razão para isso é o fato das ferramentas computacionais utilizadas carecerem de recursos para implementar estratégias alternativas de controle. Para superar as limitações encontradas, este trabalho propõe uma solução genérica baseada no ROS, especialmente sua ferramenta chamada *ros_control*, para controlar o movimento não somente dos robôs da EDROM, mas de qualquer robô equipado com motores *Dynamixel*. O novo sistema se destaca por ser modular, flexível e oferecer ferramentas que podem ser facilmente integradas com projetos de terceiros e utilizado em sistemas operacionais de tempo real de forma segura. O sistema é testado com sucesso em robô humanoide de 8GDL, sendo controlado por vários controladores genéricos e integrado com outros pacotes do ROS, como o *rviz* e a *tf*, usados para estimar o estado do robô e visualizá-lo em ambiente tridimensional. O novo sistema também é testado em Linux de tempo real, apresentando desempenho de controle de posição significamente melhor quando comparado ao sistema antigo em Linux de propósito geral.

Palavras-chave: *ros_control, ROS, RTOS, Linux de tempo real, Dynamixel, controle de movimento, robótica, rviz, tf, EDROM*

Abstract

EDROM is a research group that participates in several robotics competitions around the world, having already won several awards. Despite its success, the team has been facing difficulties in performing the motion control of humanoid robots. One reason for this is the fact the computational tools used lack resources to implement alternative control strategies. To overcome these limitations found, this paper proposes a generic solution based on ROS, especially its stack called *ros_control*, to control the motion not only of the EDROM's robots, but any robot equipped with Dynamixel servos. The new system stands out for being modular, flexible and offering tools that can be easily integrated with third-party tools and safely used in real-time operating systems. It is successfully tested with an 8DOF humanoid robot being controlled by several generic controllers and integrated with other ROS packages, especially *rviz* and *tf*, used to estimate the state of the robot and show it in a 3D environment. The new system is also tested in a real-time Linux, presenting significantly better performance in position control when compared with the old solution running in a general-purpose Linux.

Keywords: *ros_control, ROS, RTOS, real-time Linux, Dynamixel, motion control, robotics, rviz, tf, EDROM*

Lista de ilustrações

Figura 1 – Diagrama com os componentes básicos de um sistema robótico. 19
Figura 2 – Estrutura genérica de controle no espaço das juntas 20
Figura 3 – Controlador de posição dos motores 21
Figura 4 – Arquitetura da biblioteca de comunicação com os motores do sistema atual da EDROM 22
Figura 5 – Robôs humanoides integrados com ROS que utilizam motores *Dynamixel* 25
Figura 6 – Possíveis transições entre os estados de execução de um processo 27
Figura 7 – Exemplo de estrutura de ROS *nodes*, *topics* e *services* 30
Figura 8 – Exemplo de modelo URDF 31
Figura 9 – Aplicação com robô PR-2 visualizado no *rviz* 32
Figura 10 – Exemplos de robôs que utilizam o *ros_control* 33
Figura 11 – Arquitetura do *ros_control* aplicado a robô manipulador móvel. 35
Figura 12 – Estrutura de laço de controle para o *ros_control* 36
Figura 13 – Arquitetura de controle dos motores *Dynamixel* com *DxlRobotHW* . . 39
Figura 14 – Estrutura básica do *ControlLoop* 41
Figura 15 – Distribuição temporal de cada iteração do *ControlLoop* 42
Figura 16 – Exemplo de situação de acionamento do método *OnMiss* do *ControlLoop* com comportamento padrão. 42
Figura 17 – Arquitetura de *hardware* do ambiente de testes 46
Figura 18 – Arquitetura do teste de integração com o ROS 48
Figura 19 – Resposta a comando de posição de motor AX-12 no *rqt_plot* 49
Figura 20 – Resposta a comando de posição e velocidade de motor AX-12 no *rqt_plot* 49
Figura 21 – Estado da junta de motor AX-12 no *rqt_plot* em execução de trajetória 50
Figura 22 – Visualização no *rqt_plot* da posição de três juntas controladas de forma independente 51
Figura 23 – Diagrama do modelo URDF do robô de teste. 52
Figura 24 – Sequência de imagens capturadas durante teste de visualização do estado do robô no *rviz* 53
Figura 25 – Gráfico dos atrasos por iteração do teste de latência em SO Convencional 56
Figura 26 – Gráfico dos atrasos por iteração do teste de latência em RTOS 56
Figura 27 – Histograma de latências de até 150μs em SO Convencional 57
Figura 28 – Histograma de latências de até 150μs em RTOS 58
Figura 29 – Distribuição acumulada das latências de até 150μs em SO Convencional 58
Figura 30 – Distribuição acumulada das latências de até 150μs em RTOS 59
Figura 31 – Gráfico de posição da junta nos primeiros 5 segundos do experimento com o sistema de controle antigo 61

Figura 32 – Gráfico de posição da junta em momento de atraso significativo no experimento com o sistema de controle antigo 62
Figura 33 – Gráfico de posição e velocidade da junta em momento de oscilação indesejada no experimento com o *DxlRobotHW* em SO Convencional . 63
Figura 34 – Gráfico de posição e velocidade da junta em momento de maior atraso no experimento com o *DxlRobotHW* em RTOS 64
Figura 35 – Tela de configuração do modelo de preempção do *kernel* do Linux . . . 74

Lista de tabelas

Tabela 1 – Especificação de versão das ferramentas de *software* relevantes 46
Tabela 2 – Parâmetros do teste de tempo de resposta 55
Tabela 3 – Dados estatísticos do teste de latência 57

Lista de abreviaturas e siglas

API	*Application Program Interface*
DARPA	*Defense Advanced Research Projects Agency*
DCR	*DARPA Robotics Challange*
DOF	*Degree of Freedom*
EDROM	Equipe de Desenvolvimento em Robótica Móvel
FEMEC	Faculdade de Engenharia Mecânica
FIFO	*First In, First Out*
GDL	Grau de Liberdade
IMU	*Inertial Measurement System*
OROCOS	*Open RObot COntrol Software*
OSRF	*Open Source Robotics Foundation*
POSIX	*Portable Operating System Interface*
ROS	*Robot Operation System*
RTOS	*Real-Time Operating System(s)*
SDK	*Software Development Kit*
SO	*Sistema Operacional*
UFU	Universidade Federal de Uberlândia
URDF	*Unified Robot Description Format*

Sumário

1 INTRODUÇÃO 17

2 REVISÃO BIBLIOGRÁFICA 19
2.1 Controle de Movimento de Robôs 19
2.2 Soluções de *Software* 22
2.2.1 Implementação Atual da EDROM 22
2.2.2 Características de *Software* para Robótica 24
2.2.3 Ferramentas Existentes 24
2.3 Sistemas Operacionais de Tempo Real 26

3 ROBOT OPERATING SYSTEM (ROS) 29
3.1 Visão Geral do ROS 29
3.1.1 Conceitos Principais 29
3.1.2 Ferramentas Exploradas 31
3.2 ROS_CONTROL 33
3.2.1 Estrutura 34
3.2.2 Utilização 37

4 IMPLEMENTAÇÃO 39
4.1 DxlRobotHW 39
4.2 ControlLoop 41
4.3 posvel_controllers 43

5 TESTES E ANÁLISES 45
5.1 Configuração do Ambiente 45
5.1.1 *Hardware* 45
5.1.2 *Software* 46
5.2 Integração com o ROS 48
5.2.1 Funcionalidades Básicas 48
5.2.2 Visualização no *rviz* 51
5.3 Desempenho em RTOS 54
5.3.1 Preparação do Ambiente 54
5.3.2 Análise de Latência 55
5.3.3 Impacto sobre Controle de Posição das Juntas 59

6 CONSIDERAÇÕES FINAIS 65

REFERÊNCIAS 67

APÊNDICES 71

APÊNDICE A – CONFIGURAÇÃO DO SISTEMA OPERACIONAL DE TEMPO REAL 73
A.1 Preparação do Sistema Operacional 73
A.2 Configuração de Aplicações de Tempo Real 75

APÊNDICE B – CÓDIGO FONTE 77
B.1 DxIRobotHW 77
B.2 Control Loop 84
B.3 posvel_controllers 98
B.4 Programa de Registro de Latências 103

APÊNDICE C – MODELO URDF DO ROBÔ DE TESTE 105

1 Introdução

A EDROM, Equipe de Desenvolvimento em Robótica Móvel da Universidade Federal de Uberlândia (UFU), é um projeto de extensão criado para fomentar a pesquisa e o ensino em robótica, onde alunos desenvolvem robôs para cumprir desafios em competições nacionais e internacionais. Uma das modalidades mais desafiadoras disputadas é a de Futebol de Robôs Humanoides, em que robôs bípedes disputam partidas de futebol autonomamente. Dentre as diversas atividades que os robôs humanoides devem realizar, o movimento de andar é uma das mais críticas e complexas. É essencial que os robôs apresentem um bom desempenho de movimentação para executar suas tarefas. Entretanto, a equipe enfrenta dificuldades para fazer o robô andar de forma precisa, suave e robusta. Atualmente, consegue-se realizar movimentos lentos, instáveis e com baixa repetibilidade, sendo que o robô, frequentemente, sofre quedas e danos aos motores por torque excessivo.

É necessário implementar técnicas de controle de movimento mais adequadas para solucionar estes problemas. Entretanto, observou-se que as ferramentas computacionais de baixo nível utilizadas para controlar os atuadores possuem limitações. Por exemplo, elas não fazem uso de dados dos sensores dos motores durante a execução do movimento, o que seria essencial para implementar técnicas de controle com realimentação, estimar o estado do robô e analisar o comportamento das juntas. Além disso, sua estrutura de *software* não favorece a reutilização de código, a adição de novos recursos e a integração com ferramentas de terceiros.

Desse modo, o objetivo deste trabalho consiste em implementar uma solução alternativa que visa superar estas limitações e agregar novas funcionalidades. Em especial, explora-se a utilização do *Robot Operating System* (ROS), principalmente da sua ferramenta *ros_control*, para controlar e monitorar o movimento de qualquer robô que utilize os atuadores *Dynamixel*, utilizados pela EDROM. Dentre as novas funcionalidades, destacam-se o ganho em modularidade, o suporte a sistemas operacionais de tempo real (RTOS) e a integração com ferramentas de visualização do ROS.

Assim, este trabalho inicia com uma breve revisão bibliográfica sobre trabalhos e conceitos relacionados, com destaque à apresentação da estrutura do ROS e do *ros_control*. Posteriormente, apresenta-se detalhes de implementação e testes realizados: exemplifica-se o uso do novo sistema de forma integrada com ferramentas do ROS em tronco de robô humanoide de 8GDL e realizam-se testes que avaliam o desempenho de comunicação e controle de posição de um motor *Dynamixel* em sistema operacional de tempo real. Enfim, expõe-se comentários sobre a experiência obtida e os resultados alcançados, além de sugestões de trabalhos futuros.

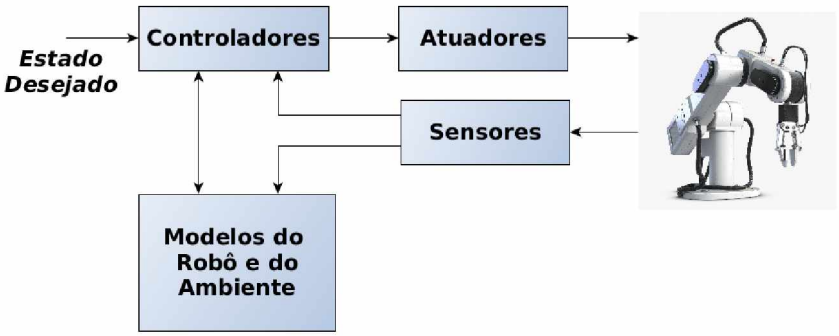
2 Revisão Bibliográfica

Visto que o presente trabalho visa apresentar uma solução de *software* genérica para facilitar o controle de movimento de robôs, apresenta-se uma breve contextualização sobre as características de sistemas de controle em robótica, alinhado ao desafio de desenvolver *software* para esta aplicação. Discute-se sobre alguns trabalhos relacionados, incluindo a solução atual da EDROM, que é comparada com a alternativa proposta. Finalmente, uma vez que uma das funcionalidades testadas é o suporte a sistemas operacionais de tempo real, apresenta-se uma breve explicação sobre eles.

2.1 Controle de Movimento de Robôs

Sistemas robóticos se caracterizam por serem muito complexos, envolvendo diferentes áreas do conhecimento, como mecânica, eletrônica, controle e computação (SICILIANO et al., 2009). Funcionalmente, sistemas robóticos podem ser resumidos como na Figura 1.

Figura 1 – Diagrama com os componentes básicos de um sistema robótico.



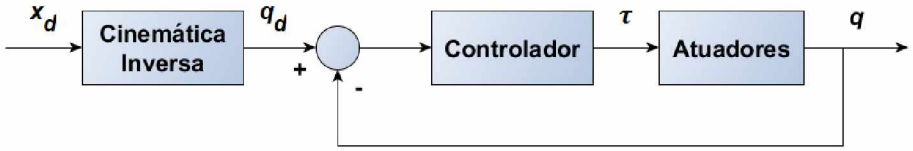
Fonte: Adaptado de Siciliano et al. (2009, p. 3)

Essencialmente, robôs possuem uma estrutura mecânica para possibilitar sua interação com o ambiente. Em geral, sistemas robóticos possuem elementos de locomoção, como rodas ou pernas mecânicas, e/ou de manipulação, como pinças e braços. O que permite o robô executar as ações, provocando o movimento das suas partes mecânicas, é o sistema de atuação. Este sistema inclui elementos eletromecânicos, como motores, transmissões e cilindros pneumáticos. A forma com que sistemas robóticos sentem o meio ocorre através do sistema de sensoriamento. Robôs também possuem sensores que captam informações de si mesmos, como transdutores de posição e unidade de medição inercial (do inglês Inertial Measurement Unit - IMU), e sensores que captam informações externas, como câmeras e lasers. Enfim, o sistema de controle faz a ligação entre as capacidades de

percepção e ação dos robôs. Este sistema é responsável por gerar os comandos de atuação baseados nas tarefas planejadas e das restrições do robô e do ambiente. Na execução destes comandos, com informações realimentados pelo sistema de percepção e modelos conhecidos do sistema, o módulo de controle busca alinhar o planejamento à realização das tarefas (SICILIANO et al., 2009, cap. 1).

Neste sentido, o "Controle de Movimento"(do inglês *Motion Control*) busca garantir que o robô execute os movimentos que o direcione aos estados desejados. Existem diversas técnicas de controle de movimento, variando de acordo com as características de cada tipo de robô, *hardware*, aplicação, etc. Entretanto, é comum que o estado desejado do robô possa ser definido por trajetórias de posição, velocidade e/ou aceleração das juntas ao longo do tempo. Com esta configuração, define-se os denominados controladores do espaço das juntas, cuja estrutura mais comum é representada na Figura 2.

Figura 2 – Estrutura genérica de controle no espaço das juntas



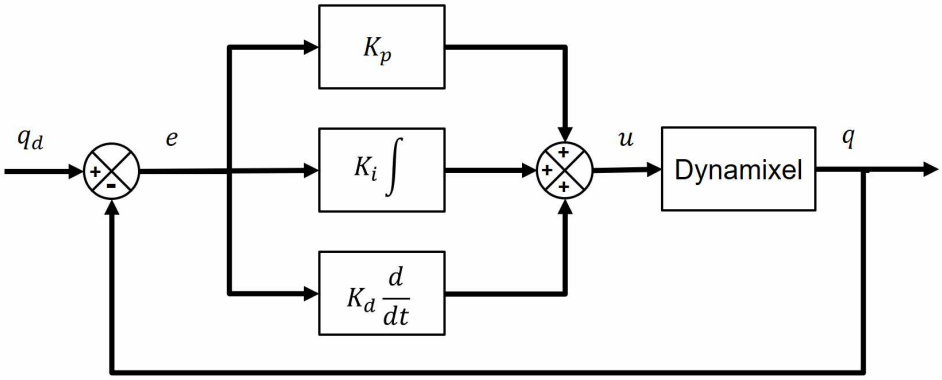
Fonte: Adaptado de Chung, Fu e Hsu (2008, p. 136)

Nestes controladores, o movimento desejado do robô é representado por trajetórias em coordenadas cartesianas e pela orientação de partes do robô, como do elemento terminal de um braço robótico ou dos pés e centro de massa de um robô humanoide. Estas coordenadas, pelo processo denominado cinemática inversa, são convertidas para as coordenadas de posição, velocidade e/ou aceleração das juntas, constituindo a referência para o controlador. Este, com base no sinal de entrada, modelos conhecidos e dados de sensores, como encoders e sensores de corrente, compara o estado desejado e o realizado pelas juntas e define o torque adequado a ser aplicado aos motores (CHUNG; FU; HSU, 2008).

Em robôs equipados com os atuadores *Dynamixel*, desenvolvidos pela emprea ROBOTIS, cada motor possui circuitos internos que realizam o controle de posição e velocidade de cada junta. A estrutura do controlador de posição implementado é apresentada na Figura 3.

A técnica de controle utilizada é chamada de PID (proporcional-integral-derivativo). Nela, o sinal de controle é composto pela soma de três componentes: o erro de posição, seu somatório ao longo do tempo (integral) e sua taxa de variação (derivada), sendo cada porção multiplicada por uma constante ajustável, K_p, K_i e K_d respectivamente (ROBOTIS, 2017a). Assim, a forma mais simples de controlar robôs equipados com estes

Figura 3 – Controlador de posição dos motores



Fonte: Adaptado de ROBOTIS (2017a)

atuadores é enviar os comandos de posição desejados para cada motor e deixar que cada um, independentemente, se encarregue de controlar suas posições. Esta configuração, em que cada junta é controlada de forma desacoplada do restante do sistema, caracteriza o chamado Controle Independente de Junta, descrito por Chung, Fu e Hsu (2008, p. 139-141).

Siciliano et al. (2009, cap. 8) e Spong, Hutchinson e Vidyasagar (2006, cap. 7–8) apresentam diversas outras variações de controladores no domínio das juntas, originalmente aplicado a robôs manipuladores, mas expansível para outras aplicações. Por exemplo, Kajita e Espiau (2008) e Lentin (2015a, cap. 5) afirmam que, ao menos para realizar movimentos discretos, no nível mais baixo, robôs humanoides apresentam controladores no domínio das juntas e Schwarz e Behnke (2013) conseguem ganho significativo de estabilidade de robô humanoide apenas com o aprimoramento do controle nesta etapa.

Apesar disso, esta configuração muitas vezes não é suficiente, sendo necessário aplicar outros controladores. Em uma mesma aplicação, vários controladores podem coexistir, seja para manipular partes do robô de forma independente, controlar variáveis diferentes ou atender propósitos dinâmicos. Isso pode ser observado em Lentin (2015a, p. 284), que apresenta a estrutura de controle do robô humanoide "P2", da empresa Honda. Este possui controladores para a inclinação do robô, a posição e orientação do pé, do ponto de momento zero (do inglês, *Zero Moment Point*, ZMP), além do controlador de posição das juntas.

Portanto, o controle de movimento de robôs pode ser uma tarefa complexa e dinâmica, que se soma a outros inúmeros desafios não mencionados, como modelagem de sistemas, projeto mecânico, processamento de dados de imagens e sensores, inteligência artificial, integração de hardware, etc. Com isso, os projetos de *software* para robôs também são complexos, sendo alvo de investigação no meio acadêmico e empresarial.

2.2 Soluções de *Software*

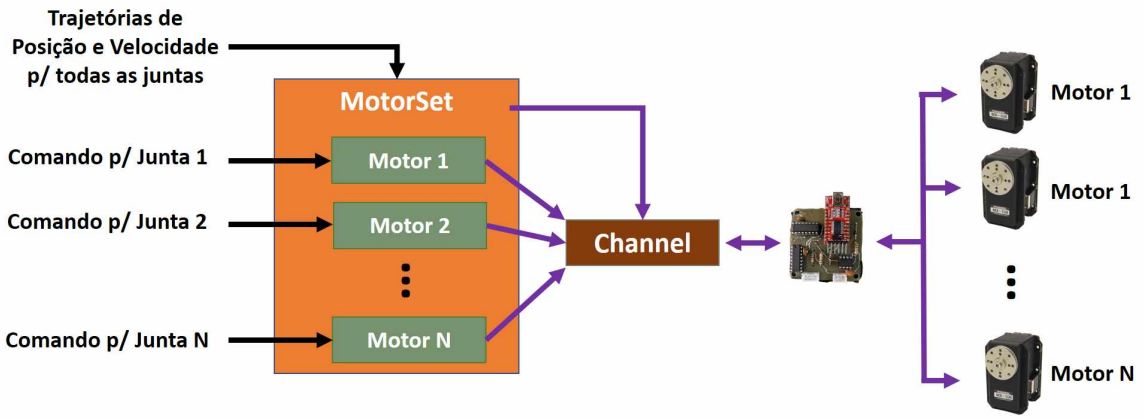
2.2.1 Implementação Atual da EDROM

Atualmente, para controlar o movimento dos robôs, a EDROM utiliza uma biblioteca desenvolvida por membros da equipe, incluindo o autor deste trabalho[1]. Ela foi projetada para ser minimamente funcional, oferecendo métodos básicos de envio de comandos de posição e velocidade para um ou mais motores *Dynamixel* com base em trajetórias especificadas. Para isso, ela se estrutura em três classes principais.

- **Channel:** representa um canal de comunicação com os motores *Dynamixel*, oferecendo métodos básicos de envio de comandos e leitura de dados dos sensores dos motores. Geralmente existe apenas uma instância do *Channel* para comunicar com todos os motores;
- **Motor:** reúne informações de cada motor, como especificação de modelo e limites de posição, e oferece métodos intuitivos de interação com um motor individualmente, como de envio de comandos de posição e velocidade;
- **MotorSet:** controla o envio de trajetórias de posição e velocidade a um conjunto de motores. Reúne instâncias da classe *Motor* para lidar com as particularidades de cada junta e se comunica diretamente com o *Channel* para, a cada ponto da trajetória, enviar as referências de todas as juntas em um mesmo comando.

A Figura 4 apresenta a a arquitetura da solução.

Figura 4 – Arquitetura da biblioteca de comunicação com os motores do sistema atual da EDROM



Fonte: Produzido pelo autor.

[1] Até a presente data não há trabalhos publicados a respeito desta solução. Portanto, as informações contidas nesta sessão tem base no conhecimento e experiência do autor em relação a biblioteca

Fisicamente, a comunicação com os motores parte do envio de pacotes de dados por uma porta USB do computador, passando por um circuito de conversão para os padrões TTL ou RS-485 que conecta os motores em rede. No mesmo barramento, trafegam pacotes com os comandos e com dados de resposta dos motores.

Para executar trajetórias de posição e velocidade das juntas do robô, o *MotorSet* espera a sequência de "poses"que representam as trajetórias de referência como parâmetro. Como descreve a Equação 2.1, cada pose i, define a posição e a velocidade desejadas de cada junta j ($q_{d_j}[i]$ e $\dot{q}_{d_j}[i]$) para cada instante, acompanhado do tempo de espera desejado $t_{prox}[i]$ entre o envio da pose corrente e a posterior.

$$Pose[i] = \begin{bmatrix} q_{d_1} & \dot{q}_{d_1} & q_{d_2} & \dot{q}_{d_2} & \cdots & q_{d_N} & \dot{q}_{d_N} & t_{prox} \end{bmatrix}[i] \tag{2.1}$$

Assim, o *MotorSet* possui um *loop* interno que, a cada iteração, envia os comandos pelo *Channel* e espera até o momento desejado de envio da próxima pose ($t_d[i+1]$) com base no momento de início da iteração corrente $t[i]$, como apresentado na Equação 2.2.

$$t_d[i+1] = t[i] + t_{prox}[i]; \tag{2.2}$$

Esta ferramenta, apesar de funcional, apresenta muitas limitações. Primeiramente, sua estrutura é bastante fechada, ou seja, não prevê a implementação de novas funcionalidades e algoritmos. Por exemplo, ela não faz uso da leitura de dados dos motores nem possui recursos explícitos para implementar controladores entre o envio de cada pose. Para incrementar estas funcionalidades no código atual, seria necessário fazer mudanças no *MotorSet* internamente, em regiões sensíveis a alterações.

Esta estrutura também não possui suporte para controlar mais de uma parte do robô de forma independente. É necessário definir todas posições e velocidades de todas as juntas em uma mesma pose. Seria possível enviar comandos individualmente a cada motor ou definir mais de um *MotorSet*, separando os motores em conjuntos, porém, haveria o uso concorrente do *Channel*, o que degradaria o desempenho de controle de todas as partes.

Além disso, os testes desenvolvidos ao longo deste trabalho revelam problemas na estratégia de envio de comandos de trajetória realizada pelo *MotorSet*. Será demonstrado que este algoritmo falha significativamente quando é executado em paralelo com outros processos que sobrecarregam o uso do processador e memória do computador.

Em resposta, investigou-se sobre soluções alternativas que pudessem sanar estes problemas e abrir espaço para outras funcionalidades. Nas próximas sessões serão apresentadas as características almejadas para o novo sistema e os trabalhos relacionados mais relevantes.

2.2.2 Características de *Software* para Robótica

Não existe uma única arquitetura de *software* que atenda todas as aplicações de robótica, sendo que, praticamente, cada projeto possui uma solução própria. Apesar disso, muitos esforços foram feitos na tentativa de tornar o desenvolvimento de robôs uma tarefa mais simples, robusta e produtiva, não só no controle de movimento, mas em todos os aspectos envolvidos no desenvolvimento em robótica (SICILIANO et al., 2009, Cap. 6).

Atualmente, existem diversas ferramentas computacionais com este intuito. Considerando as ferramentas de propósito geral deste ramo, Iñigo-Blasco et al. (2012), Dutta (2012) e Elkady e Sobh (2012) apresentam comparativos entre mais de 15 soluções, avaliando diversos aspectos tidos como desejáveis em aplicações para robótica, dentre elas:

- **Modularidade**: arquitetura que permita a subdivisão da aplicação em múltiplos módulos simples e independentes. Esta configuração contribui em minimizar a comunicação entre os módulos, reduzindo a complexidade e aumentando a robustez do sistema;
- **Escalabilidade**: possibilidade de utilizar a ferramenta em diversos tipos de aplicações e robôs;
- **Reusabilidade**: possuir estrutura que facilite a utilização de código de terceiros. Também é desejável que haja o incentivo ao compartilhamento de código. Neste sentido, muitas destas ferramentas são gratuitas e de código aberto;
- **Ferramentas**: disponibilidade de recursos prontos, como algoritmos pertinentes, simuladores, ferramentas de depuração e visualização;
- **Suporte a sistemas de tempo real**: ser capaz de rodar em sistemas operacionais de tempo real. Esta capacidade é desejável para implementar controladores de baixo nível que interagem com *hardware* em alta frequência, em que o cumprimento dos prazos de envio e recebimento de dados pode ser decisivo para um bom funcionamento do sistema.

Neste contexto, foi realizada uma pesquisa sobre as principais ferramentas que poderiam ser utilizadas para implementar um sistema genérico de controle de movimento no domínio das juntas (com foco em robôs que utilizam motores *Dynamixel*) e que agregue características vislumbradas em sistemas de *software* para robôs ainda não contempladas na solução atual da EDROM.

2.2.3 Ferramentas Existentes

Apesar do grande número de soluções existentes, o *Robot Operating System* (ROS) se destaca como um dos mais populares (LAGES; IORIS; SANTINI, 2014). Diferente do que

o nome sugere, o ROS não é um sistema operacional, mas sim uma coleção de ferramentas, bibliotecas e convenções que visam simplificar a tarefa de desenvolver comportamentos complexos e robustos a uma vasta variedade de robôs (QUIGLEY et al., 2009).

Existem diversas soluções para controle de movimento de robôs integrado ao ROS, inclusive que utilizam os atuadores *Dynamixel*. Por exemplo, a própria empresa que produz estes motores, a ROBOTIS, recentemente disponibilizou o ROBOTIS-Framework, uma biblioteca que gerencia o envio e recebimento de dados de sensores e atuadores comercializados pela empresa, sendo utilizado para controlar o robô humanoide THORMANG3, apresentado na Figura 5a (ROBOTIS, 2017c; ROBOTIS, 2017b). Outro projeto relacionado, é o NimbroOP-ROS, Figura 5b, projeto de *software* completo de robô de destaque da competição de futebol de robôs humanoide, desenvolvido na Universidade de Bonn, da Alemanha. Allgeuer et al. (2013) descreve com detalhes a sua arquitetura.

Figura 5 – Robôs humanoides integrados com ROS que utilizam motores *Dynamixel*



(a) THORMANG3



(b) Nimbro-OP

Fonte: a) ROBOTIS (2017c); b) AIS (2017).

Visto o sucesso e a semelhança com o contexto da EDROM destes projetos, eles se apresentam como boas alternativas para investigação. Entretanto, ao analisar o código fonte destes projetos (ROBOTIS, 2017c; AIS, 2017), percebeu-se que ambos não oferecem muita flexibilidade nem recursos explícitos para implementar modificações e novas funcionalidades, como aplicar variações de *hardware* e controladores. Assim, o trabalho de adaptar esta solução a outros projetos pode ser elevado ou restrito. Além disso, estas soluções não apresentam, explicitamente, suporte a sistemas operacionais de tempo real.

Na realidade, uma das limitações do ROS é que ele ainda não oferece, oficialmente, ferramentas para aplicações de tempo real[2]. Com esta limitação, também é válido citar o projeto OROCOS[3] (*Open Robot COntrol Software*), que tem como foco, oferecer ferramentas de implementação de controle de tempo real para robôs e máquinas (OSRF, 2017a). Visto esta especificidade, o OROCOS é comumente utilizado em conjunto com o ROS, como fazem Lages, Ioris e Santini (2014) e Peekema, Renjewski e Hurst (2013)

Ao final, optou-se por utilizar o pacote ROS chamado *ros_control*, que se destacou por ser uma ferramenta simples, flexível e que oferece recursos que possibilitam seu uso em sistemas operacionais de tempo real[4]. Com isso, o próximo capítulo é dedicado a explicar detalhes destas ferramentas.

2.3 Sistemas Operacionais de Tempo Real

Sistemas operacionais (SO) são programas que gerenciam o uso dos componentes de um computador, como memória, processadores e dispositivos de armazenamento, abstraindo detalhes de utilização de *hardware* aos programas do usuário. As estratégias de gerenciamento destes recursos, por sua vez, variam em diversos aspectos. Neste sentido, sistemas operacionais de tempo real (do inglês *Real-Time Operating Systems*, RTOS) especificam-se por terem o tempo como parâmetro chave (TANENBAUM; BOS, 2015).

O objetivo principal dos RTOS é garantir que a execução de tarefas críticas sejam realizadas de forma determinística, ou seja, que as restrições de prazo de conclusão das operações sejam cumpridas dentro de limites de tempo previsíveis. Entende-se como tarefas críticas, aplicações em que pequenos atrasos de execução podem invalidar a operação ou gerar impactos significativos. Por exemplo, se, enquanto um carro se move em uma linha de montagem, o robô soldador atua um pouco antes ou depois do momento correto, os automóveis podem ser arruinados (TANENBAUM; BOS, 2015); e um atraso no acionamento de um *airbag* pode causar a morte dos seus passageiros (NATIONAL INSTRUMENTS, 2011).

Neste trabalho, a aplicação desta funcionalidade busca aumentar confiabilidade no cumprimento dos prazos de envio e recebimento de dados na comunicação com os atuadores dos robôs, refletindo em maior segurança e precisão na execução dos movimentos. Nos robôs humanoides, esta melhoria pode, por exemplo, evitar quedas por desvios de posição causados por atrasos de comunicação.

Para entender como um RTOS funciona, é importante entender os estados básicos

[2] Está previsto para que o ROS 2.0, atualmente em desenvolvimento, ofereça ferramentas nativas de suporte a RTOS. <http://design.ros2.org/>

[3] <http://www.orocos.org/>

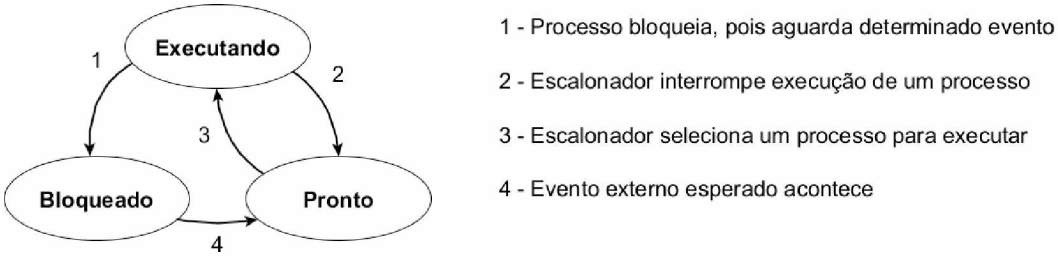
[4] Até a versão Kinect do ROS, o *ros_control* não está entre os pacotes padrão do ROS. Assim, oficialmente, o ROS não

em que um processo do SO pode assumir. Tanenbaum e Bos (2015, p. 92-93) apresentam que um processo, em determinado instante, encontra-se em um dentre três estados possíveis:

- **Executando:** o processo está realmente utilizando o processador;
- **Pronto:** o processo está pronto para usar o processador, mas está temporariamente parado para deixar que outro processo o utilize;
- **Bloqueado:** inapto a executar até que determinado evento externo ocorra.

A Figura 6 apresenta as possíveis transições entre estes estados.

Figura 6 – Possíveis transições entre os estados de execução de um processo



Fonte: Adaptado de Tanenbaum e Bos (2015, p. 93).

A transição 1 ocorre quando o SO percebe que o processo não pode executar mais, pois depende de um evento externo, como um sinal de outro processo, uma interrupção de *hardware* ou a expiração de um *timer*. As transições 2 e 3 são executadas pelo escalonador do SO, que decide qual processo deve executar em determinado momento e por quanto tempo, de modo a garantir o uso compartilhado do processador da melhor forma. Enfim, a transição 4 ocorre quando o evento o qual o processo bloqueado estava esperando acontece.

Neste contexto, o RTOS, para cumprir seu objetivo, adota estratégias de escalonamento que garantam que processos que executem tarefas críticas estejam em execução nos momentos desejados. Para isso, ao gerenciar a alocação de processos críticos, busca-se minimizar os tempos de resposta em todas as transições de estado e evitar a ocorrência das transições 2 e 3 com processos menos prioritários (SIRIO, 2017).

Neste contexto, investigou-se como o SO utilizado neste projeto, o Linux, se comporta em relação a estes aspectos. Verificou-se que, originalmente, ele é voltado ao uso genérico, tendo como objetivo oferecer uma experiência fluida ao usuário no uso de aplicações diversas, o que não implica na garantia do cumprimento de prazos. Entretanto, foram desenvolvidos diversos projetos independentes para agregar suporte a aplicações de tempo real a este SO. Basicamente, este suporte se dá por meio de modificações e/ou complementação do *kernel*[5] e do oferecimento de bibliotecas pertinentes (PUHAN, 2015).

[5] *Kernel* é o conjunto de programas essenciais do sistema operacional (PUHAN, 2015)

A solução mais simples consiste em aplicar um pacote de modificações, chamado PREEMPT_RT, ao *kernel* do Linux. Esta mudança possibilita o uso efetivo da biblioteca padrão POSIX[6] para definir regras de escalonamento de tempo real e atribuir prioridades a *threads* individualmente (PUHAN, 2015). *Threads* são fluxos de processamento de um processo, funcionando como processos internos, sendo que, na realidade, o conceito de estado do processo e de escalonamento são aplicados a nível de *threads* (TANENBAUM; BOS, 2015).

Basicamente, o RTLinux, como é chamado o Linux de tempo real, permite a configuração de, três diretrizes principais de escalonamento, definidas no código pelas seguintes constantes:

- **SCHED_OTHER:** modelo de escalonamento padrão do Linux, adequado para processos de propósito geral. Todas as *threads* com esta configuração possuem prioridade estática mínima (zero), mas o SO atribui prioridades dinâmicas entre elas para garantir o compartilhamento justo do processador;
- **SCHED_FIFO:** escalonador de tempo real que atribui prioridade estática configurável superior a zero. A *thread* configurada, quando se encontra no estado "pronto", imediatamente toma uso do processador caso as *threads* em execução e em espera sejam de prioridade menor. *Threads* do tipo SCHED_FIFO com mesma prioridade, quando em estado "pronto", são ordenados em fila sob o modelo FIFO (*"First In, First Out"*);
- **SCHED_RR:** escalonador de tempo real *Round-Robin*. Difere-se do escalonador FIFO apenas pelo fato de definir um tempo máximo de uso contínuo do processador. Quando este tempo excede, caso haja processos de mesma prioridade em espera, a *thread* deixa de executar e é alocada no final da fila. Esta alternativa evita que uma *thread* domine o uso do processador em detrimento de outras tarefas de tempo real de mesma prioridade (KERRISK, 2016).

No RT-Linux, cada *thread* pode assumir uma regra de escalonamento diferente. Assim, na prática, a forma de operação do Linux de tempo real é praticamente a mesma do convencional, pois as tarefas de propósito geral são gerenciados da mesma forma. A principal mudança é que se torna possível configurar as *threads* que requerem desempenho de tempo real individualmente.

[6] Do inglês, *Portable Operating System Interface*, é um conjunto de padrões especificados pela *IEEE Computer Society* para manter compatibilidade entre sistemas operacionais.

3 Robot Operating System (ROS)

Este capítulo apresenta os principais conceitos e funcionalidades do *Robot Operating System* e do *ros_control*, dando fundamentação teórica necessária para compreender os códigos, testes e análises desenvolvidos.

3.1 Visão Geral do ROS

Originalmente, o ROS foi projetado para superar desafios encontrados no desenvolvimento de robôs de serviço do projeto STAIR (*STanford Artificial Intelligence Robot*) e do programa *Personal Robots* (PR), desenvolvidos na Universidade de Standford e na incubadora Willow Garage em meados da década de 2000. Estes esforços, porém, resultaram em uma arquitetura muito mais genérica, expandindo-se a inúmeras outras aplicações. O ROS reúne características que, não somente potencializaram o desenvolvimento de soluções de *software* avançadas e robustas para robótica, como estimularam a formação de um ecossistema sólido de colaboração e compartilhamento de código (OSRF, 2017a).

Segundo Quigley et al. (2009), o *framework* foi projetado para ser modular e enxuto, limitando-se a oferecer mecanismos de troca de mensagens entre processos, encorajando o desenvolvimento de códigos específicos de *hardware* e algoritmos em bibliotecas independentes. O ROS se destaca por oferecer suporte a múltiplas linguagens, ser gratuito e de código aberto, sendo distribuído com licença de *software* permissiva que permite utilizá-lo em projetos comerciais. Com isso, hoje, o ROS é utilizado por dezenas de milhares de usuários ao redor do mundo em aplicações que variam desde projetos de mesa de "hobbistas"até grandes sistemas de automação industrial (OSRF, 2017a).

3.1.1 Conceitos Principais

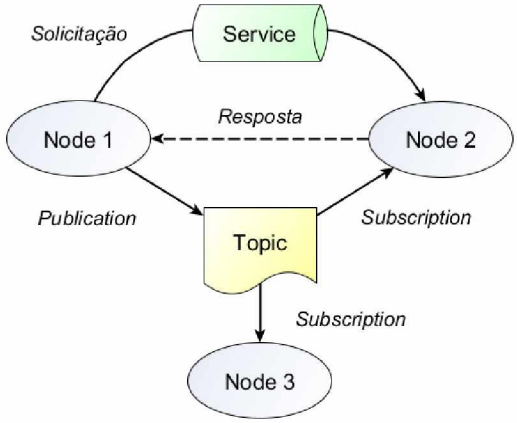
Essencialmente, o ROS define uma infraestrutura de comunicação sobre a qual suas ferramentas e as aplicações desenvolvidas pelos usuários se estruturam. As aplicações se configuram como uma rede de processos que trocam mensagens entre si, formando o chamado *Computational Graph*, que reúne diversos conceitos, como:

- ***Node*:** os nós são os processos em si. O ROS é projetado para ser modular em alta granularidade. Geralmente, sistemas robóticos são compostos por muitos nós;
- ***Parameter Server:*** o servidor de parâmetros é um repositório de informações onde podem ser armazenados dados estáticos, geralmente informações de configuração, dispostos aos *nodes* utilizarem em tempo de execução;

- ***Messages:*** nós se comunicam através da troca de mensagens. Uma mensagem é uma simples estrutura de dados que são arbitrariamente compostas por campos de tipos primitivos (números inteiros, ponto flutuante, caracteres, etc), vetores e outras estruturas aninhadas;
- ***Topics*:** as mensagens podem ser transportadas utilizando o padrão *publish-subscribe*. Um nó envia mensagens para outros *nodes* externos publicando-as em tópicos. Um nó interessado em receber determinado tipo de dado, se inscreve nos tópicos onde são publicados os dados que deseja (*subscriber*). Tópicos são identificados por um nome e um tipo de mensagem e, geralmente, *publishers* e *subscribers* de um tópico não têm conhecimento uns dos outros;
- ***Services:*** estabelece comunicação entre nós no modelo *request-reply* (solicitação-resposta). Um serviço é como uma chamada de procedimento remoto, em que o nó cliente faz uma requisição e espera uma mensagem de resposta do nó servidor. Um serviço é definido por um nome e um par de tipos de mensagens, uma de solicitação e a outra de resposta (OSRF, 2014).

A Figura 7 ilustra graficamente o funcionamento da troca de mensagens entre nós por meio de tópicos e serviços.

Figura 7 – Exemplo de estrutura de ROS *nodes*, *topics* e *services*



Fonte: Adaptado de OSRF (2014)

Neste exemplo, *Node* 1 publica mensagens em um *Topic*, enquanto *Nodes* 2 e 3, inscritos neste tópico, recebem as mensagens publicadas assincronamente. Além disso, *Node* 2 oferece um serviço, por onde *Node* 1 envia uma mensagem com uma solicitação e espera uma mensagem de resposta.s

Portanto, o desenvolvimento de uma aplicação em ROS consiste em definir uma estrutura com estes componentes e implementar os algoritmos nos *nodes* criados. Códigos desenvolvidos em ROS são organizados em *packages* (pacotes) que, além de conter

algoritmos e ferramentas, podem conter arquivos com definição de mensagens e dados de configuração, por exemplo.

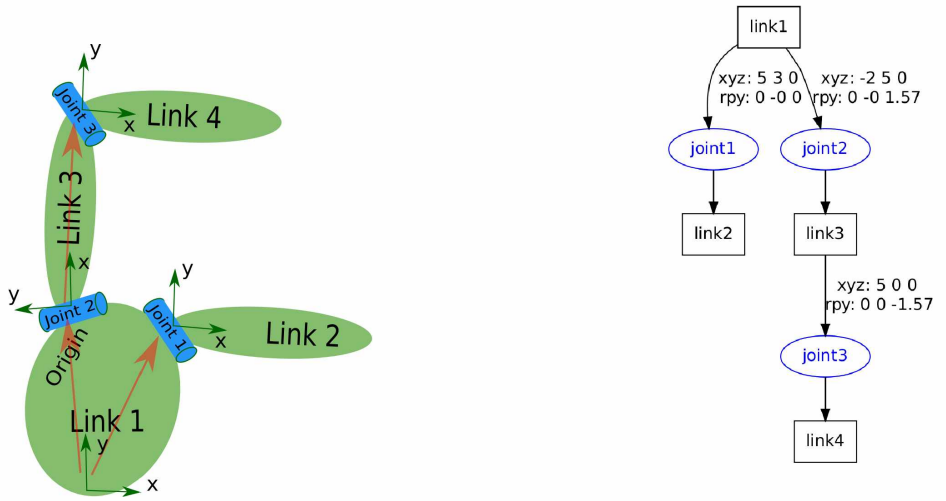
Geralmente, cada *package* oferece uma funcionalidade específica, podendo ser agrupada com outros *packages*, formando uma *stack* (pilha). Mesmo as ferramentas mais básicas do ROS são estruturadas em *packages*. Anualmente é lançada uma distribuição do ROS, com coleções das principais *stacks*. Além disso, o *site* do ROS mantém um repositório, onde qualquer pessoa pode requisitar hospedagem dos seus *packages* e *stacks* (OSRF, 2014).

3.1.2 Ferramentas Exploradas

Além dos componentes básicos apresentados, o ROS também oferece bibliotecas e ferramentas específicas comumente utilizadas em sistemas robóticos. Destacam-se algumas que foram utilizadas neste projeto.

Por exemplo, o ROS define um padrão de descrição de robôs denominado *Unified Robot Description Format (URDF)*, em que se especifica parâmetros geométricos, cinemáticos, dinâmicos, visuais e de colisão de robôs em arquivos. Deve-se definir, ao menos, a disposição das juntas (*joints*) e estruturas de ligação (*links*) (KUNZE; ROEHM; BEETZ, 2011). Para ilustrar como esta definição é feita, a Figura 8a mostra a estrutura de um robô sob o aspecto da disposição dos *links*, juntas e eixos de coordenada, e a Figura 8b apresenta um diagrama correspondente que explicita o relacionamento e relações numéricas dos seus componentes.

Figura 8 – Exemplo de modelo URDF



(a) Representação espacial simplificada (b) Diagrama relacional

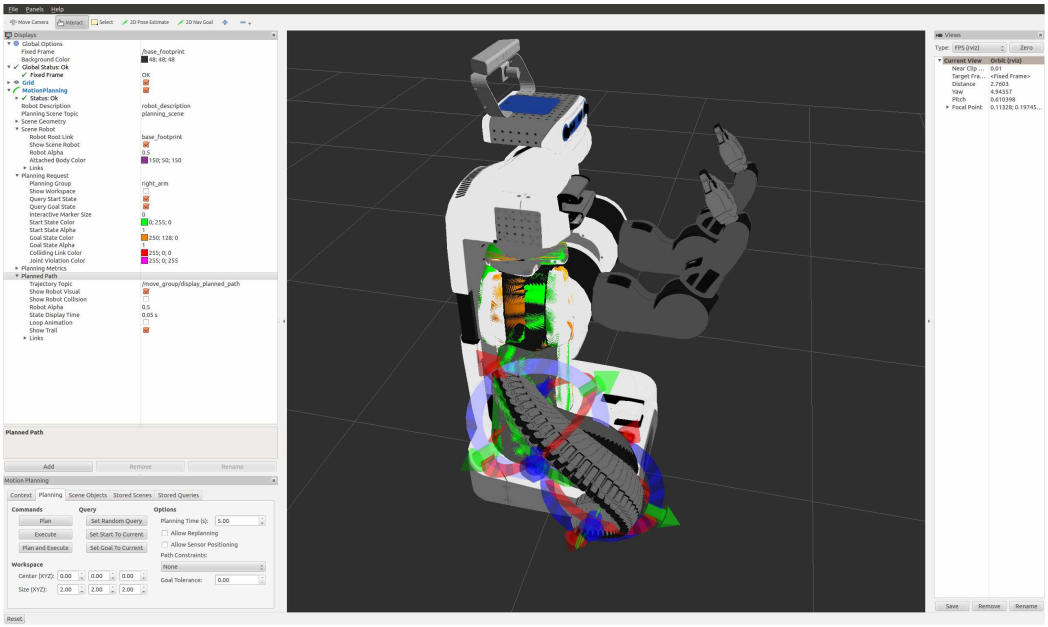
Fonte: OSRF (2017c)

Portanto, em um URDF, *links* são conectados entre si por juntas, formando uma estrutura em árvore, partindo de um *link* base fixo (no caso, o *link* 1). Especificam-se as coordenadas de origem, os deslocamentos "xyz"e as rotações "rpy"(ângulos de rotação nos eixos x, y e z, chamados em inglês como *roll*, *pitch* e *yaw*) das juntas em relação à origem do *link* anterior.

Outra ferramenta muito utilizada é o pacote *Transform Library*, abreviado como *tf*. Ele é projetado para monitorar a posição e orientação dos eixos de coordenadas, facilitando operações de transformações geométricas entre eles. Em robótica, é muito comum situações em que se deseja saber a posição de uma parte do robô em relação a determinado ponto. Por exemplo, um robô humanoide manipulador, para pegar um objeto, deve saber a posição de sua pinça em relação ao alvo a partir da posição estimada do mesmo em relação à câmera fixada na cabeça (FOOTE, 2013).

O ROS também se destaca pelas ferramentas de visualização. O *rviz* é o pacote com mais funcionalidades neste quesito. O *rviz* permite a visualização em três dimensões do que o robô está "pensando"e "sentindo". É possível representar diversos tipos de mensagens padrões do ROS, como nuvem de pontos (muito utilizado em navegação) e imagens de câmeras. Com base na descrição URDF do robô e o pacote *tf*, é possível visualizar o robô e as coordenadas monitoradas em tempo real. Visualizar várias informações da aplicação em uma única ferramenta é vantajoso, pois, dentre outras razões, permite identificar problemas muito mais rapidamente (OSRF, 2017b). A Figura 9 apresenta o modelo visual do robô PR-2 no *rviz*.

Figura 9 – Aplicação com robô PR-2 visualizado no *rviz*



Fonte: MoveIt!... (2017)

Enfim, o ROS também oferece um *framework* chamado *rqt* para desenvolver interfaces gráficas para o robô, dispondo de algumas interfaces prontas que podem ser integradas com às próprias com facilidade. Por exemplo, o *rqt_graph* mostra, graficamente, a estrutura de nós e tópicos em execução; o *rqt_publisher* permite publicar mensagens em tópicos facilmente, sendo adequado para experimentos rápidos; e o *rqt_plot* exibe gráficos dinâmicos de dados numéricos postados em tópicos ao longo do tempo (OSRF, 2017b).

3.2 ROS_CONTROL

O *ros_control* consiste em uma *stack* do ROS, contendo bibliotecas e programas voltados a facilitar o desenvolvimento e o gerenciamento de controladores para robôs de modo geral, minimizando as restrições de *hardware* e de aplicação. Especificamente, os pacotes são uma adaptação do projeto *pr2_mechanism*, utilizado para controlar o robô PR-2, tornando-o genérico para ser utilizado em outros projetos (MEEUSSEN, 2017).

Primariamente, o *ros_control* é desenvolvido pela empresa espanhola PAL Robotics e é utilizado em inúmeros projetos ao redor do mundo. A Figura 10 apresenta exemplos de robôs que utilizam a ferramenta.

Figura 10 – Exemplos de robôs que utilizam o *ros_control*



(a) StockBot (b) Florian (c) LWA 4D

Fonte: Tsouroukdissian (2014)

A Figura 10a apresenta o robô StockBot, que é vendido pela PAL Robotics para realizar controle de estoques; a Figura 10b mostra o robô humanoide "Florian", desenvolvido pela equipe *ViGIR* para competir no *DARPA Robotics Challenge* (DRC)[1]; e a Figura 10c apresenta o robô manipulador "LWA 4D", comercializado pela empresa SCHUNK (TSOUROUKDISSIAN, 2014).

[1] Competição organizada pela *Defense Advanced Research Projects Agency* dos Estados Unidos (DARPA), em que robôs humanoides devem executar tarefas complexas em situações de desastre.

Em suma, as características mais relevantes do *ros_control*, que podem justificar o seu sucesso e que contribuíram para que fosse selecionado para ser usado neste projeto, são:

- Possui uma estrutura simples e flexível que torna o desenvolvimento mais ágil e permite a realização de modificações ou expansões com facilidade;
- Permite desenvolver controladores independentemente da implementação de comunicação com *hardware*, o que potencializa a reutilização de código, o desenvolvimento colaborativo e o uso em múltiplas aplicações;
- É preparado para ser utilizado em sistemas de tempo real, mesmo sendo integrado ao ROS, dispondo de recursos que facilitam o uso de suas ferramentas de forma segura (TSOUROUKDISSIAN, 2014).

3.2.1 Estrutura

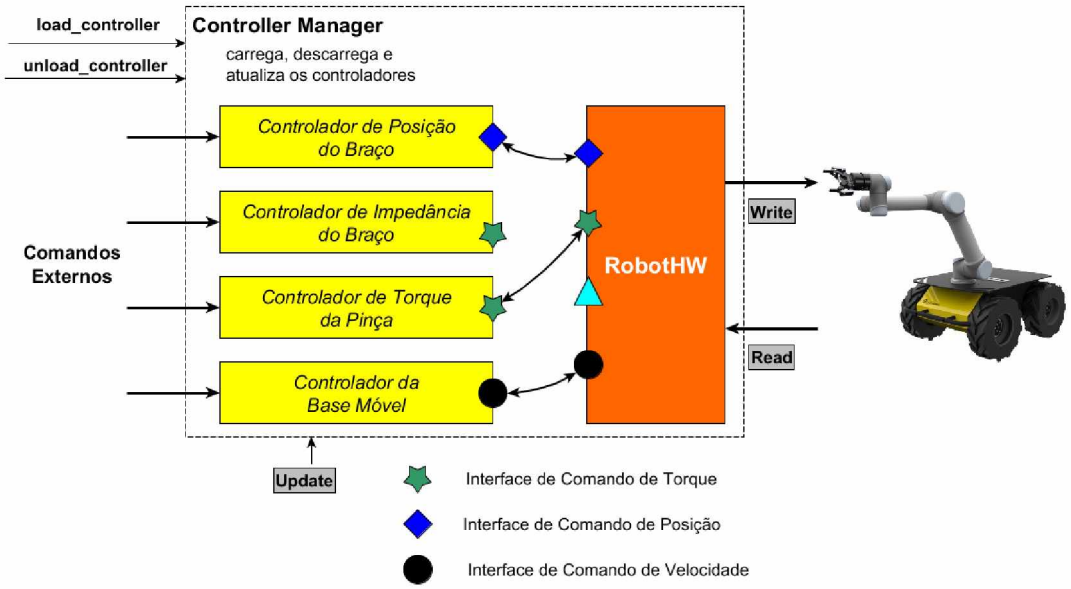
O *ros_control* define uma estrutura de *software* simples, flexível e bem definida, sobre as quais o usuário pode desenvolver códigos para a sua aplicação. Basicamente, ele define 4 componentes principais:

- ***RobotHW:*** interface de comunicação com o *hardware* do robô. Oferece as chamadas *hardware interfaces* por onde o *RobotHW* interage com os controladores. Implementa um método de escrita (*write*), que executa os comandos passados pelas interfaces, e um método de leitura (*read*), que capta os dados dos sensores e os disponibilizam nas interfaces correspondentes;
- ***Hardware Interface:*** são canais de comunicação entre os controladores e o *RobotHW*. Funcionam como local de armazenamento de comandos dos controladores para o *RobotHW* e de dados lidos do *RobotHW* para os controladores.
- ***Controller:*** representa um controlador. Implementa o método *update* que, através das *hardware interfaces*, pode utilizar os últimos dados lidos do robô (da última chamada do método *read* do *RobotHW*) e configurar comandos a serem realizados na próxima escrita (próxima chamada do método *write* do *RobotHW*). Os controladores também oferecem uma interface de programação, referida em inglês como *Application Programming Interface* (API), para receber comandos ou fornecer dados;
- ***Controller Manager:*** é possível que um mesmo robô possua vários controladores. Assim, o *Controller Manager* é um componente que gerencia o carregamento, descarregamento, ativação, parada, substituição e atualização dos controladores. Também oferece o método *update*, que dispara as atualizações de todos os *Controllers* que estiverem ativos.

Com exceção do *Controller Manager*, todos os componentes dispõem-se como classes C++ abstratas, das quais os usuários podem criar suas próprias versões por meio de herança de classes. Em uma mesma aplicação, geralmente há uma instância do *RobotHW* e do *Controller Manager* e múltiplas de *Controllers* e *Hardware Interfaces*

Para ilustrar melhor como estes elementos se relacionam, a Figura 11 apresenta um exemplo de configuração do *ros_control* aplicado a um manipulador móvel.

Figura 11 – Arquitetura do *ros_control* aplicado a robô manipulador móvel.



Fonte: Produzido pelo autor.

Neste exemplo, observa-se que o *RobotHW* oferece interfaces para envio de comando de posição, velocidade, torque e uma quarta não especificada, indicada apenas para ilustrar que *hardware interfaces* são propriedades do *RobotHW*, podendo ser utilizadas, ou não, pelos controladores. Ela poderia representar uma interface onde o *RobotHW* disponibilizaria dados de uma IMU, por exemplo.

A Figura 11 apresenta quatro controladores carregados, sendo que cada um controla uma parte do robô (base móvel, braço ou pinça) por meio de uma *hardware interface*. Por exemplo, o controlador da base móvel controla os motores acoplados às rodas do robô por meio de comandos de velocidade.

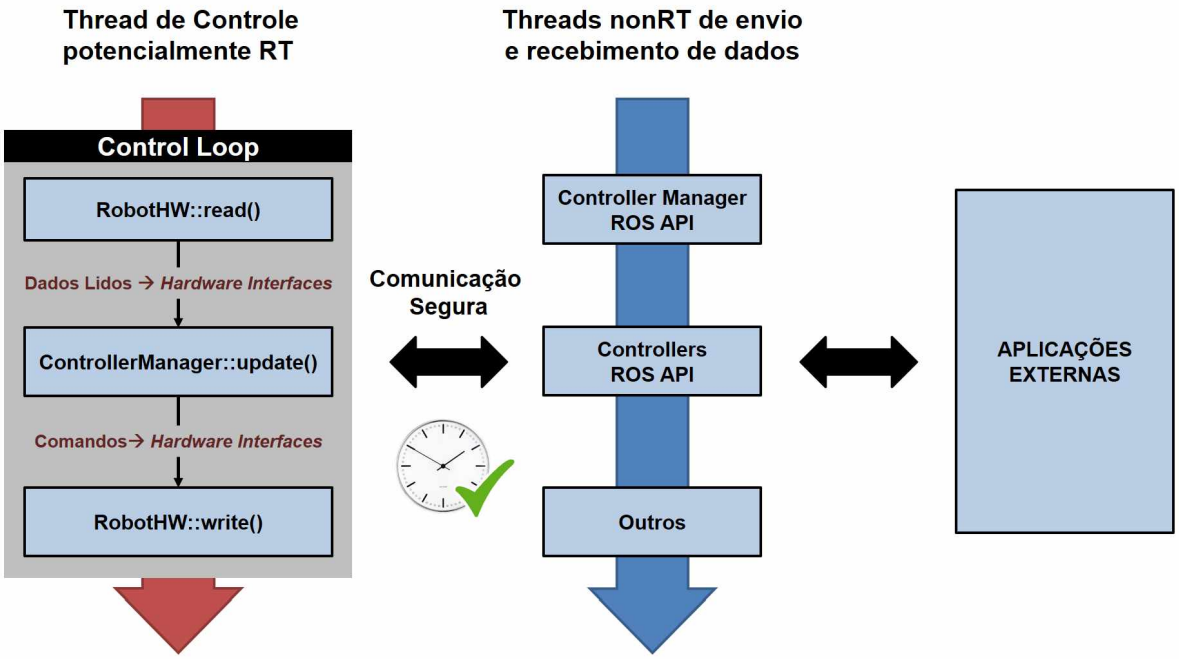
Por padrão, o *Controller Manager*, gerencia os recursos utilizados por cada controlador, evitando que dois controladores tentem manipular os mesmos elementos. Assim, neste exemplo, o controlador de impedância do braço não está ativado, pois já existe um controlador de posição para a mesma parte do robô. Apesar disso, é possível, em execução,

parar um controlador e iniciar o outro e, embora não exemplificado, não há problema em mais de um controlador utilizar a mesma interface.

A Figura 11 também ilustra a interface de comunicação com o *Controller Manager* e os controladores. O *Controller Manager* oferece *ROS Services* e métodos para o usuário comandar o (des)carregamento e a (des)ativação de controladores. De forma semelhante, aplicações externas podem trocar dados com os controladores, geralmente por meio de *ROS Topics*, enviando os estados desejados para cada controlador e/ou recebendo dados processados. Embora não representado, é comum que os controladores e o *RobotHW* também utilizem dados de configuração do *parameter server*.

O *ros_control* não define componentes que gerenciem a chamada dos métodos principais de escrita e leitura do *RobotHW* e de atualização do *ControllerManager*. Porém, propõe a construção de um *loop* de controle de controle como mostra a Figura 12.

Figura 12 – Estrutura de laço de controle para o *ros_control*



Fonte: Adaptado de Tsouroukdissian (2014)

A cada iteração, executa-se o método *read* de *RobotHW*, que faz uma leitura do estado do robô e disponibiliza os valores nas interfaces correspondentes. Em seguida, executa-se o método *update* do *Controller Manager*, que atualiza (chama o método *update* de) todos os *Controllers* ativos. Estes, por sua vez, podem utilizar os dados recém-atualizados nas interfaces de leitura, processar os comandos e preencher as interfaces de escrita. Enfim, executa-se o método *write* de *RobotHW*, que pega os comandos definidos nas interfaces e envia aos atuadores.

A Figura 12 também mostra que os procedimentos de recebimento e envio de dados de/para aplicações externas dos controladores e do *Controller Manager*, geralmente feita por ROS *topics* e *services*, acontecem em outras *threads* que não são de tempo real. Porém, a troca de dados com o *loop* de controle é feita de forma segura, ou seja, sem comprometer o desempenho de tempo real do laço. Em outras palavras, o *ros_control* possibilita a *thread* de controle enviar dados e receber comandos de aplicações externas com o auxílio de *threads* auxiliares não de tempo real sem que ocorra atrasos que degradem o determinismo desejado.

Ressalta-se que o desenvolvedor, para usufruir desta funcionalidade, deve se atentar para não incluir códigos que ferem esta segurança ao desenvolver novos controladores e o *RobotHW*, principalmente ao implementar os métodos *update*, *read* e *write*. Deve-se tomar cuidado para não incluir rotinas de prazo de execução não determinísticos, como de comunicação em rede, de espera de outras aplicações que não são de tempo real ou que se comunicam com dispositivos de entrada e saída. Como o ROS não oferece recursos para implementar *softwares* de tempo real oficialmente, o *ros_control* disponibiliza o pacote *realtime_tools*[2] com ferramentas que possibilitam a integração de aplicações de tempo real com o ROS.

3.2.2 Utilização

Com esta arquitetura, o trabalho dos usuários consiste em desenvolver versões próprias dos componentes do *ros_control* de acordo com as necessidades. Esta característica, além de proporcionar flexibilidade e melhor direcionamento no desenvolvimento, facilita o compartilhamento de código com terceiros. Neste sentido, os próprios desenvolvedores do *ros_control* oferecem componentes genéricos prontos para serem reutilizados. O *ros_control* já disponibiliza diversas interfaces de *hardware* como:

- **Joint State Interface:** disponibiliza os dados de posição, velocidade e torque dos motores lidos pelo *RobotHW* para os controladores. A este conjunto de informações da junta dá-se o nome de estado da junta (do inglês *joint state*);
- **Joint Command Interfaces:** para envio de um comando numérico para cada junta. Apresentam-se em 3 versões principais, variando o tipo de comando:
  - *Effort Joint Interface:* comando de esforço (força ou torque)
  - *Velocity Joint Interface:* comando de velocidade
  - *Position Joint Interface:* comando de posição
  - *"PosVel"Joint Interface:* comando de posição e velocidade

[2] <http://wiki.ros.org/realtime_tools>

- ***ImuSensorInterface:*** para o *RobotHW* disponibilizar dados de IMU aos controladores

De forma semelhante, os criadores do *ros_control* oferecem controladores de uso genérico prontos em outra *ROS stack*, chamada *ros_controllers*[3]. Segue alguns dos controladores:

- ***Joint State Controller:*** controlador que interage com *Joint State Interfaces* e que, simplesmente, repassa o estado das juntas a um tópico do ROS em tempo real, podendo ser utilizados por aplicações externas, como as ferramentas de visualização do ROS *rviz* e *rqt_plot*;
- **Effort Controllers:** controladores que interagem com *Effort Joint Interface*.
  - *Joint Effort Controller:* simplesmente repassa os comandos de esforço recebidos;
  - *Joint Position Controller:* implementa controle PID de posição com base nos dados lidos de posição e envio de comando de torque;
- ***Diff Drive Controller:*** interage com *Velocity Joint Interface* para controlar bases móveis de tração diferencial. Recebe comandos de velocidade angular e linear, convertendo-os em comandos de velocidade aos motores. Este controlador também estima a posição do robô no espaço ao longo do tempo com base em leituras de posição dos motores (odometria), disponibilizando a informação em ROS *Topics*, podendo ser integrado com outros pacotes ROS como *tf* e *navigation* (MAGYAR, 2017).
- ***Joint Trajectory Controller:*** recebe trajetórias de posição e opcionalmente, de velocidade, aceleração e/ou torque das juntas, repassando os comandos aos motores de forma sincronizada aos motores. O pacote oferece três versões deste controlador, variando-se a interface de *hardware* com qual interagem: o primeiro simplesmente repassa comandos de posição para uma *Position Joint Interface* e os outros dois interagem com *VelocityJointInterface* e *EffortJointInterface* para controlar as posições por controle PID através comandos de velocidade e torque respectivamente. Este controlador também destaca-se por realizar interpolação *spline* de até a 5ª ordem para amostrar pontos da trajetória em tempos não explicitamente definidos (TSOUROUKDISSIAN, 2017).

Portanto, na prática, os usuários se preocupam em desenvolver o *RobotHW*, o laço de controle e, quando os componentes genéricos do *ros_control* não atendem a aplicação, os controladores e as interfaces de *hardware*.

[3] <http://wiki.ros.org/ros_controllers>

4 Implementação

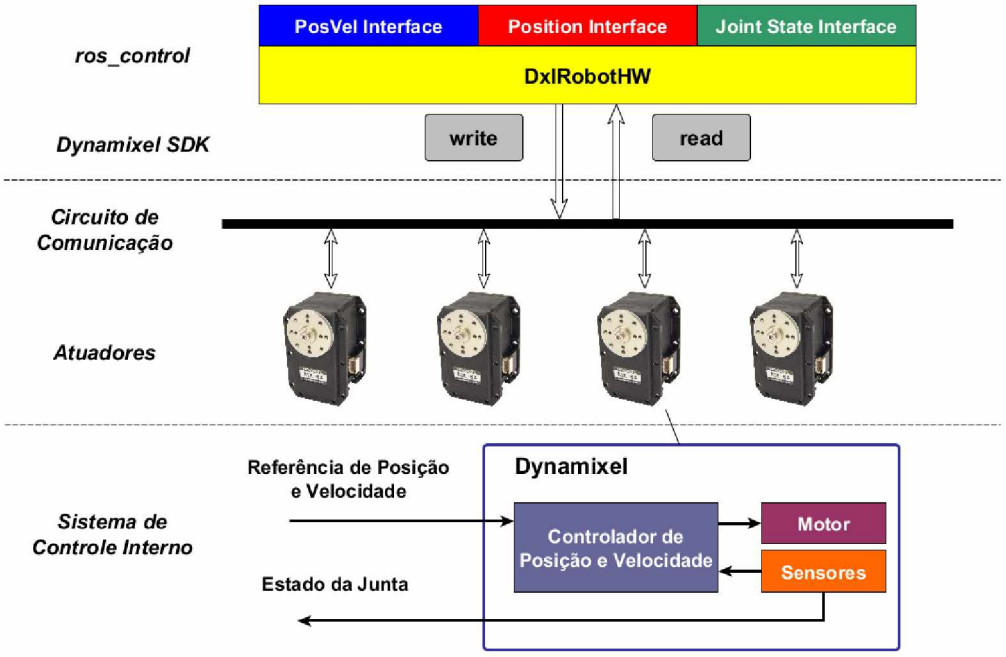
Em relação ao desenvolvimento de código, o trabalho consistiu em, basicamente, desenvolver versões próprias dos componentes do *ros_control*, além de programas específicos de testes.

4.1 DxlRobotHW

O *DxlRobotHW* é um *RobotHW* genérico para comunicar com robôs compostos por atuadores *Dynamixel*. No momento, ele oferece três *hardware interfaces* nativas do *ros_control*: *JointStateInterface*, *PositionJointInterface* e *PosVelJointInterface*. Portanto, o *DxlRobotHW* é capaz de receber comandos de posição, acompanhados ou não por comando de velocidade, além de disponibilizar dados sobre o estado das juntas. Embora ainda não implementado, os motores também permitiriam envio de comando de velocidade e torque por meio das interfaces *VelocityJointInterface* e *EffortJointInterface*.

Internamente, o *DxlRobotHW* se assemelha com a solução da EDROM em muitos aspectos. Aproveita-se o controlador de posição e velocidade embutido dos motores, apenas repassando os comandos recebidos pelas interfaces para os motores. O código-fonte do *DxlRobotHW* está disponível no Apêndice B.1 e sua arquitetura é ilustrada na Figura 13.

Figura 13 – Arquitetura de controle dos motores *Dynamixel* com *DxlRobotHW*



Fonte: Produzido pelo autor.

No topo, rodando em um SO, encontra-se o *DxlRobotHW*, que fará interface com os controladores nas camadas superiores através das interfaces. Ao chamar os métodos *read* e *write*, o *DxlRobotHW* faz uso da *Dynamixel* SDK, biblioteca disponibilizada pela empresa responsável (ROBOTIS), que oferece funções básicas de construções e interpretação de pacotes, e de transporte de dados entre computador e motor via USB. Sinais elétricos com os pacotes são enviados aos atuadores por meio de circuitos apropriados até ser recebido pelo programa instalado no *Dynamixel*. Ressalta-se que os motores são ligados em paralelo em um mesmo barramento, podendo receber comandos em um mesmo pacote de dados. Os comandos recebidos alimentam o sistema de controle de posição e velocidade embutido que, em uma frequência maior do que a de comunicação com o *DxlRobotHW*, atua nos motores com base na realimentação dos dados lidos dos sensores. No caso em que se envia comandos de leitura, o *Dynamixel* retorna os dados lidos, que definem o estado da junta, e os encaminha para o *DxlRobotHW*.

A *Dynamixel SDK*, originalmente, oferece apenas recursos básicos de comunicação. Assim, sobre ela, desenvolveu-se códigos que agregam novas funcionalidades e abstração de detalhes. O protocolo de comunicação dos atuadores *Dynamixel* define unidades de posição, velocidade e torque específicas para cada modelo. Assim, o *DxlRobotHW* abstrai estes detalhes, realizando identificação de modelo e conversão de unidades automaticamente. Por padrão, posição, velocidade e torque são expostos em radianos, radianos por segundo e porcentagem em relação ao torque máximo, respectivamente.

Além disso, o *DxlRobotHW* recebe como parâmetro uma lista de descrição das juntas controladas para facilitar a identificação e a definição dos comandos. Definiu-se a estrutura chamada *DxlJointId*, que reúne as seguintes informações:

- **ID:** identificador numérico único, propriedade específica dos motores *Dynamixel*, configurada em um de seus registradores;

- **Nome:** identificador alfanumérico único definido pelo usuário. A atribuição de nomes sugestivos facilita o desenvolvimento;

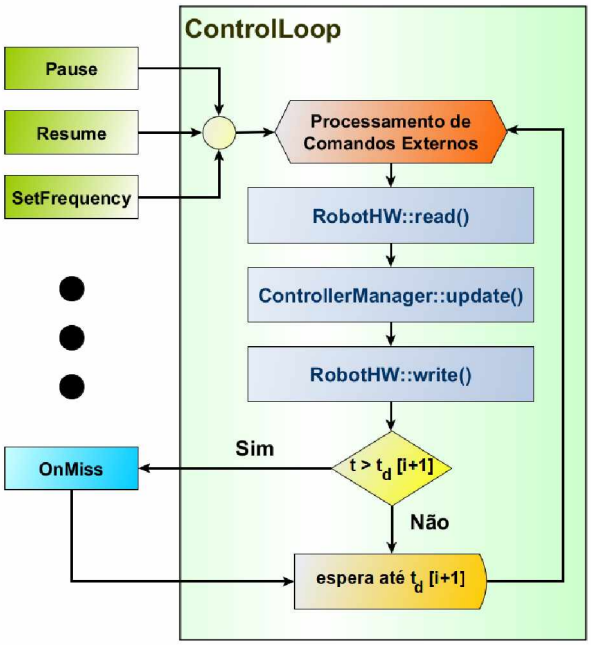
- **Referência:** posição de referência em radianos, em relação à posição zero do motor. Os valores de posição enviados e lidos de um junta são relativos à posição de referência configurada para ela;

- **Direção:** especifica a direção de giro dos motores. +1 indica que os valores de posição crescem no sentido anti-horário e decrescem no sentido horário, e vice-versa quando configurado como -1.

4.2 ControlLoop

Desenvolveu-se uma classe[1] que representa um laço de controle genérico com capacidade de ser executado em sistemas operacionais Linux de tempo real, com recursos de monitoramento e parametrização em tempo de execução. O Apêndice B.2 contém o código-fonte e a Figura 14 apresenta a estrutura do novo componente:

Figura 14 – Estrutura básica do *ControlLoop*



Fonte: Produzido pelo autor.

Em concordância com a proposta de estruturação do *loop* de controle apresentada na sessão 3.2.1, o *ControlLoop* possui uma *thread* de tempo real interna dedicada a executar a sequência de comandos *RobotHW::read()*, *ControllerManager::update()* e *RobotHW::write()* periodicamente. A classe também oferece métodos para que o usuário possa ajustar a frequência e iniciar, pausar, continuar e parar o *loop* externamente de forma assíncrona e em tempo de execução. A rotina de captação e tratamento destes comandos se dá no início de cada iteração, sendo projetada a não absorver atrasos das *threads* externas que chamam os comandos, assim como sugere a Figura 12 (sessão 3.2.1), para não comprometer o desempenho de tempo real da *thread* interna.

O *ControlLoop* também permite que o desenvolvedor defina o comportamento desejado nos casos em que os prazos especificados de comunicação não são cumpridos por meio da invocação do método *OnMiss*. A estratégia de temporização do laço adotada, baseia-se em iniciar cada iteração em pontos no tempo igualmente espaçados. Assim,

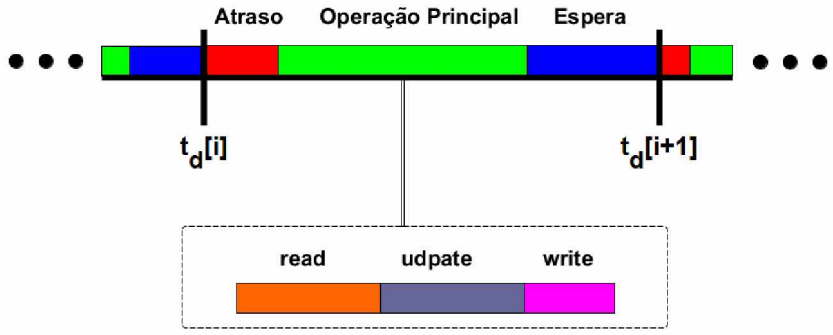
[1] Na realidade, são um conjunto de classes. No código, o *ControlLoop* descrito nesta sessão refere-se à classe *RosControlLoop*.

a partir do período T especificado, partindo da primeira iteração, define-se os tempos desejados para iniciar a execução de cada iteração i segundo a Equação 4.1.

$$t_d[i] = Ti \qquad\qquad i = 0, 1, 2, ... \tag{4.1}$$

Ao término de cada iteração, roda-se um comando de espera até o próximo momento planejado. Quando a espera termina, há um tempo de atraso até a execução dos comandos principais do laço, que também leva determinado tempo para ser executado (RAMMIG et al., 2009). Portanto, a utilização do tempo em cada período é distribuído como apresenta a Figura 15.

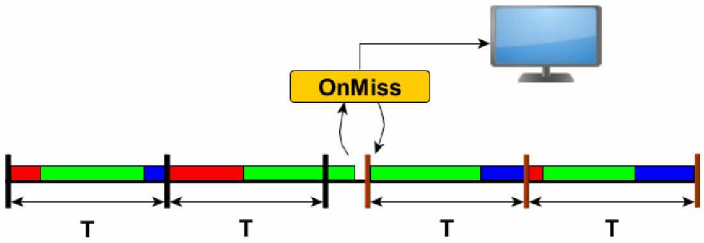
Figura 15 – Distribuição temporal de cada iteração do *ControlLoop*



Fonte: Produzido pelo autor.

Dessa forma, a duração do período deve ser superior a este tempo para que os prazos possam ser cumpridos. Infelizmente, a duração de cada parcela pode sofrer variações por diversos motivos, como por interrupções externas (minimizadas em RTOS) e problemas de comunicação com *hardware*. Assim, o método *OnMiss* é invocado quando os limites de tempo de alguma iteração é extrapolado, assim como a Figura 16 ilustra.

Figura 16 – Exemplo de situação de acionamento do método *OnMiss* do *ControlLoop* com comportamento padrão.



Fonte: Produzido pelo autor.

O código dá liberdade para que o comportamento deste método seja definido pelo desenvolvedor para que possa intervir na execução do *loop* quando os prazos são

extrapolados, podendo implementar rotinas de parada de emergência ou ações corretivas por exemplo. Por padrão, este método exibe uma mensagem na tela com o tempo extrapolado e inicia a próxima iteração, atualizando os próximos tempos de envio com base no tempo corrente, tentando manter a frequência especificada. Este recurso de monitoramento pode ser utilizado de diversas formas. Por exemplo, a ocorrência muitos atrasos de magnitude próximas umas das outras pode significar que o tempo médio de execução da operação principal esteja excedendo o período especificado. Neste caso, o valor médio do tempo extrapolado exibido na tela pode ser acrescido ao período, alcançando-se a frequência máxima do sistema.

Outro motivo para que o método *onMiss* seja chamado com frequência é a ocorrência de atrasos significativos no início de cada iteração. Geralmente, este tipo de problema é identificado quando os valores apresentados na tela apresentam variações significativas (na ordem de milissegundos) e o *loop* é executado em uma *thread* que não seja de tempo real em paralelo com outros processos que alocam muitos recursos da máquina. Nestes casos, os atrasos também podem ocorrer durante a operação principal quando o processo é interrompido pelo SO para compartilhar o uso do processador com outras tarefas.

4.3 posvel_controllers

Embora o pacote *ros_control* dispõe da *PosVelJointInterface*, *ros_controllers* não oferece controladores que interagem com ela. Os motores *Dynamixel*, por sua vez, podem receber comandos de posição e velocidade, funcionalidade usada com frequência na implementação atual da EDROM. Assim, desenvolveu-se dois controladores básicos para esta interface:

- ***Joint Trajectory Controller:*** adaptação do controlador de trajetórias das juntas do pacote *ros_controllers* à interface *PosVelJointInterface*;
- ***"PosVel"Joint Controller***[2]: permite enviar comandos isolados (em momentos esporádicos) de posição e velocidade a um conjunto de juntas. Equivalente ao controlador *PositionControllers/JointGroupPositionController* do pacote *ros_controllers*.

O código destes componentes está disponível do Apêndice B.3.

[2] Na realidade, o nome do controlador no código é *JointGroupPosVelController*.

5 Testes e Análises

Para evidenciar as vantagens e funcionalidades da nova ferramenta, propõe-se uma série de testes em robô real. Especificamente, apresentam-se exemplos de uso integrado do sistema com ferramentas do ROS e realizam-se análises de desempenho em sistemas de tempo real.

5.1 Configuração do Ambiente

5.1.1 *Hardware*

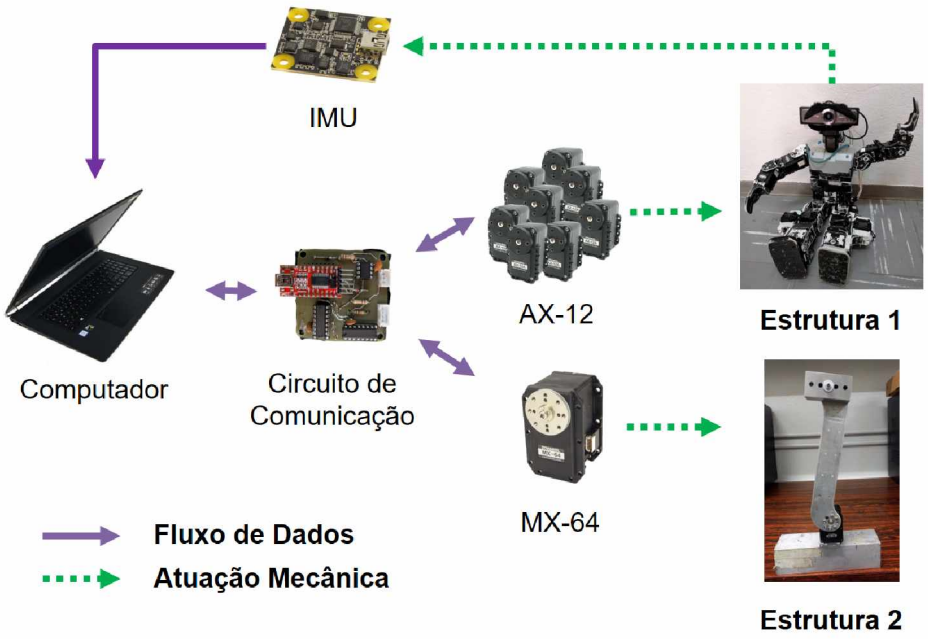
Os equipamentos utilizados nos testes resumem-se a:

- **Atuadores:** oito motores *Dynamixel* do modelo AX-12 e um do modelo MX-64;
- **Computador:** *notebook* Acer Aspire V15 Black, equipado com processador Intel i7-4720HQ 2.60GHz e 8GB de memória RAM;
- **Placa de Comunicação com Motores:** circuito desenvolvido pela equipe EDROM com base no projeto USBzDXL[1]. Este circuito repassa os comandos aos motores via protocolos TTL e RS-485 enviados do computador pela porta USB;
- **IMU:** Sensor *PhidgetSpatial 3/3/3 Basic 1044* da empresa Phidgets[2]. Este dispositivo é equipado com sensores acelerômetro, giroscópio e bússola. A IMU é utilizada para estimar a orientação do tronco do robô e exibir no *rviz*.

Para testes de controle de movimento, montou-se duas estruturas. A primeira é um robô humanoide constituído por três atuadores AX-12 em cada braço e outros dois na cabeça, além da IMU fixada no seu tronco. Já a segunda estrutura é formada por um motor MX-64 fixado, cujo eixo é conectado à extremidade de uma barra de metal, aplicando torque sobre a mesma. A Figura 17 ilustra a arquitetura de *hardware*, apresentando os componentes mencionados e suas conexões.

[1] <https://github.com/MojtabaKarimi/AUTMan-USBzDXL>
[2] <https://www.phidgets.com/?tier=3&catid=10&pcid=8&prodid=30>

Figura 17 – Arquitetura de *hardware* do ambiente de testes



Fonte: Produzido pelo autor.

5.1.2 *Software*

A Tabela 1 apresenta as especificações de versão mais relevantes dos *softwares* e bibliotecas utilizados nos testes.

Tabela 1 – Especificação de versão das ferramentas de *software* relevantes

Ferramenta de Software	**Especificação**
Sistema Operacional	Ubuntu 16.04.2 (Linux)
Kernel de Propósito Geral	4.8.0-56
Kernel de Tempo Real	4.8.15-rt10
ROS	Kinect 2016
Dynamixel SDK	3.4.5
ros_control	0.11.4
ros_controllers	0.12.3

Fonte: Produzido pelo Autor.

Os procedimentos adotados para configurar e instalar o *kernel* de tempo real com o pacote *PREEMPT_RT* encontram-se no apêndice A. Em todos os testes realizados no RTOS, as *threads* de tempo real foram configuradas com a configuração padrão do *ControlLoop*: escalonador *Round-Robin* (SCHED_RR) e prioridade 31.

Em relação ao ROS, além dos pacotes padrão da distribuição, do *ros_control*,

ros_controllers e do *realtime_tools*, utilizam-se os pacotes *phidgets_drivers*[3] e *imu_filter_madgwick*[4] para possibilitar o uso da IMU.

Os códigos de implementação dos componentes e de comunicação com o motores foram desenvolvidos em linguagem C++, enquanto testes de envio de comandos aos controladores foram feitos em Python. Todos os códigos desenvolvidos encontram-se em repositório no GitHub através do endereço <https://github.com/awtreth/tcc>. O trabalho foi subdividido entre os seguintes pacotes ROS:

- **control_loop:** implementação do *ControlLoop*;
- **dxl_robot_hw:** implementação do *DxlRobotHW* e programas de teste com a estrutura 2 da Figura 17;
- **posvel_controllers:** implementação dos novos controladores da sessão X que interagem com a *PosVelInterface*;
- **bioloid_example:** programas de testes e arquivos de configuração da estrutura 1 da Figura 17.

A estrutura de organização dos pacotes seguiu o padrão proposto por Lentin (2015b). Além destes pacotes, porém, estruturou-se mais dois projetos independentes:

- **dxl_interface:** biblioteca de comunicação com motores construído sobre a *Dynamixel SDK* projetada para abstrair certos detalhes e adicionar novas funcionalidades. É utilizada no pacote *dxl_robot_hw* e, por enquanto, apenas a classe *ModelSpec*, que abstrai dados dos modelos, está em funcionamento;
- **dxl_tests:** testes isolados, como de desempenho de RTOS e de comunicação com os motores, descritos nas próximas sessões.

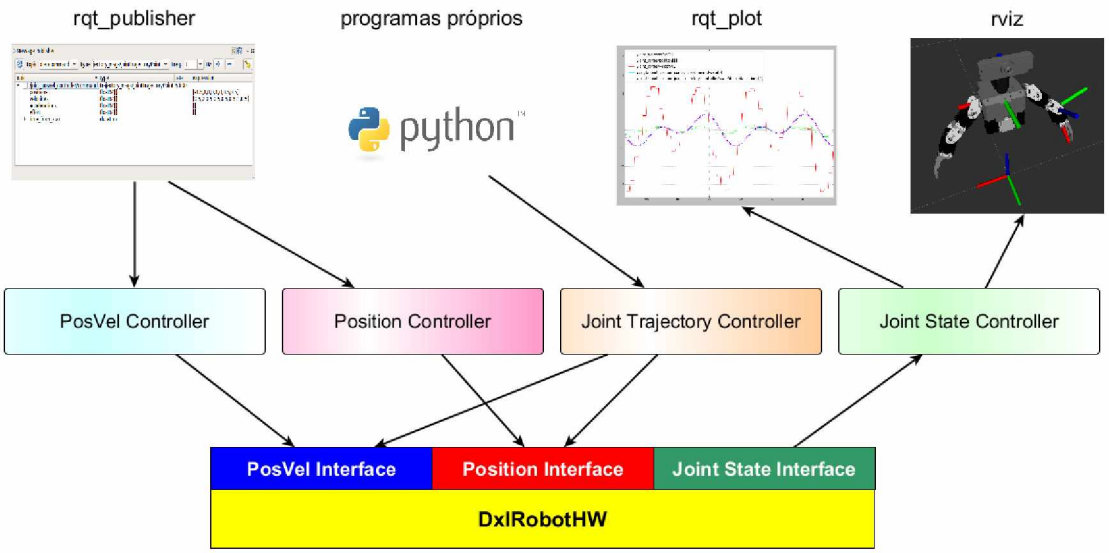
Ressalta-se que o repositório contém instruções básicas de instalação das ferramentas desenvolvidas.

[3] <http://wiki.ros.org/phidgets_drivers>
[4] <http://wiki.ros.org/imu_filter_madgwick>

5.2 Integração com o ROS

Ao utilizar interfaces de *hardware* desenvolvidas pelo *ros_control* (como foi feito com o *DxlRobotHW*), é possível reutilizar controladores do pacote *ros_controllers* ou de terceiros que seguem o padrão proposto. Assim, foram realizados vários testes de integração do *DxlRobotHW* com este controladores genéricos, integrando-os com algumas ferramentas do ROS. A Figura 18 resume a arquitetura dos principais testes realizados.

Figura 18 – Arquitetura do teste de integração com o ROS



Fonte: Produzido pelo autor.

Por meio da ferramenta *rqt_publisher*, testou-se o envio de comandos aos controladores que esperam mensagens contendo um comando por junta, como o *PositionController* e o novo *PosVelController*. Para testar os *JointTrajectoryControllers*, que esperam uma sequência de comandos para as juntas, desenvolveu-se programas em Python que constroem e enviam mensagens com as trajetórias.

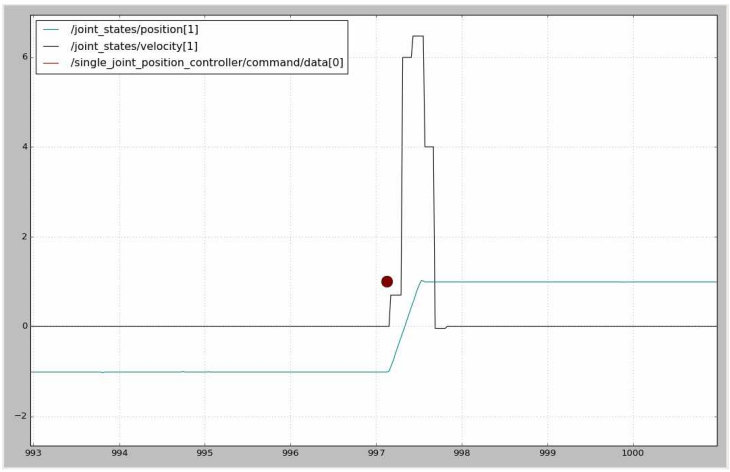
Em relação a visualização de dados, utilizou-se os pacotes *rqt_plot* e *rviz*. Com o *rqt_plot*, exibe-se, em tempo de execução, valores de posição, velocidade e torque das juntas publicados pelo *Joint State Controller* enquanto o *rviz* é usado para exibir o estado do robô em 3D, refletindo a postura do tronco, braços e cabeça do robô real.

5.2.1 Funcionalidades Básicas

A partir do momento em que se concluiu as primeiras versões funcionais do *DxlRobotHW* e do *ControlLoop*, houve um salto considerável no potencial de expansão a curto prazo da solução. Rapidamente, foi possível enviar comandos de posição e velocidade às juntas através do *rqt_publisher* e visualizar seu estado em tempo real pelo *rqt_plot*. As

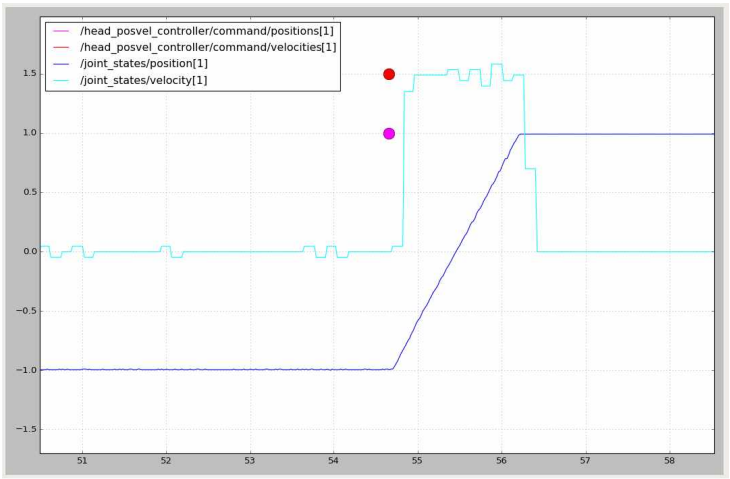
Figuras 19 e 20 apresentam a resposta ao degrau de um motor AX-12 controlado por um *PositionController* e um *PosVelController* respectivamente.

Figura 19 – Resposta a comando de posição de motor AX-12 no *rqt_plot*



Fonte: Produzido pelo autor.

Figura 20 – Resposta a comando de posição e velocidade de motor AX-12 no *rqt_plot*



Fonte: Produzido pelo autor.

À primeira vista, observa-se que os gráficos das Figuras 19 e 20 não apresentam legendas nem rótulos claros. Isso acontece, pois o *rqt_plot* não se preocupa com a geração de gráficos para documentação, sendo projetado apenas para oferecer recursos básicos de visualização em tempo real. Sua utilização se resume a especificar os tópicos que recebem as mensagens a serem plotadas (listados no quadro superior esquerdo das Figuras), a resolução dos eixos e propriedades visuais dos marcadores. A partir do momento em que os tópicos recebem dados, eles são plotados dinamicamente.

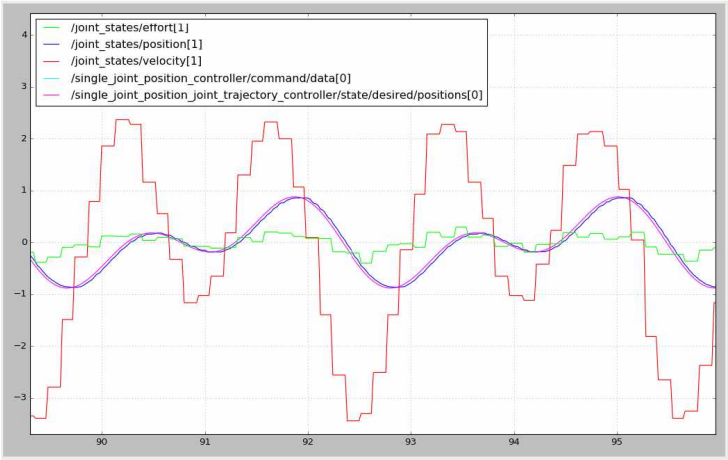
Desse modo, as curvas das Figuras correspondem aos dados de posição e velocidade lidos dos sensores, enquanto os pontos destacados indicam, na Figura 19, o comando de

posição e, na Figura 20, os comandos de posição e velocidade. Em ambos os casos, partiu-se da posição $-1rad$ para $+1rad$ com velocidade aproximadamente constante.

Com base na documentação do motores *Dynamixel*(ROBOTIS, 2017a), conclui-se que as respostas obtidas estão condizentes com o esperado. O atuador, ao receber somente o comando de posição, Figura 19, tenta levar sua posição até a desejada o mais rápido possível, o que justifica o pico de velocidade. Quando o comando de posição é acompanhado pelo comando de velocidade, o atuador tenta aproximar a velocidade média de trajeto à especificada, o que realmente acontece (Figura 20), visto que a velocidade mantém-se próxima à especificada durante a realização do movimento.

Da mesma forma, testou-se o envio de trajetórias de posição. A Figura 21 apresenta o estado da junta de um motor AX-12 controlado por pelo *Joint Trajectory Controller* com sinal de referência contínuo e periódico, sendo monitorado pelo *rqt_plot*.

Figura 21 – Estado da junta de motor AX-12 no *rqt_plot* em execução de trajetória

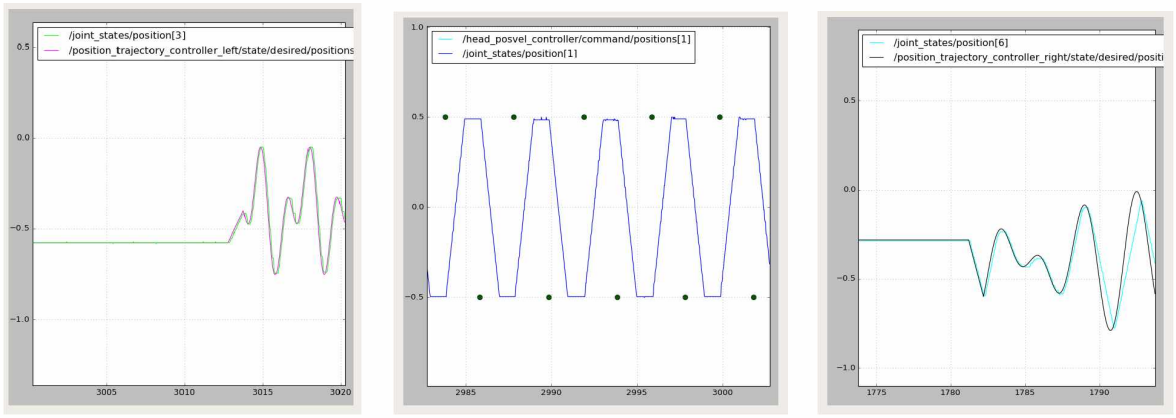


Fonte: Produzido pelo autor.

A Figura 21 apresenta as posições desejada e lida (em azul e rosa), a velocidade (em vermelho) e o torque (em verde). Observa-se que o sinal de torque acompanha o sinal da velocidade corretamente, porém apresentam descontinuidades em degraus ao longo de todo o trajeto. Este comportamento, porém, não é observado ao realizar o mesmo teste com motores MX-64, podendo ser uma deficiência particular dos motores AX-12, mas que não compromete o seu desempenho significativamente.

Enfim, testou-se a capacidade de controlar partes diferentes do robô de forma independente, uma das melhorias propostas perante a solução da EDROM. Configurou-se a cabeça para receber comandos isolados de posição e velocidade pelo *rqt_publisher* e, para cada braço, carregou-se um *Joint Trajectory Controller*, que recebe trajetórias de posição de dois programas isolados. A Figura 22 mostra as posição ao longo do tempo de um dos motores de cada parte controlada independentemente sendo visualizada no *rqt_plot*.

Figura 22 – Visualização no *rqt_plot* da posição de três juntas controladas de forma independente



(a) (b) (c)

Fonte: Produzido pelo autor

Os gráficos das Figuras 22a e 22c são das juntas de movimento lateral dos braços esquerdo e direito respectivamente, enquanto que o que a Figura 22b é referente à junta que move a cabeça horizontalmente. Embora os marcadores de tempo no eixo horizontal dos gráficos estejam diferentes, a resolução é a mesma e a imagem foi capturada enquanto os dados eram exibidos dinamicamente em uma mesma janela do *rqt*. Neste caso, o controlador da cabeça já estava em movimento, enquanto o do braço direito e do esquerdo foram ativados um após o outro.

Esta capacidade de controlar partes do sistema de maneira independente se deve, em grande parte, à existência das *hardware interfaces* e de um fluxo único de envio de comandos. Os controladores, ao receberem os comandos, ao invés de os encaminharem aos motores imediatamente, o que poderia congestionar a rede de comunicação com os motores em casos de múltiplos acessos concorrentes (semelhante ao que acontece na solução atual), os armazenam nas *hardware interfaces* para que todas as requisições sejam processados de uma só vez na próxima iteração.

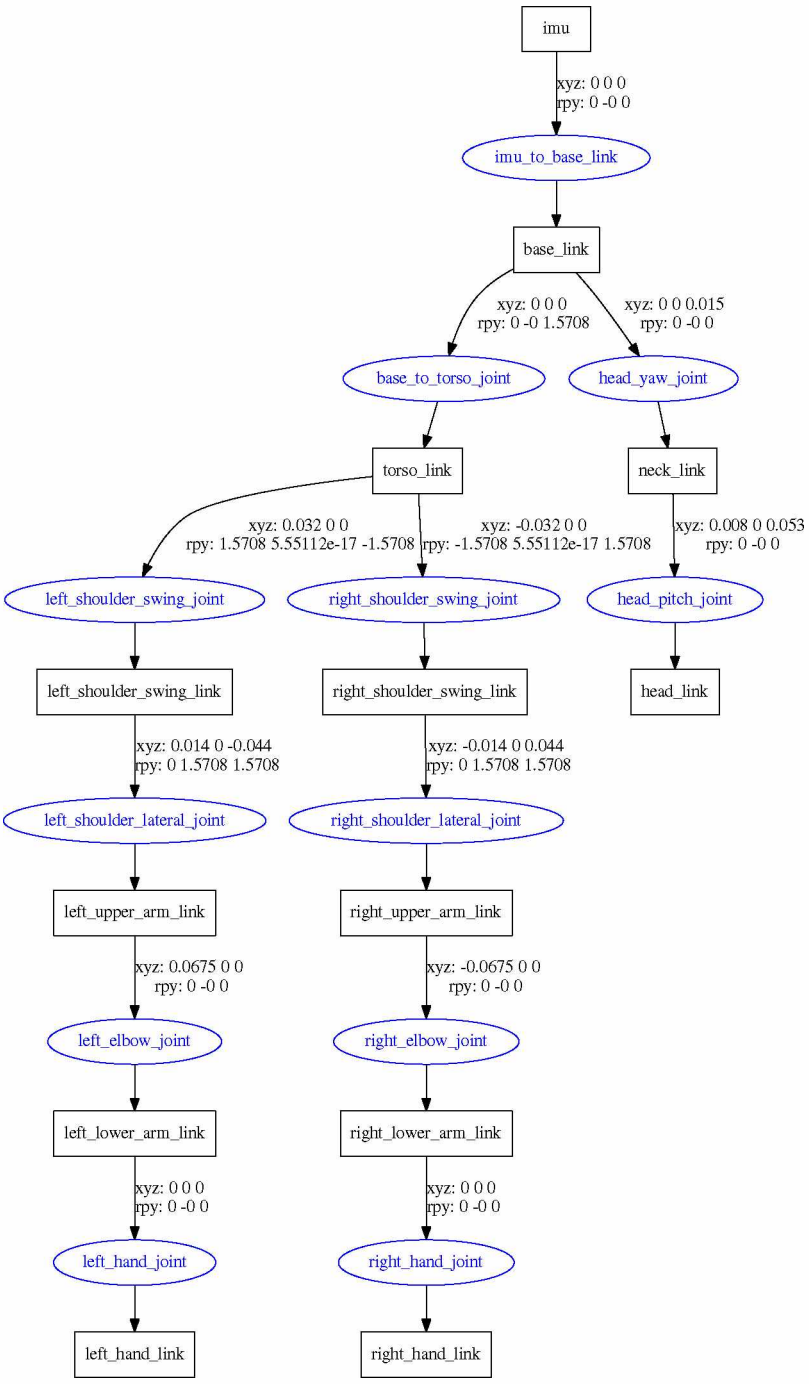
5.2.2 Visualização no *rviz*

Para exemplificar o uso integrado do *ros_control* com o *rviz*, configurou-se o ambiente para visualizar o estado das juntas e da postura do robô em três dimensões. Inicialmente, criou-se um modelo em URDF do tronco do robô humanoide de testes da Figura 17. Reaproveitou-se desenhos das peças e o modelo URDF do pacote *ros-bioloid*[5], que descreve o robô humanoide "Bioloid", cujo tronco é quase idêntico ao robô de teste. Basicamente, retirou-se as pernas, modificou-se a cabeça e adicionou-se a IMU. O Apêndice

[5] <https://github.com/dxydas/ros-bioloid>

C contém os arquivos que geram o modelo URDF do modelo produzido e a Figura 23 mostra o diagrama associado, partindo do *link* que representa a IMU. Nesta imagem, ressalta-se a formação de três ramos, correspondentes aos braço esquerdo, o direito e a cabeça.

Figura 23 – Diagrama do modelo URDF do robô de teste.

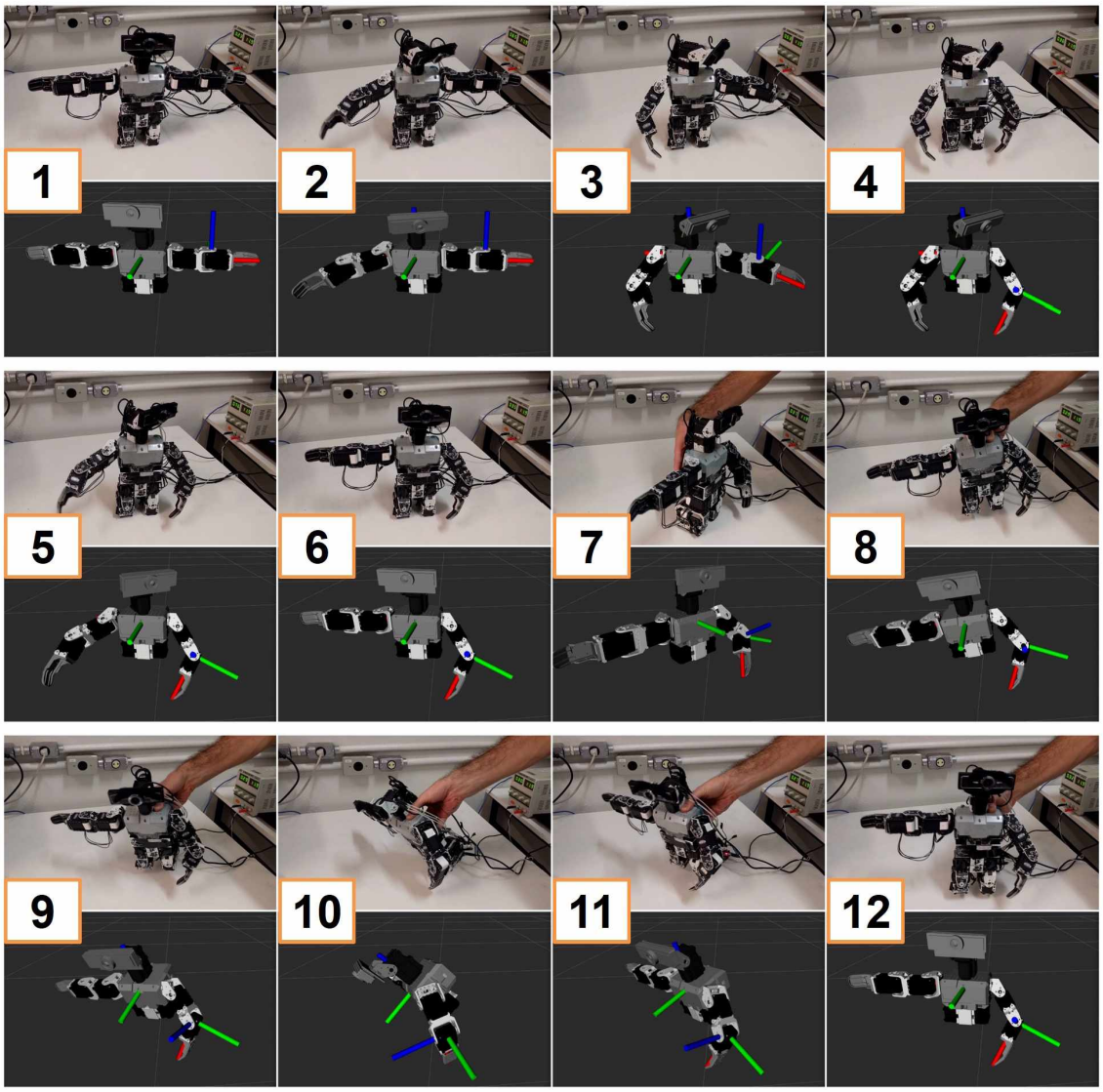


Fonte: Produzido pelo autor.

Para estimar a orientação do robô, utilizou-se dois pacotes ROS complementares: o *phidgets_drivers* e o *imu_filter_madgwick*. O primeiro possui um programa que faz a leitura periódica dos dados do sensor inercial e os publica em tópicos. Já o segundo, lê os dados publicados e aplica o algoritmo de fusão de dados desenvolvido por Madgwick (2010), que processa a orientação estimada do robô com base nos dados do acelerômetro, giroscópio e, opcionalmente, da bússola, e dispõe à *tf*, que aplica as rotações correspondentes à junta que representa a IMU no modelo URDF exibido no *rviz*.

A Figura 24 apresenta uma sequência de quadros selecionados de um vídeo que mostra um robô real em movimento, em paralelo com a tela do *rviz*, que exibe um modelo visual do robô em tempo real.

Figura 24 – Sequência de imagens capturadas durante teste de visualização do estado do robô no *rviz*



Fonte: Produzido pelo autor.

Na primeira metade do experimento, enviou-se comandos para movimentar os braços e a cabeça e, na segunda metade (a partir do sétimo quadro), com o robô fixado na posição que finalizou o movimento anterior, moveu-se o tronco com a mão, rotacionando-o em torno de todos os eixos. Observa-se que a reprodução do estado do robô no *rviz* se aproxima muito bem do estado real do robô, tanto em relação à disposição das juntas quanto à postura do robô. Mesmo quando o robô é rotacionado, a posição de todas as partes do robô refletem o estado real.

Este resultado reforça a eficiência e robustez do pacote *tf*, que constantemente processa a posição e orientação dos eixos de coordenadas especificados e envia ao *rviz*, do sistema de controle desenvolvido, que disponibiliza os dados de posição e velocidade das juntas corretamente, e das ferramentas de processamento de dados da IMU, que calculam a orientação do robô com muita rapidez e qualidade.

Ressalta-se que esta visualização pode ser de grande valia em uma aplicação real, pois, principalmente, facilita encontrar erros de implementação e calibração de forma fácil e rápida. Por exemplo, durante a construção deste experimento, por muitas vezes, ao enviar determinadas posições aos motores, no *rviz*, partes não condizentes do robô se movimentavam, as juntas se moviam em direção oposta ou, simplesmente, nada acontecia. Ao tentar entender os comportamentos, identificavam-se erros de definição de coordenadas no modelo URDF, de não correspondência de identificação das juntas entre o código e modelo, erros de conversão de unidades, falhas de conexão com *hardware*, etc. Além disso, embora não tenha sido demonstrado, é possível utilizar todas os dados de configuração, como o modelo URDF, e acessar as mensagens publicadas nos tópicos por meio de código.

5.3 Desempenho em RTOS

Como apresentado, uma das capacidades do nova biblioteca é ter suporte a sistemas operacionais de tempo real. Neste sentido, levantou-se métricas e gráficos comparativos entre testes realizados no sistema operacional convencional e no modificado, de tempo real.

5.3.1 Preparação do Ambiente

Em ambos os sistemas operacionais, simulou-se uma situação de estresse, ou seja, executou-se os testes em paralelo à execução de outros processos que sobrecarregam o uso de recursos da máquina, como memória e processador. Nesta situação, a concorrência pelo uso dos componentes do computador é elevada, aumentando a frequência de interrupções de processos e mudanças de contexto, dentre outros eventos que contribuem para a diminuição de desempenho do sistema.

A situação de estresse foi gerada através do programa *stress*[6]. Partindo de um estado com praticamente 0% de uso do processador e aproximadamente 40% de uso de memória RAM, configurou-se o *stress* para utilizar 100% dos oito núcleos do processador e sobrecarregar o uso da memória com dois processos que constantemente alocam e desalocam 2GB cada, de forma que o seu uso atingia até aproximadamente 90%.

Nesta situação, realizou-se um teste que analisa os atrasos, ou latências, em ambos os sistemas e o seu impacto no controle de movimento.

5.3.2 Análise de Latência

Uma das principais métricas de desempenho de um RTOS é a latência, que se refere ao atraso entre a ocorrência de um evento e o seu tratamento, em outras palavras, é o tempo entre o momento de disparo de uma interrupção e o início da execução propriamente dita da *thread* que espera por ela (NATIONAL INSTRUMENTS, 2011). No *loop* de controle do sistema desenvolvido, a latência seria o tempo gasto entre o instante esperado para iniciar uma iteração e o momento em que ela realmente se inicia, correspondendo à parcela indicada como "atraso"na Figura 15 da sessão 4.2.

Desse modo, desenvolveu-se um programa, disponível no Apêncice B.4, que dispara *timers* de duração fixa periodicamente e, a cada iteração, registra o atraso ocorrido. Os parâmetros utilizados neste teste são apresentados na Tabela 2.

Tabela 2 – Parâmetros do teste de tempo de resposta

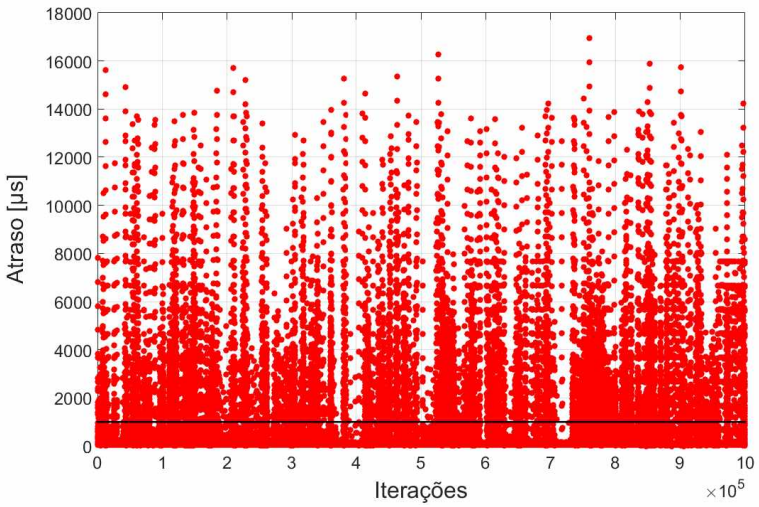
Parâmetro	**Valor**
Período	1ms
Número de Iterações	1.000.000
Frequência	1000Hz
Duração	16min e 40s

Fonte: Produzido pelo Autor.

As Figuras 25 e 26 mostram os atrasos registrados a cada iteração do teste realizado no SO convencional e no de tempo real respectivamente.

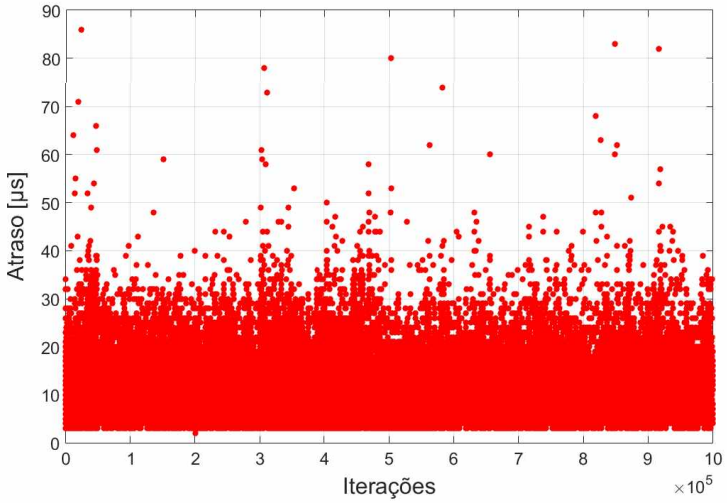
[6] <https://linux.die.net/man/1/stress>

Figura 25 – Gráfico dos atrasos por iteração do teste de latência em SO Convencional



Fonte: Produzido pelo autor

Figura 26 – Gráfico dos atrasos por iteração do teste de latência em RTOS



Fonte: Produzido pelo autor

No teste realizado no Linux Convencional, Figura 25, registrou-se muitos valores acima da marca indicada de $1000\mu s$, que corresponde a 100% do período do laço. Apesar disso, percebe-se que a concentração de pontos se torna menor à medida que se afasta do ponto de latência zero, indicando que a maioria dos atrasos podem estar em níveis aceitáveis. Por outro lado, no RTOS, Figura 26, todos os atrasos foram inferiores a $90\mu s$. A Tabela 3 apresenta detalhes numéricos dos dados coletados.

Tabela 3 – Dados estatísticos do teste de latência

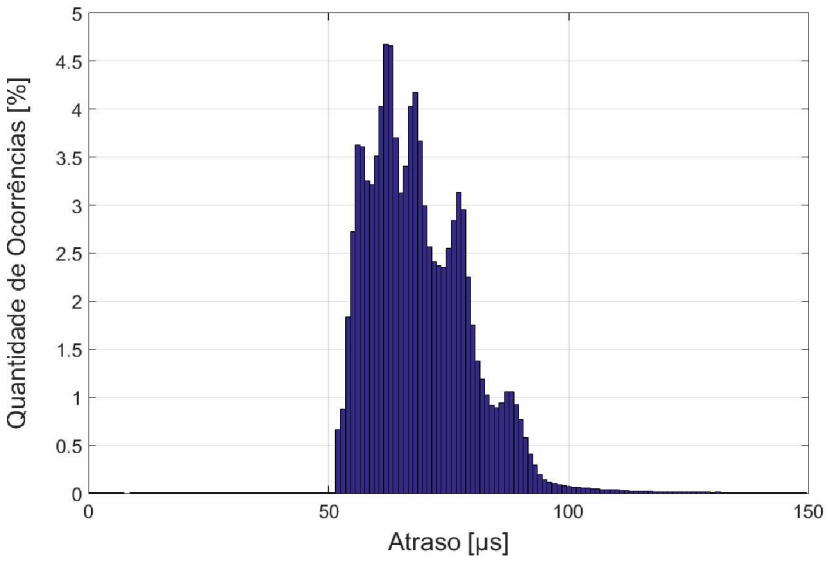
Métrica	SO Convencional	RTOS
Latência Máxima	16949 μs	86 μs
Latência Média	111,84 μs	5,13 μs
Desvio Padrão	476,04 μs	2,373 μs
Moda	62 μs	4 μs

Fonte: Produzido pelo Autor.

Constata-se que, no SO convencional, embora tenha-se alcançado uma média relativamente baixa de pouco mais de 100μs, a latência máxima chegou a quase 17 vezes o período do laço. Em um processo de amostragem de sinal, por exemplo, isso significaria que até 17 amostras seguidas poderiam ser perdidas. Nota-se também que o desvio padrão está 4 vezes a latência média, ressaltando a alta incerteza no cumprimento de prazos. Em compensação, no Linux de tempo real, a latência máxima registrada foi inferior a 100μs, apresentando média e desvio padrão abaixo de 10 μs. Este resultado indica que, com esta configuração, é possível cumprir prazos a frequências de até aproximadamente 10kHz.

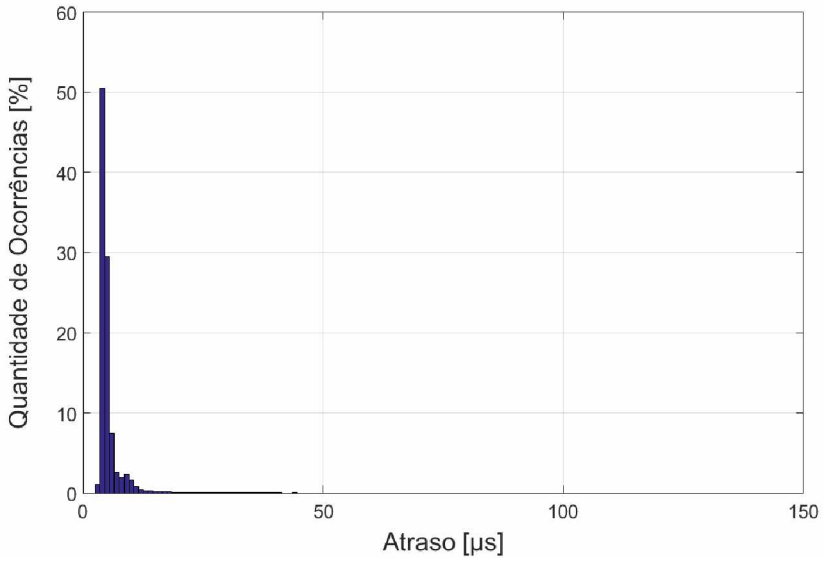
Para ilustrar melhor o comportamento de cada sistema, as Figuras 27 e 28 apresentam um histograma das latências de até 150μs para cada configuração e as Figuras 29 e 30 mostram as distribuições acumuladas correspondentes.

Figura 27 – Histograma de latências de até 150μs em SO Convencional



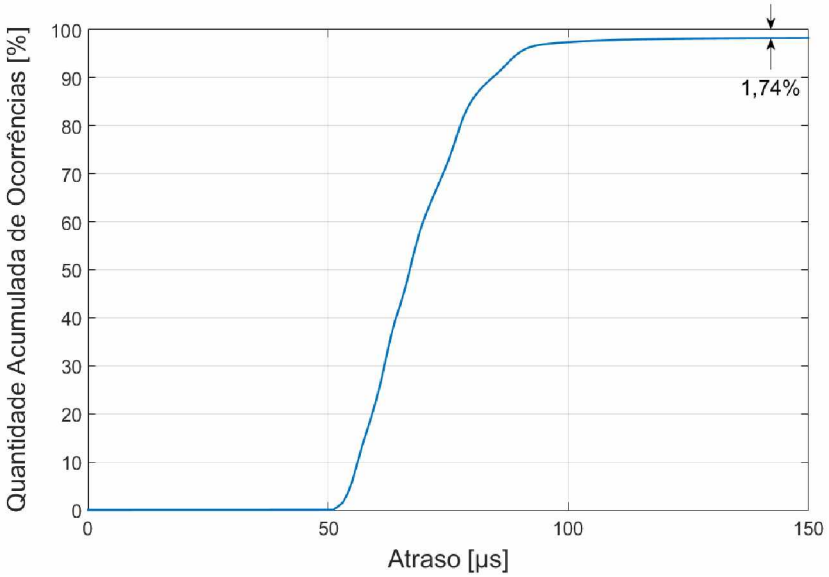
Fonte: Produzido pelo autor

Figura 28 – Histograma de latências de até 150μs em RTOS



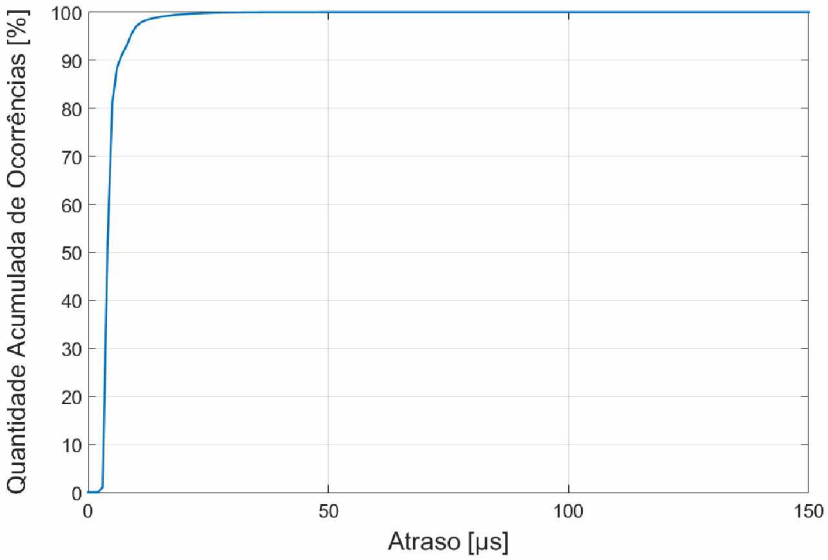
Fonte: Produzido pelo autor

Figura 29 – Distribuição acumulada das latências de até 150μs em SO Convencional



Fonte: Produzido pelo autor

Figura 30 – Distribuição acumulada das latências de até 150μs em RTOS



Fonte: Produzido pelo autor

Pelas Figuras 27 e 29, verifica-se que, para o teste no SO Convencional, embora a média dos atrasos tenha ficado próxima de 100μs, mais de 95% dos casos ficaram abaixo deste valor, concentrando-se em torno de aproximadamente 60μs (vide moda apresentada na Tabela 3). Porém, na Figura 29, nota-se que 1,74% dos atrasos foram superiores a 150μs e, embora não mostrado, 1,09% das latências foram superiores a 1ms, o período de amostragem.

Em contrapartida, observa-se uma diferença significativa com os gráficos correspondentes do do SO de tempo real. Nas Figuras 28 e 30, verifica-se que quase que 100% dos atrasos foram inferiores a 25μs, concentrando-se em torno de 5μs (vide média apresentada na Tabela 3).

5.3.3 Impacto sobre Controle de Posição das Juntas

Basicamente, propõe-se comparar o desempenho do controle das juntas em situação de estresse. Este teste foi executado em 3 ambientes:

1. Sistema de controle atual[7] + Linux Convencional
2. *DxlRobotHW* + Linux Convencional
3. *DxlRobotHW* + Linux de Tempo Real

[7] A solução atual, originalmente, não contempla leitura dos motores. Portanto, foi necessário fazer algumas adaptações.

Assim, enviou-se comandos de posição e velocidade ao motor MX-64 da estrutura 2 da Figura 17 e registrou-se os valores de posição e velocidade das juntas, além dos tempos de envio. O sistema com o motor com MX-64 foi preferido para este teste, pois apresenta melhor desempenho de controle de posição e velocidade em relação aos motores AX-12 e permite aplicação de maior carga, o que torna o sistema mais sensível a erros de posição. Para todos os testes configurou-se os seguintes comandos de posição, Equação 5.1, e velocidade (Equação 5.2), obtida através da primeira derivada da posição:

$$q(t) = \frac{1}{2}[sen(2t) - sin(4t)] \tag{5.1}$$

$$\dot{q}(t) = cos(2t) - 2cos(4t) \tag{5.2}$$

Lembra-se que o sistema é discreto e, portanto, os tempos t são definidos pela Equação 4.1. A frequência de envio foi configurada para 320Hz, valor alto para este sistema, visto que se observou que o atuador utilizado opera bem em até 333Hz aproximadamente. A escolha de uma frequência alta foi proposital para que os atrasos sejam mais frequentes e os seus possíveis impactos no controle de movimento sejam mais evidentes.

Ressalta-se que, em situação de baixa carga de processamento e frequências mais baixas, todos os sistemas funcionam bem visualmente. Entretanto, na situação proposta, espera-se observar desvios indesejados na trajetória ocasionados por latências significativas.

Em linhas gerais, ao observar os experimentos, notou-se as seguintes características:

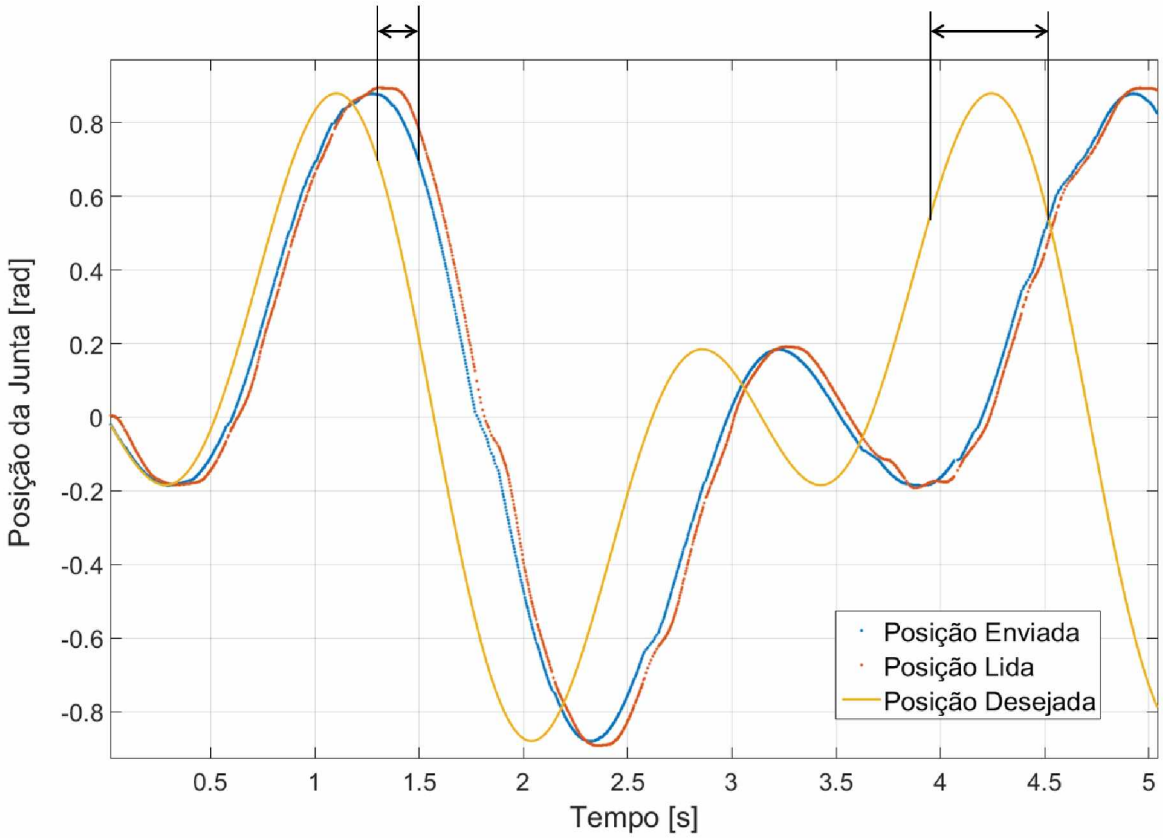
1. **Sistema de controle atual + Linux Convencional:** Nitidamente, o sistema não cumpre as trajetórias desejadas. A barra realizou muitos movimentos bruscos acompanhados de ruídos elevados;
2. **DxlRobotHW + Linux Convencional:** Foi possível notar eventuais movimentos bruscos e ruídos de baixa intensidade. Porém, na maior parte do tempo o movimento foi suave;
3. **DxlRobotHW + Linux de Tempo Real:** Em relação aos testes anteriores, este apresentou o melhor desempenho. Em nenhum momento observou-se acelerações bruscas nem ruídos fora da normalidade.

Ao analisar os dados coletados e visualizá-los em gráficos, estas observações preliminares podem ser explicadas.

Sistema de Controle Atual + Linux Convencional

A Figura 31 apresenta o comportamento da posição da junta nos primeiros cinco segundos do experimento, em que os estalos e movimentos bruscos foram intensos.

Figura 31 – Gráfico de posição da junta nos primeiros 5 segundos do experimento com o sistema de controle antigo

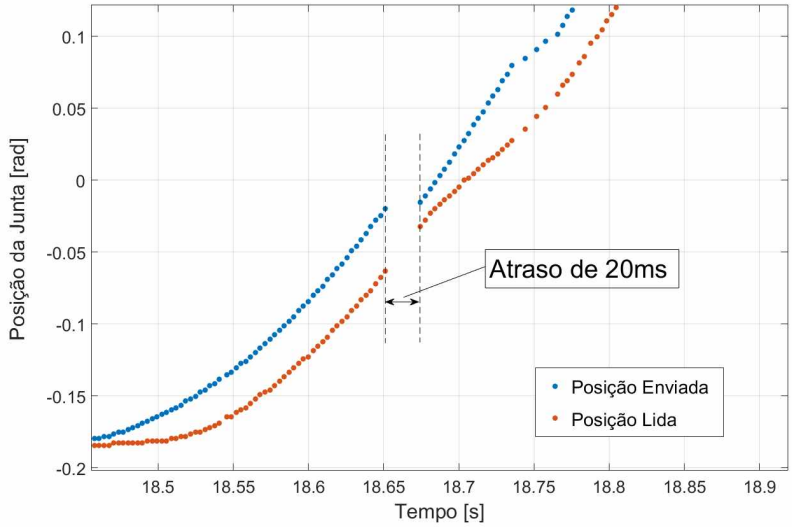


Fonte: Produzido pelo autor

Primeiramente, é evidente que a posição enviada aos motores não condiz com a posição desejada. Além de estar atrasada no tempo, não segue o formato desejado, vide oscilações de posição significativas observadas, podendo ser um indicativo de atrasos de envio de comandos. De fato, ao analisar os atrasos ocorridos ao longo deste experimento, como esperado, obteve-se um perfil semelhante ao do teste de tempo de resposta no SO Convencional, Figura 25.

Além disso, observou-se que a duração do experimento se estendeu além dos 30 segundos previstos. Isso indica que os atrasos podem estar sendo absorvidos no processo de envio e recebimento de dados. Ao analisar o gráfico de posição enviada e lida em um ponto em que ocorreu um atraso significativo é possível entender o que acontece. A Figura 32 mostra um destes momentos.

Figura 32 – Gráfico de posição da junta em momento de atraso significativo no experimento com o sistema de controle antigo



Fonte: Produzido pelo autor

Primeiramente, observa-se que a curva da "posição enviada"após a ocorrência do atraso marcado se ligaria à anterior se movesse 20ms à esquerda, ou seja, se não houvesse o atraso. Isso acontece, pois o algoritmo de envio de poses do sistema da EDROM, como apresentado na sessão 2.2.1, faz o envio sequencial das poses, orientando-se pelo número da iteração. Este comportamento é evidenciado quando se obtém a curva desejada ao plotar o sinal em função das iterações igualmente espaçadas no eixo X. Desse modo, implicitamente, esta estratégia leva em conta que os comandos estão sendo enviados nos momentos planejados.

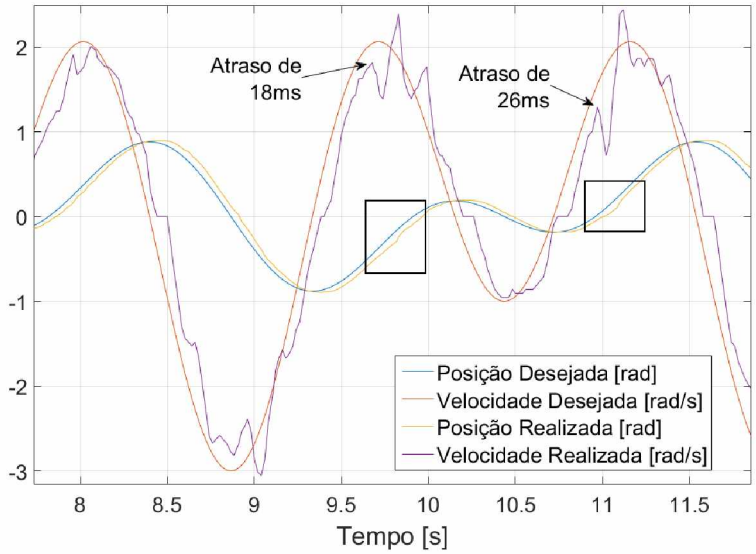
Além disso, verificou-se que a estratégia de temporização de envio deste sistema, apresentada pela Equação 2.2, não considera o tempo de atraso de inicialização das iterações, vide Figura 15. Neste algoritmo, o ponto de referência para a definição do tempo de início da iteração é obtido no momento imediatamente anterior à operação principal, diferentemente do *ControlLoop*, que define o tempo da próxima iteração com base no momento planejado de início da iteração corrente. Assim, embora o algoritmo não perceba, as parcelas de atraso vão se acumulando, somando quase 5 segundos neste exemplo.

DxlRobotHW + Linux Convencional

Em resposta aos problemas identificados no teste anterior, o *Joint Trajectory Controller* do *ros_control*, utilizado para realizar os testes com o *DxlRobotHW*, a cada iteração, não envia os comandos (equivalente às poses) na sequência definida, e sim, amostragem do sinal de referência em função do tempo real de envio dos comandos. Como introduzido na sessão 3.2.2, nos pontos em que a trajetória não é explicitamente definida, realiza-se interpolação *spline* entre os pontos adjacentes.

Assim, o teste com o *DxlRobotHW* no SO convencional, embora tenha apresentado um perfil de latências semelhante ao do experimento anterior, enviou comandos de posição em correspondência à curva de posição desejada, cumprindo a execução da trajetória na duração esperada de 30 segundos. Entretanto, identificou-se oscilações indesejadas no perfil de posição e velocidade da junta nos momentos em que ocorreram atrasos significativos. A Figura 33 destaca o comportamento das juntas em dois destes momentos.

Figura 33 – Gráfico de posição e velocidade da junta em momento de oscilação indesejada no experimento com o *DxlRobotHW* em SO Convencional



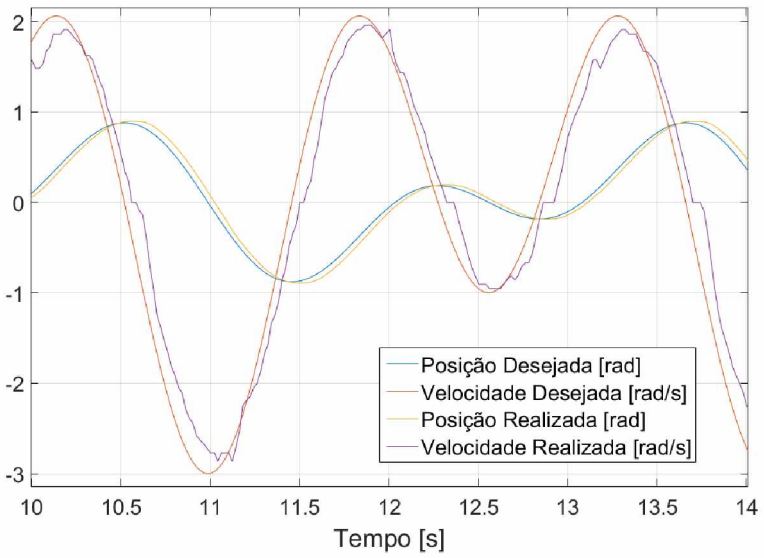
Fonte: Produzido pelo autor

Observa-se que, em ambos os casos, houveram mudanças bruscas de velocidade, refletindo em oscilações perceptíveis na posição. Isso ocorre, pois o sistema de controle embutido nos motores *Dynamixel*, ao receber um comando de posição e velocidade, tenta enviar o motor até o repouso na posição desejada com velocidade média de trajeto igual à especificada. Portanto, na ocorrência de grandes atrasos, o tempo sem comunicação com o motor é suficiente para que o mesmo atinja o estágio de desaceleração para repousar na última posição desejada.

DxlRobotHW + Linux de Tempo Real

No Linux de tempo real, foram registrados apenas nove atrasos, sendo $0,32ms$ em média e máximo de $0,85ms$. A Figura 34 mostra o comportamento da junta no momento de maior atraso.

Figura 34 – Gráfico de posição e velocidade da junta em momento de maior atraso no experimento com o *DxlRobotHW* em RTOS



Fonte: Produzido pelo autor

O maior atraso registrado ocorreu por volta de $12s$ no gráfico da Figura 34. Neste ponto, observa-se um pequeno pico na velocidade, mas nenhuma oscilação evidente na posição. Na realidade, ao longo de toda a curva, observa-se pequenas oscilações na velocidade. Entretanto, elas são presentes mesmo quando se realiza o mesmo teste sem carga de processamento paralelo, sendo, portanto, uma característica dos motores. Da mesma forma, os desvios de posição observados, principalmente nos momentos de inversão de movimento, são comuns, devido à inércia do sistema e das limitações do controlador de posição embutido.

6 Considerações Finais

Produziu-se uma estrutura de *software* genérica para gerenciar o controle de movimento de robôs equipados com motores *Dynamixel* por meio da utilização do *ros_control*. Em comparação com sistema de controle de movimento utilizado pela EDROM atualmente, alcançou-se diversas melhorias em sintonia com as tecnologias e princípios encontrados na literatura. Demonstrou-se, por meio de testes e análises, que a solução atual possui uma estrutura modular e flexível que possibilita o uso integrado com ferramentas do ROS e oferece suporte a sistemas operacionais de tempo real.

A divisão em componentes independentes e personalizáveis do *ros_control* possibilitou a reutilização de código e proporcionou um direcionamento claro no desenvolvimento, tornando-o mais ágil e com grande potencial de crescimento. Estes benefícios ficaram claros nos testes de integração com o ROS, em que a conclusão do desenvolvimento de dois componentes, o *DxlRobotHW* e o *ControlLoop*, foi suficiente para explorar mais de três tipos de controladores genéricos e utilizar ferramentas básicas de visualização e troca de mensagens do ROS. Depois, com um pouco mais de envolvimento com o ROS, conseguiu-se estimar o estado do robô com precisão e visualizá-lo em 3D no *rviz*.

Em relação ao desempenho de tempo real, os testes demonstraram que as modificações no Linux foram feitas com sucesso, registrando-se latências abaixo de $100\mu s$. Foi mostrado que o RTOS contribui significativamente na qualidade de controle de posição dos motores, por meio da minimização de oscilações indesejadas causadas por atrasos de envio de comandos. Além disso, este teste evidenciou problemas na estratégia adotada pela EDROM para enviar trajetórias de posição e velocidade às juntas.

Embora o sistema desenvolvido não tenha sido testado diretamente no controle do movimento de andar de robôs humanoides, a motivação para este trabalho, conclui-se que a nova ferramenta oferece infraestrutura adequada para aplicar novas estratégias de controle, garante mais segurança contra o impacto das latências no processo comunicação com os atuadores e abre portas para a integração com novas ferramentas.

Trabalhos Futuros

De imediato, sugere-se as seguintes melhorias aos componentes desenvolvidos:

- Transferir as configurações do *DxlRobotHW* (como de definição dos *JointIDs*) e do *ControlLoop* (como frequência, tipo de escalonador e prioridade) para o *parameter server* do ROS;

- Prover uma ROS API para o *ControlLoop*, tornando-se possível chamar suas funções e captar as ocorrências do método *OnMiss* por meio de tópicos;
- Agregar novas interfaces de *hardware* ao *DxlRobotHW*, como a *Velocity Interface* e *Effort Interface* do *ros_control*. Também seria possível implementar novas interfaces que englobem outras funcionalidades do motor, como de leitura de temperatura e configuração das constantes PID do controlador de posição;
- Estender o suporte do *DxlRobotHW* para motores que utilizem o protocolo 2.0 de comunicação da *Dynamixel SDK*.

Em relação à aplicação da ferramenta ao contexto da EDROM, sugere-se definir uma estrutura de controladores para o robô humanoide, avaliando-se o que pode ser reaproveitado do pacote *ros_controllers* e o que deve ser desenvolvido. Acredita-se que os controladores explorados são suficientes para controlar a cabeça e os braços do robô. Para realizar o controle do movimento de andar, recomenda-se investigar estratégias de controle adequadas para a aplicação, como a proposta por Schwarz e Behnke (2013), introduzindo novos *Controllers* à biblioteca. Além disso, é possível explorar outras ferramentas do ROS para contemplar outras partes da aplicação, como processamento de imagens, navegação e inteligência artificial, que podem ser de grande valia ao projeto.

Em relação ao suporte a RTOS, recomenda-se realizar testes com robôs em ambiente de trabalho para avaliar o seu desempenho com mais atuadores e com carga de processamento paralelo usual. Também seria válido investigar o impacto das limitações do *ros_control*, que restringe o *loop* de controle a uma única *thread* de tempo real e não oferece recursos explícitos para comunicação entre controladores. Caso estas limitações sejam significativas, recomenda-se explorar ferramentas mais avançadas, como o OROCOS (LAGES; IORIS; SANTINI, 2014; PEEKEMA; RENJEWSKI; HURST, 2013). Também vale testar o sistema antigo da EDROM em Linux de tempo real para analisar, separadamente, o impacto da redução dos atrasos atingido com o RTOS e do algoritmo de envio de comandos realizado *Joint Trajectory Controller*.

Além disso, é possível fazer muitos estudos sobre o comportamento dos motores *Dynamixel*. Por exemplo, pode-se fazer uma estimação do tempo de resposta dos motores, do atraso máximo permissível para o que não ocorra desvios de posição indesejados e de frequência máxima de comunicação para diferentes comandos, modelos de motores e configurações mecânicas.

Visto as inúmeras formas de exploração das ferramentas propostas, considera-se que a maior contribuição deste trabalho consistiu em abrir o leque de possibilidades de soluções para controle para robôs. Acredita-se que um ambiente que favorece o contato com muitas oportunidades abre espaço para a criatividade e, consequentemente, soluções inovadoras.

Referências

ALLGEUER, P.; SCHWARZ, M.; PASTRANA, J.; SCHUELLER, S.; MISSURA, M.; BEHNKE, S. A ROS-based software framework for the NimbRo-OP humanoid open platform. In: *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE-RAS Int. Conference on Humanoid Robots, Atlanta, USA*. [s.n.], 2013. Disponível em: <http://www.is.uni-bonn.de/papers/HSR13_Allgeuer_ROS_NimbRo-OP.pdf>. 25

AUTONOMOUS INTELLIGENT SYSTEMS GROUP, RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITäT BONN. *nimbro-op-ros: NimbRo-OP ROS software release*. 2017. Original-date: 2013-10-02T12:23:51Z. Disponível em: <https://github.com/NimbRo/nimbro-op-ros>. 25

CHUNG, W.; FU, L.-C.; HSU, S.-H. Motion control. In: SICILIANO, B.; KHATIB, O. (Ed.). *Springer Handbook of Robotics*. 1. ed. Berlin: Springer, 2008. cap. 6, p. 133–157. 20, 21

DUTTA, A. *Robotic systems: applications, control and programming*. Rijeka: InTech, 2012. OCLC: 908264538. ISBN 978-953-307-941-7. 24

ELKADY, A.; SOBH, T. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, v. 2012, p. 1–15, 2012. ISSN 1687-9600, 1687-9619. Disponível em: <http://www.hindawi.com/journals/jr/2012/959013/>. 24

FOOTE, T. tf: The transform library. In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. IEEE, 2013. p. 1–6. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6556373/>. 32

IñIGO-BLASCO, P.; RIO, F. Diaz-del; ROMERO-TERNERO, M. C.; CAGIGAS-MUñIZ, D.; VICENTE-DIAZ, S. Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, v. 60, n. 6, p. 803–821, jun. 2012. ISSN 09218890. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0921889012000322>. 24

KAJITA, S.; ESPIAU, B. Legged robots. In: SICILIANO, B.; KHATIB, O. (Ed.). *Springer Handbook of Robotics*. 1. ed. Berlin: Springer, 2008. cap. 16, p. 361–387. 21

KERRISK, M. *sched(7) - overview of CPU scheduling [Linux User's Manual]*. [S.l.], 2016. Disponível em: <http://man7.org/linux/man-pages/man7/sched.7.html>. 28

KUNZE, L.; ROEHM, T.; BEETZ, M. Towards semantic robot description languages. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011. p. 5589–5595. Disponível em: <http://ieeexplore.ieee.org/abstract/document/5980170/>. 31

LAGES, W. F.; IORIS, D.; SANTINI, D. C. An architecture for controlling the barrett wam robot using ros and orocos. In: VDE. *ISR/Robotik 2014; 41st International Symposium on Robotics;*. [S.l.], 2014. p. 1–8. 24, 26, 66

LENTIN, J. *Mastering ROS for robotics programming: design, build, and simulate complex robots using Robot Operating System and master its out-of-the-box functionalities*. 1. ed.

Birmingham Mumbai: Packt Publishing, 2015. (Community experience distilled). OCLC: 947085604. ISBN 978-1-78355-179-8. 21

LENTIN, J. *Mastering ROS for robotics programming: design, build, and simulate complex robots using Robot Operating System and master its out-of-the-box functionalities*. Birmingham Mumbai: Packt Publishing, 2015. (Community experience distilled). OCLC: 947085604. ISBN 978-1-78355-179-8. 47

MADGWICK, S. An efficient orientation filter for inertial and inertial/magnetic sensor arrays. *Report x-io and University of Bristol (UK)*, v. 25, abr. 2010. Disponível em: <https://www.samba.org/tridge/UAV/madgwick_internal_report.pdf>. 53

MAGYAR, B. *diff_drive_controller - ROS Wiki*. 2017. Disponível em: <http://wiki.ros.org/diff_drive_controller>. 38

MEEUSSEN, W. *ros_control - ROS Wiki*. 2017. Disponível em: <http://wiki.ros.org/ros_control>. 33

MOVEIT! RViz Plugin Tutorial. 2017. Disponível em: <http://docs.ros.org/hydro/api/moveit_ros_visualization/html/doc/tutorial.html>. 32

NATIONAL INSTRUMENTS. *O que é a Tecnologia de Tempo Real? - National Instruments*. 2011. Disponível em: <http://www.ni.com/white-paper/3938/pt/>. 26, 55

OPEN SOURCE ROBOTICS FOUNDATION. *ROS/Concepts - ROS Wiki*. 2014. Disponível em: <http://wiki.ros.org/ROS/Concepts>. 30, 31

OPEN SOURCE ROBOTICS FOUNDATION. *ROS.org | About ROS*. 2017. Disponível em: <http://www.ros.org/history/>. 26, 29

OPEN SOURCE ROBOTICS FOUNDATION. *ROS.org | Core Components*. 2017. Disponível em: <http://www.ros.org/core-components/>. 32, 33

OPEN SOURCE ROBOTICS FOUNDATION. *urdf/Tutorials/Create your own urdf file - ROS Wiki*. 2017. Disponível em: <http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file>. 31

PEEKEMA, A.; RENJEWSKI, D.; HURST, J. Open-Source Real-Time Robot Operation and Control System for Highly Dynamic, Modular Machines. In: . ASME, 2013. p. V07AT10A063. ISBN 978-0-7918-5596-6. Disponível em: <http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?doi=10.1115/DETC2013-12493>. 26, 66

PUHAN, J. *Operating systems, embedded systems and real-time systems*. Ljubljana: Založba FE, 2015. OCLC: 905310946. ISBN 978-961-243-275-1. Disponível em: <http://fides.fe.uni-lj.si/~janezp/operating_systems,_embedded_systems_and_real-time_systems.pdf>. 27, 28

QUIGLEY, M.; CONLEY, K.; GERKEY, B.; FAUST, J.; FOOTE, T.; LEIBS, J.; WHEELER, R.; NG, A. Y. ROS: an open-source Robot Operating System. In: *ICRA workshop on open source software*. Kobe, 2009. v. 3, p. 5. Disponível em: <https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf>. 25, 29

RAMMIG, F.; DITZE, M.; JANACIK, P.; HEIMFARTH, T.; KERSTAN, T.; OBERTHUER, S.; STAHL, K. Basic concepts of real time operating systems. In: *Hardware-dependent Software*. Springer, 2009. p. 15–45. Disponível em: <http://link.springer.com/chapter/10.1007/978-1-4020-9436-1_2>. 42

ROBOTIS. *MX-64T / MX-64R / MX-64AT / MX-64AR - ROBOTIS e-Manual*. [S.l.], 2017. Disponível em: <http://support.robotis.com/en/product/actuator/dynamixel/mx_series/mx-64at_ar.htm>. 20, 21, 50

ROBOTIS. *ROBOTIS-Documents: ROBOTIS Documents (Manuals & Tutorials)*. ROBOTIS GIT, 2017. Original-date: 2016-04-11T09:52:18Z. Disponível em: <https://github.com/ROBOTIS-GIT/ROBOTIS-Documents>. 25

ROBOTIS. *ROBOTIS-Framework: ROS packages for the ROBOTIS Platform Framework*. ROBOTIS GIT, 2017. Original-date: 2016-03-04T11:57:16Z. Disponível em: <https://github.com/ROBOTIS-GIT/ROBOTIS-Framework>. 25

SCHWARZ, M.; BEHNKE, S. Compliant robot behavior using servo actuator models identified by iterative learning control. In: *Robot Soccer World Cup*. Springer, 2013. p. 207–218. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-662-44468-9_19>. 21, 66

SICILIANO, B.; SCIAVICCO, L.; VILLANI, L.; ORIOLO, G. *Robotics: Modelling, Planning and Control*. 1. ed. London: Springer London, 2009. (Advanced Textbooks in Control and Signal Processing). DOI: 10.1007/978-1-84628-642-1. ISBN 978-1-84628-641-4 978-1-84628-642-1. 19, 20, 21, 24

SIRIO, G. D. *ChibiOS free embedded RTOS - RTOS Concepts*. 2017. Disponível em: <http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts>. 27

SPONG, M. W.; HUTCHINSON, S.; VIDYASAGAR, M. *Robot modeling and control*. 1. ed. Hoboken, NJ: Wiley, 2006. OCLC: 266014409. ISBN 978-0-471-64990-8. 21

TANENBAUM, A. S.; BOS, H. *Modern operating systems*. [S.l.: s.n.], 2015. OCLC: 890473514. ISBN 978-0-13-359162-0. 26, 27, 28

THE LINUX FOUNDATION. *Real-Time Linux [Linux Foundation Wiki]*. 2017. Disponível em: <https://wiki.linuxfoundation.org/realtime>. 73

TSOUROUKDISSIAN, A. R. *[ROSCon 2014] Adolfo Rodriguez Tsouroukdissian: ros_control: An overview (HD)*. 2014. Disponível em: <https://vimeo.com/107507546>. 33, 34, 36

TSOUROUKDISSIAN, A. R. *joint_trajectory_controller - ROS Wiki*. 2017. Disponível em: <http://wiki.ros.org/joint_trajectory_controller>. 38

Apêndices

APÊNDICE A – Configuração do Sistema Operacional de Tempo Real

O procedimento adotado para tornar o Linux como de tempo real foi baseado, principalmente, na documentação apresentada em The Linux Foundation (2017). Nestas fontes, encontra-se alguns tutoriais sobre como configurar o SO e de como utilizar as funcionalidades alcançadas com a modificação. Porém, à primeira vista, o conteúdo pode ser confuso, pois aborda detalhes de que não são estritamente necessários para obter resultados satisfatórios ou, muitas vezes, não explicam o funcionamento dos comandos utilizados. Assim, apresenta-se o procedimento utilizado com comentários.

A.1 Preparação do Sistema Operacional

Basicamente, o procedimento consiste em aplicar o pacote de mudanças PREEMPT_RT ao código-fonte do *kernel* do Linux. Segue a sequência de passos:

1. **Baixar o código fonte do *kernel* desejado e do *patch* correspondente**

   Os códigos-fonte dos *kernels* podem ser encontrados em <https://www.kernel.org/pub/linux/kernel/> e os *patches* em <https://www.kernel.org/pub/linux/kernel/projects/rt/>. Recomenda-se baixar o código-fonte de versão igual ou próxima da que já estiver em uso e, de preferência, que esteja marcada como *long-term* ou *stable* no *site* oficial <https://www.kernel.org/>. O *patch* deve ser exatamente da mesma versão escolhida. Para descobrir a versão do *kernel* instalado na máquina basta executar o seguinte comando no terminal:

```
1 $ uname -r
```

2. **Aplicar o pacote de mudanças**

   Descompacte os arquivos baixados, mova o *patch* descompactado para a pasta raiz do código-fonte do *kernel* e execute o comando:

```
1 $ patch -p1 NomeDoArquivo
```

   Este procedimento aplica mudanças em pontos específicos do código-fonte do *kernel* para torná-lo de tempo real. No *link* <https://wiki.linuxfoundation.org/realtime/

documentation/technical_details/start> encontra-se informações técnicas sobre as principais modificações.

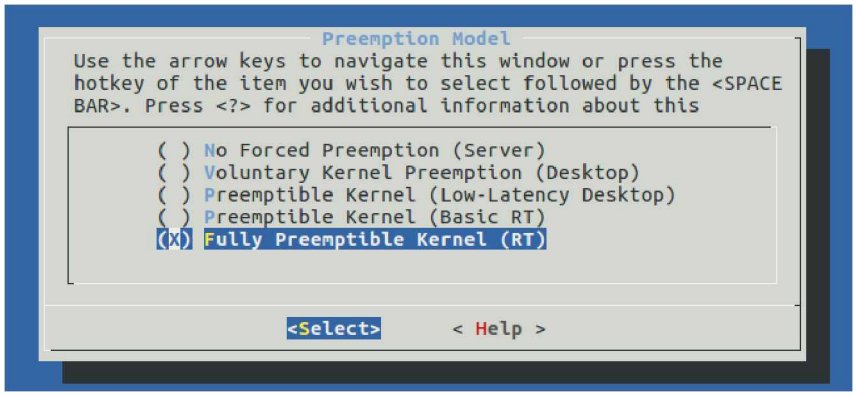
3. **Configurar os parâmetros de compilação do novo *kernel***

   O *kernel* oferece uma infinidade de parâmetros de configuração no momento da compilação, porém, apenas alguns são importantes para atingir o objetivo proposto. Desse modo, recomenda-se fazer as mudanças necessárias a partir das configurações do *kernel* instalado. Para isso, faz-se uma cópia do arquivo de configurações do *kernel* em execução e abre-se o menu de edição de parâmetros.

```
1 $ cp /boot/config-`uname -r`* .config
2 $ make menuconfig
```

Aparecerá uma tela com todas as configurações organizadas em conjuntos e listas. A configuração essencial a ser modificada é a de modelo de preempção. Deve-se selecionar o modelo totalmente preemptivo, como mostra a Figura 35 (ressalta-se que a tela pode variar em função da versão do *kernel*).

Figura 35 – Tela de configuração do modelo de preempção do *kernel* do Linux



Fonte: Produzido pelo autor

Com esta modificação, o arquivo *.config* resultante deve constar a definição da constante CONFIG_PREEMPT_RT_FULL. ??) apresenta outras opções que podem ser modificadas para reduzir a latência do sistema, mas que não são estritamente necessárias. A única mudança adicional feita nos testes é a configuração do *Timer* de alta resolução, ajustando-o para 1kHz.

4. **Compilar e instalar o *kernel***

   Finalmente, basta executar os comandos de compilação (etapa mais demorada, que pode levar várias horas), instalação e atualização das configurações de *boot* respectivamente:

```
$ make -j 8
$ sudo make modules_install install
$ sudo update-grub2
```

Na próxima inicialização do sistema, nas opções avançadas da tela de *boot*, deve-se encontrar o novo *kernel* como alternativa.

A.2 Configuração de Aplicações de Tempo Real

Configurar a prioridade e o escalonador de uma *thread* e usufruir das funcionalidades do RTOS, basta utilizar a função *pthread_setschedparam* da biblioteca padrão POSIX *pthread*. Além disso, para evitar a ocorrência de atrasos por conta de alocação e desalocação automática de memória RAM para dividir o uso com outros processos, também recomenda-se utilizar a função *mlockall* e *munlockall* para impedir que o SO não retire os dados alocados pela *thread* de tempo real da memória. O código a seguir apresenta como isso é feito nos testes de tempo real deste trabalho.

```
mlockall(MCL_FUTURE);

struct sched_param param;
param.__sched_priority = 31;

pthread_setschedparam(pthread_self(), SCHED_RR, &param);

/* YOUR CODE */

munlockall();
```

As funções *munlockall* e *pthread_setschedparam* requerem permissão especial para serem utilizadas. Assim, é necessário rodar o programa que as utilizam como super usuário ("sudo") ou atribuir as capacidades CAP_SYS_NICE e CAP_IPC_LOCK respectivamente, por meio do comando (sem as chaves do último argumento):

```
$ sudo setcap CAP_SYS_NICE,CAP_IPC_LOC=+ep {nome_do_programa}
```

APÊNDICE B – Código Fonte

Todo o código-fonte e arquivos de configuração deste trabalho estão disponíveis no endereço <https://github.com/awtreth/tcc>. Visto que repositório soma centenas de linhas de código, apresenta-se apenas os códigos mais relevantes deste trabalho.

B.1 DxlRobotHW

code/dxl_robot_hw/include/DxlRobotHW.h

```
#include <dynamixel_sdk.h>
#include <ihardware.h>//control_loop package (FIXME: install control_loop)
#include <hardware_interface/robot_hw.h>
#include <hardware_interface/joint_state_interface.h>
#include <hardware_interface/joint_command_interface.h>
#include <hardware_interface/posvel_command_interface.h>
#include <map>
#include <string>
#include <dxl_interface/dxl_model_spec.h>
#include <utility>
#include <realtime_tools/realtime_buffer.h>

class JointID{
public:
    std::string name;
    int id;
    double reference = 0;

    JointID(){}
    JointID(std::string _name, int _id, double _reference, int _direction){
        name = _name;
        id = _id;
        reference = _reference;
        setDirection(_direction);
    }

    int getDirection() const{
        return direction;
    }

    void setDirection(int value){
        direction = (value >= 0)?1:-1;
    }

private:
    int direction = 1;
};


class DxlRobotHW : public hardware_interface::RobotHW{

private:

```

```
44      struct DxlInfo {
45  //          std::string jointName;
46  //          int id;
47  //          double posRef = 0;
48
49          JointID jointID;
50
51          dxl_interface::ModelSpec spec;
52
53          double pos;
54          double vel;
55          double eff;
56
57          double posCmd = 0;
58          double velCmd = 1;
59
60          uint16_t posCmd_dxl;
61          uint16_t posVelCmd_dxl[2];
62
63
64          DxlInfo(JointID _jointID){
65              jointID = _jointID;
66          }
67
68
69          DxlInfo(JointID _jointID, dxl_interface::ModelSpec _spec){
70              jointID = _jointID;
71              spec = _spec;
72          }
73
74          DxlInfo(){}
75      };
76
77      dynamixel::PortHandler* portHandler_;
78      dynamixel::PacketHandler* packetHandler_;
79
80      std::vector<DxlInfo> dxlInfos;
81      std::map<std::string,size_t> dxlNameIdxMap;
82
83      hardware_interface::JointStateInterface     jointStateInterface_;
84      hardware_interface::PositionJointInterface   positionInterface_;
85      hardware_interface::PosVelJointInterface    posVelInterface_;
86
87      std::vector<std::string> posIfaceClaimsNonRT;
88      std::vector<std::string> posVelIfaceClaimsNonRT;
89
90      realtime_tools::RealtimeBuffer<std::vector<std::string> > posIfaceClaimBuffer;
91      realtime_tools::RealtimeBuffer<std::vector<std::string> > posVelIfaceClaimBuffer;
92
93  public:
94      DxlRobotHW(std::vector<JointID> jointIDs, const char* deviceName = "/dev/ttyUSB0",
            const float protocol = 1.0, const int baud_rate = 1000000);
95
96      // RobotHW interface
97  public:
98      void doSwitch(const std::list<hardware_interface::ControllerInfo> & start_list,
99                    const std::list<hardware_interface::ControllerInfo> & stop_list);
100
101
102     // RobotHW interface
```

```
103 public:
104     void read(const ros::Time &, const ros::Duration &);
105     void write(const ros::Time &, const ros::Duration &);
106 
107 private:
108     void read();
109     void write();
110 };
```

code/dxl_robot_hw/src/DxlRobotHW.cpp

```
1  #include <DxlRobotHW.h>
2  #include <hardware_interface/robot_hw.h>
3  #include <hardware_interface/joint_state_interface.h>
4  #include <hardware_interface/joint_command_interface.h>
5  #include <algorithm>
6  #include <pthread.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9  #include <sys/syscall.h>
10 #include <thread>
11 
12 using namespace hardware_interface;
13 
14 #define WORD_LEN             2
15 
16 #define GOAL_POS_ADDR        30
17 
18 #define PRESENT_POS_ADDR     36
19 #define PRESENT_VEL_ADDR     38
20 #define PRESENT_EFF_ADDR     40
21 
22 
23 
24 DxlRobotHW::DxlRobotHW(std::vector<JointID> jointIDs, const char *deviceName, const float
       protocol, const int baud_rate)
25 {
26 
27 //     struct sched_param param;
28 
29 //     //It can't be greater than 98
30 //     param.__sched_priority = 98;
31 
32 //     //TODO: success check
33 //     std::cout << pthread_setschedparam(pthread_self(), SCHED_FIFO, &param) << std::endl
      ;
34 
35     portHandler_ = dynamixel::PortHandler::getPortHandler(deviceName);
36     packetHandler_ = dynamixel::PacketHandler::getPacketHandler(protocol);
37 
38     // Open port
39     if (portHandler_->openPort())
40         printf("Succeeded to open the port!\n");
41     else
42         printf("Failed to open the port!\n");
43 
44     // Set port baudrate
45     if (portHandler_->setBaudRate(baud_rate))
46         printf("Succeeded to change the baudrate!\n");
47     else
```

```
48          printf("Failed to change the baudrate!\n");
49
50      size_t i = 0;
51      for(auto jointID : jointIDs){
52
53          uint16_t model_number = 12;
54
55          if(packetHandler_->ping(portHandler_,uint8_t(jointID.id),&model_number)==
                COMM_SUCCESS){
56              DxlInfo dxl(jointID,dxl_interface::ModelSpec::getByNumber(model_number,
                    protocol, DEFAULT_MODEL_SPEC_FOLDER,DEFAULT_MODEL_SPEC_FILE_EXTENSION));
57              dxlInfos.push_back(dxl);
58              dxlNameIdxMap[jointID.name] = i++;
59          }
60      }
61
62      read();
63
64      for(DxlInfo& dxl : dxlInfos){
65          jointStateInterface_.registerHandle(JointStateHandle(dxl.jointID.name,&dxl.pos,&
                dxl.vel,&dxl.eff));
66          dxl.posCmd = dxl.pos;
67          dxl.velCmd = 0;
68          positionInterface_.registerHandle(JointHandle(jointStateInterface_.getHandle(dxl.
                jointID.name),&dxl.posCmd));
69          posVelInterface_.registerHandle(PosVelJointHandle(jointStateInterface_.getHandle(
                dxl.jointID.name),&dxl.posCmd,&dxl.velCmd));
70      }
71
72      registerInterface(&jointStateInterface_);
73      registerInterface(&positionInterface_);
74      registerInterface(&posVelInterface_);
75
76  }
77
78
79
80  void DxlRobotHW::read()
81  {
82      // FIXME: restrito a protocol 1.0
83
84      uint16_t values[3] = {512,0,0};
85
86  //    int policy;
87  //    struct sched_param param;
88
89  //    auto tid = std::this_thread::get_id();
90  //    pthread_getschedparam(pthread_self(),&policy,&param);
91
92  //    std::cout << tid << " " << policy << " " << param.sched_priority << std::endl;
93
94      //    std::cout << "READ" << std::endl;
95
96      uint8_t error;
97
98      for(DxlInfo& dxl : dxlInfos){
99
100         auto res = packetHandler_->readTxRx(portHandler_,uint8_t(dxl.jointID.id),
                PRESENT_POS_ADDR,6,reinterpret_cast<uint8_t*>(&values[0]),&error);
101
```

```
102 //          packetHandler_->printRxPacketError(error);
103 //          packetHandler_->printTxRxResult(res);
104
105         dxl.pos = dxl.jointID.getDirection()*(dxl.spec.valueToRadian(values[0])-dxl.
                jointID.reference);
106         dxl.vel = dxl.jointID.getDirection()*dxl.spec.valueToVelocity(values[1]);
107
108         if(values[2] < 1024)
109             dxl.eff = values[2]/1024.;
110         else
111             dxl.eff = (1024 - values[2])/1024.;
112
113 //          std::cout << "name: " << dxl.jointID.name << " pos: " << dxl.pos << " vel: " <<
      dxl.vel << " eff: " << dxl.eff;
114 //          std::cout << " pos_dxl: " << values[0] << " vel_dxl: " << values[1] << "
     eff_dxl: " << values[2] << std::endl;
115
116     }
117 }
118
119 void DxlRobotHW::write()
120 {
121     //FIXME: restrito a protocolo 1.0
122
123     std::vector<std::string>& posIfaceClaimsRT = *posIfaceClaimBuffer.readFromRT();
124     std::vector<std::string>& posVelIfaceClaimsRT = *posVelIfaceClaimBuffer.readFromRT();
125
126     if(posIfaceClaimsRT.size() > 0){
127         //          std::cout << "WRITE POSITION:" << std::endl;
128
129         dynamixel::GroupSyncWrite writePacket(portHandler_,packetHandler_,GOAL_POS_ADDR
                ,2);
130
131         for(auto joint : posIfaceClaimsRT){
132             DxlInfo& dxl = dxlInfos[dxlNameIdxMap[joint]];
133             dxl.posCmd_dxl = uint16_t(dxl.spec.radianToValue(dxl.jointID.getDirection()*
                    dxl.posCmd+dxl.jointID.reference));
134             writePacket.addParam(uint8_t(dxl.jointID.id),reinterpret_cast<uint8_t*>(&dxl.
                    posCmd_dxl));
135             //              std::cout << "name: " << dxl.jointID.name << " posCmd: " << dxl
                    .posCmd << "posCmd_dxl: " << dxl.posCmd_dxl << std::endl;
136         }
137
138         writePacket.txPacket();
139     }
140
141     if(posVelIfaceClaimsRT.size() > 0){
142
143         //          std::cout << "WRITE POSVEL:" << std::endl;
144
145         dynamixel::GroupSyncWrite writePacket(portHandler_,packetHandler_,GOAL_POS_ADDR
                ,4);
146
147         for(auto joint : posVelIfaceClaimsRT){
148             DxlInfo& dxl = dxlInfos[dxlNameIdxMap[joint]];
149
150             dxl.posVelCmd_dxl[0] = uint16_t(dxl.spec.radianToValue(dxl.jointID.
                    getDirection()*dxl.posCmd+dxl.jointID.reference));
151             dxl.posVelCmd_dxl[1] = uint16_t(dxl.spec.velocityToValue(dxl.velCmd));
152
```

```
153             //dxl.posVelCmd_dxl[1] = (dxl.posVelCmd_dxl[1]==0)?1:dxl.posVelCmd_dxl[1];
154
155             writePacket.addParam(uint8_t(dxl.jointID.id),reinterpret_cast<uint8_t*>(dxl.
                    posVelCmd_dxl));
156             //              std::cout << "name: " << dxl.jointID.name << " posCmd: " << dxl
                    .posCmd << " velCmd: " << dxl.velCmd;
157             //              std::cout << " posCmd_dxl: " << dxl.posVelCmd_dxl[0] <<  "
                    velCmd_dxl: " << dxl.posVelCmd_dxl[1] << std::endl;
158         }
159
160         writePacket.txPacket();
161     }
162 }
163
164 void DxlRobotHW::read(const ros::Time&, const ros::Duration&) {
165     read();
166 }
167
168 void DxlRobotHW::write(const ros::Time&, const ros::Duration&) {
169     write();
170 }
171
172 void DxlRobotHW::doSwitch(const std::list<ControllerInfo> &start_list, const std::list<
    ControllerInfo> & stop_list)
173 {
174
175     std::cout << "DO SWITCH" << std::endl;
176
177
178     std::cout << "Stop List" << std::endl;
179     for(auto stopInfo : stop_list){
180         for(auto ifaceRes : stopInfo.claimed_resources){
181
182             std::cout << ifaceRes.hardware_interface << ":";
183
184             for(auto res : ifaceRes.resources)
185                 std::cout << " " << res;
186
187             std::cout << std::endl;
188
189             if(ifaceRes.hardware_interface == "hardware_interface::PositionJointInterface
                    "){
190                 for(auto res : ifaceRes.resources)
191                     posIfaceClaimsNonRT.erase(std::remove(posIfaceClaimsNonRT.begin(),
                            posIfaceClaimsNonRT .end(),res));
192             }
193
194             if(ifaceRes.hardware_interface == "hardware_interface::PosVelJointInterface")
                    {
195                 for(auto res : ifaceRes.resources)
196                     posVelIfaceClaimsNonRT.erase(std::remove(posVelIfaceClaimsNonRT.begin
                            (),posVelIfaceClaimsNonRT.end(),res));
197             }
198         }
199     }
200
201     std::cout << "Start List" << std::endl;
202     for(auto startInfo : start_list){
203         for(auto ifaceRes : startInfo.claimed_resources){
204
```

```
205             std::cout << ifaceRes.hardware_interface << ":";
206
207             for(auto res : ifaceRes.resources)
208                 std::cout << " " << res;
209
210             std::cout << std::endl;
211
212             if(ifaceRes.hardware_interface == "hardware_interface::PositionJointInterface
                    "){
213                 for(auto res : ifaceRes.resources)
214                     posIfaceClaimsNonRT.push_back(res);
215             }
216             if(ifaceRes.hardware_interface == "hardware_interface::PosVelJointInterface")
                    {
217                 for(auto res : ifaceRes.resources)
218                     posVelIfaceClaimsNonRT.push_back(res);
219             }
220         }
221     }
222
223     std::cout << "posIfaceClaims:";
224     for(auto res : posIfaceClaimsNonRT)
225         std::cout << " " << res;
226     std::cout << std::endl;
227
228     std::cout << "posVelIfaceClaims:";
229     for(auto res : posVelIfaceClaimsNonRT)
230         std::cout << " " << res;
231     std::cout << std::endl;
232
233     posIfaceClaimBuffer.writeFromNonRT(posIfaceClaimsNonRT);
234     posVelIfaceClaimBuffer.writeFromNonRT(posVelIfaceClaimsNonRT);
235 }
```

B.2 Control Loop

code/control_loop/include/icontroller.h

```
#ifndef ICONTROLLER_INTERFACE_H
#define ICONTROLLER_INTERFACE_H

#include <chrono>
#include <memory>

using namespace std;

namespace control_loop{

/**
 * @brief Represents a controller to ControlLoop Similar to controller_interface::
     ControllerInterface of ros_control.
 *
 * The way the controllers interact with the control_loop::IHardware is up to the user.
 */
class IController {

public:


    /**
     * @brief Method called right after IHardware::write(), designed to specify the
         parameters of IHardware::read() if needed
     *
     * @param now current time
     * @return bool nothing yet
     */
    virtual bool prepareRead(std::chrono::steady_clock::time_point now) = 0;


    /**
     * @brief Similar to ros_control ControllerInterface::update. Designed to get the
         last values
     * read from IHardware and specify the commands of the next IHardware::write()
     *
     * @param now current time
     * @return bool nothing yet
     */
    virtual bool update(std::chrono::steady_clock::time_point now) = 0;

    virtual ~IController(){}
};

typedef shared_ptr<IController> ControllerPtr;

}

#endif
```

code/control_loop/include/ihardware.h

```
#ifndef IHARDWARE_INTERFACE_H
#define IHARDWARE_INTERFACE_H

```

```
4  #include <memory>
5
6  using namespace std;
7
8  namespace control_loop{
9
10
11 /**
12  * @brief Represents a 'Hardware' to ControlLoop. Similar to hardware_interface::RobotHW
        of ros_control.
13  *
14  * The way the controllers interact with this entity is up to the user implementation.
15  *
16  */
17 class IHardware {
18
19     public:
20
21     /**
22      * @brief Send the last specified commands to actuadors
23      *
24      */
25     virtual void write() = 0;
26
27     /**
28      * @brief Read values from robot's sensors
29      *
30      */
31     virtual void read() = 0;
32
33     virtual ~IHardware(){}
34 };
35
36 typedef shared_ptr<IHardware> HardwarePtr;
37
38 }
39
40 #endif
```

code/control_loop/include/control_loop.h

```
1  #ifndef CONTROLLER_TIME_HANDLER_H
2  #define CONTROLLER_TIME_HANDLER_H
3
4  #include <map>
5  #include <string>
6  #include <chrono>
7  #include <thread>
8  #include <mutex>              // mutex, unique_lock
9  #include <condition_variable> // condition_variable
10 #include <icontroller.h>
11 #include <ihardware.h>
12 #include <functional>
13
14 namespace control_loop {
15
16 /**
17  * @brief Represents a potencially real-time control loop.
18  *
19  * It has a single real-time inner thread that periodically
```

```
20  * read values from the sensors, process the input commands and finally send the proper
        signals to the actuators. The interaciton with the
21  * system is handled by control_loop::IHardware and the "control boxes" are handled by
        instances of control_loop::IController.
22  *
23  */
24 class ControlLoop {
25
26 public:
27
28 //    ControlLoop();
29     ControlLoop(HardwarePtr hwInterface);
30
31     virtual ~ControlLoop();
32
33 //    void setHardware(HardwarePtr hw);
34
35
36     //TODO: check if it works
37     /**
38      * @brief Load a controller so that, on the next iteration, it will be processed
39      *
40      * @param controllerName Given name to the controller
41      * @param controller An instance of controller_loop::IController
42      * @return bool not implemented yet
43      */
44     bool loadController(std::string controllerName, ControllerPtr controller);
45
46
47     /**
48      * @brief
49      *
50      * @param controllerName
51      * @return ControllerPtr
52      */
53     ControllerPtr unloadController(std::string controllerName);
54
55
56     /**
57      * @brief
58      *
59      * @param controller_out_name
60      * @param controller_in_name
61      * @param controller_in
62      * @return ControllerPtr
63      */
64     ControllerPtr switchController(std::string controller_out_name, std::string
          controller_in_name, ControllerPtr controller_in);
65
66
67     /**
68      * @brief Simply continue the loop iterations
69      *
70      * @param readWaitTime
71      * @return bool returns true if the loop is successfully resumed and false when it
          was not possible (usually, it happens when the IHardware was not loaded yet)
72      */
73     bool resumeLoop(std::chrono::microseconds readWaitTime = std::chrono::microseconds(0)
          );
74
```

```
75
76      /**
77       * @brief Imediatelly pauses the loop
78       *
79       */
80      void pauseLoop();
81
82
83      /**
84       * @brief Stops the ControlLoop so that it cannot be used anymore
85       *
86       */
87      void close();
88
89
90      /**
91       * @brief Sets the loop frequency in real-time. It can be called during execution
92       *
93       * @param freq the desired frequency in Hz
94       * @return bool not implemented yet
95       */
96      bool setFrequency(double freq);
97
98
99      /**
100      * @brief safelly gets the set frequency
101      *
102      * @return double Loop's frequency in Hz
103      */
104     double getFrequency();
105
106 protected:
107
108     virtual void prepareRead();//entre a escrita e a leitura
109
110     /**
111      * @brief By default, only calls hardwareInterface->read()
112      *
113      */
114     virtual void read();
115
116
117     /**
118      * @brief By default, calls the method "update" of all controllers loaded (similar to
             ros_control's ControllerManager)
119      *
120      */
121     virtual void update();
122
123
124     /**
125      * @brief By default, calls hardwareInterface->write()
126      *
127      */
128     virtual void write();
129
130     /**
131      * @brief This method is called when the deadlines aren't met. The default
             implementation only prints the estimated delay and the desired period.
132      * By default, the missed iteration is processed and the next deadline is updated to
```

```
        "realization + period"
133     *
134     * @param desired The missed deadline time_point
135     * @param realization Current time
136     */
137    virtual void onMiss(std::chrono::steady_clock::time_point desired, std::chrono::
           steady_clock::time_point realization);
138
139
140 private:
141
142    std::thread mainThread; /**< inner real-time tread*/
143
144    //TODO: change it to multiple hardware interfaces
145    HardwarePtr hardwareInterface;
146    std::map<std::string,ControllerPtr> controllers;
147    std::map<std::string,ControllerPtr> rtControllers;
148
149    std::chrono::microseconds period = std::chrono::microseconds(long(20e3));//default:
           20ms
150    std::chrono::microseconds rtPeriod;
151
152    //Variable access control flags
153    bool isPaused = true;
154    bool isClosed = false;
155
156    bool hasPeriodChange = true;
157    bool hasControllerChange = true;
158
159    std::mutex requestMtx;
160
161    std::condition_variable pauseCv;
162
163    std::chrono::steady_clock::time_point resumeTime;
164    std::chrono::steady_clock::time_point nextLoopTime;
165
166    //Initialization helper methods
167    void defaultInit();
168    bool initThread();
169
170    void loop();
171    void loopUpdateCheck();//return false if it was closed
172
173 };
174
175 }
176 #endif
```

code/control_loop/src/control_loop.cpp

```
1  #include <control_loop.h>
2  #include <iostream>
3  #include <functional>
4  #include <unistd.h>
5  #include <sys/mman.h>
6  #include <sys/capability.h>
7
8  using namespace control_loop;
9
10 void ControlLoop::update()
```

```
11 {
12     for(auto controller : controllers)
13         controller.second->update(std::chrono::steady_clock::now());
14 }
15
16 void ControlLoop::write()
17 {
18     hardwareInterface->write();
19 }
20
21 void ControlLoop::defaultInit()
22 {
23     setFrequency(50);
24     initThread();
25 }
26
27
28 void ControlLoop::prepareRead()
29 {
30     for(auto controller : controllers)
31         controller.second->prepareRead(std::chrono::steady_clock::now());
32 }
33
34 void ControlLoop::onMiss(std::chrono::steady_clock::time_point desire, std::chrono::_V2::
       steady_clock::time_point realization){
35
36     std::cout << "ON MISS:"
37               << "\tDelay="<< std::chrono::duration_cast<std::chrono::microseconds>(
                      realization-desire).count()/1e3 << "ms\t"
38               << "\tPeriod="<< period.count()/1e3 << "ms\t"
39               << std::endl;
40
41 }
42
43
44 //ControlLoop::ControlLoop()
45 //{
46 //    defaultInit();
47 //}
48
49 ControlLoop::ControlLoop(HardwarePtr hwInterface)
50 {
51     defaultInit();
52     hardwareInterface = hwInterface;
53 }
54
55 ControlLoop::~ControlLoop(){
56     this->close();
57 }
58
59 //void ControlLoop::setHardware(HardwarePtr hw)
60 //{
61 //    this->hardwareInterface = hw;
62 //}
63
64
65 bool ControlLoop::loadController(std::string controllerName, ControllerPtr controller)
66 {
67     //TODO: fazer tratamentos adequados para nao substituir existentes
68     std::lock_guard<std::mutex> lck(requestMtx);
```

```
69
70      this->controllers[controllerName] = controller;
71
72      return true;
73  }
74
75
76  ControllerPtr ControlLoop::unloadController(std::string controllerName)
77  {
78      //TODO: fazer tratamentos adequados para retirar um controller
79
80      std::lock_guard<std::mutex> lck(requestMtx);
81
82      auto controller_out = controllers[controllerName];
83
84      controllers.erase(controllerName);
85
86      return controller_out;
87  }
88
89
90  ControllerPtr ControlLoop::switchController(std::string controller_out_name, std::string
        controller_in_name, ControllerPtr controller_in)
91  {
92      //TODO: fazer tratamentos adequados do switch
93      std::lock_guard<std::mutex> lck(requestMtx);
94
95      auto controller_out = controllers[controller_out_name];
96
97      controllers.erase(controller_out_name);
98
99      controllers[controller_in_name] = controller_in;
100
101     return controller_out;
102 }
103
104
105 bool ControlLoop::resumeLoop(std::chrono::microseconds readWaitTime)
106 {
107
108     std::unique_lock<std::mutex> lck(requestMtx);
109
110     //The loop is resumed just when there is a hardware loaded
111     if(this->hardwareInterface){
112
113         this->resumeTime = std::chrono::steady_clock::now();
114
115         //Sets the first iterations one period from the resume time
116         this->nextLoopTime = this->resumeTime + readWaitTime;
117
118         this->isPaused = false;
119
120         //Notify the main thread
121         pauseCv.notify_all();
122
123         lck.unlock();
124         return true;
125     }else{
126         lck.unlock();
127         return false;//TODO: error msg or exception
```

```
128     }
129
130 }
131
132
133
134
135 void ControlLoop::loop()
136 {
137     cap_t cap = cap_get_pid(getpid());
138     cap_flag_value_t pFlag = CAP_CLEAR, eFlag = CAP_CLEAR;
139
140     cap_get_flag(cap,CAP_IPC_LOCK,CAP_PERMITTED,&pFlag);
141     cap_get_flag(cap,CAP_IPC_LOCK,CAP_EFFECTIVE,&eFlag);
142
143     bool mlock = pFlag==CAP_SET && eFlag == CAP_SET;
144
145     if(mlock)
146         std::cout << "mlock: " << mlockall(MCL_FUTURE) << std::endl;;
147
148
149     std::chrono::steady_clock::time_point now;
150
151
152     //Check if the loop is paused
153     std::unique_lock<std::mutex> lck(requestMtx);
154     while(isPaused)
155         pauseCv.wait(lck);
156     lck.unlock();
157
158
159     while(true) {
160
161         //Process external commands, such as frequency update
162         loopUpdateCheck();
163
164         if (isClosed)
165             break;
166
167         //Right after the last write()
168         prepareRead();
169
170         auto now = std::chrono::steady_clock::now();
171
172         std::this_thread::sleep_until(nextLoopTime);
173
174         //Without the first condition, the method could be called on the first iteration
175         if(nextLoopTime > resumeTime && now > nextLoopTime){
176             onMiss(nextLoopTime,now);
177             nextLoopTime = now;//update the current deadline
178         }
179
180         read();
181         update();
182         write();
183
184         nextLoopTime += rtPeriod;
185
186     }
187
```

```
188     if(mlock)
189         munlockall();
190 
191     //This command is needed to give time to close() method to join this thread
192     sleep(1);
193 }
194 
195 void ControlLoop::loopUpdateCheck()
196 {
197 
198     //try_lock is used to be real-time safe, otherwise it could depend on other non-
        realtime threads
199     std::unique_lock<std::mutex> lck(requestMtx,std::try_to_lock);
200 
201     if(lck.owns_lock()){
202         while(isPaused)
203             pauseCv.wait(lck);
204 
205         if(hasPeriodChange){
206             rtPeriod = period;
207             hasPeriodChange = false;
208         }
209 
210         if(hasControllerChange){
211             rtControllers = controllers;
212             hasControllerChange = false;
213         }
214 
215         lck.unlock();
216     }
217 }
218 
219 bool ControlLoop::initThread()
220 {
221     this->isPaused = true;
222 
223     this->isClosed = false;
224 
225     mainThread = std::thread(&ControlLoop::loop, this);
226 
227     //    mainThread.detach();
228 
229     //TODO:: make scheduler policy and priority settable on construction
230 
231     struct sched_param param;
232 
233     //It can't be greater than 98
234     param.__sched_priority = 31;
235 
236     std::cout << "realtime: " << pthread_setschedparam(mainThread.native_handle(),
        SCHED_RR, &param) << std::endl;
237 
238     //TODO: success check
239 
240 
241     return true;
242 }
243 
244 void ControlLoop::read()
245 {
```

```
246     hardwareInterface->read();
247 }
248
249
250 void ControlLoop::close()
251 {
252     std::unique_lock<std::mutex> lck(requestMtx);
253
254     isClosed = true;
255     isPaused = false;
256
257     //Unpause loop
258     pauseCv.notify_all();
259
260     lck.unlock();
261
262     //Wait mainThread to finish
263     if(mainThread.joinable())
264         this->mainThread.join();
265
266 }
267
268
269 bool ControlLoop::setFrequency(double freq)
270 {
271     //TODO: exception?
272     if(freq<=0)
273         return false;
274
275     std::lock_guard<std::mutex> lck(requestMtx);
276
277     hasPeriodChange = true;
278     period = std::chrono::microseconds(long((1/freq)*1e6));
279
280     return true;
281
282 }
283
284
285 double ControlLoop::getFrequency()
286 {
287     return (1./(period.count()/double(1e6)));
288 }
289
290
291 void ControlLoop::pauseLoop()
292 {
293     std::lock_guard<std::mutex> lck(requestMtx);
294
295     isPaused = true;
296
297 }
```

code/control_loop/include/robot_hw_adapter.h

```
1 #ifndef ROBOT_HW_ADAPTER_H
2 #define ROBOT_HW_ADAPTER_H
3
4 #include <control_loop.h>
5 #include <ihardware.h>
```

```
6  #include <hardware_interface/robot_hw.h>
7  #include <ros/ros.h>
8
9  namespace control_loop {
10
11 class RobotHWAdapter : public IHardware {
12
13 private:
14     hardware_interface::RobotHW* robotHW;
15     ros::Time lastWriteTime;
16     ros::Time lastReadTime;
17
18     // IHardware interface
19 public:
20     void write();
21
22     void read();
23
24     RobotHWAdapter(hardware_interface::RobotHW *rh);
25
26     hardware_interface::RobotHW* getRobotHW();
27
28 };
29
30 }
31 #endif
```

code/control_loop/src/robot_hw_adapter.cpp

```
1  #include <robot_hw_adapter.h>
2
3  void control_loop::RobotHWAdapter::write(){
4      auto now = ros::Time::now();
5      robotHW->write(now,now-lastWriteTime);
6      lastWriteTime = now;
7  }
8
9  void control_loop::RobotHWAdapter::read(){
10     auto now = ros::Time::now();
11     robotHW->read(now,now-lastReadTime);
12     lastReadTime = now;
13
14 }
15
16 control_loop::RobotHWAdapter::RobotHWAdapter(hardware_interface::RobotHW *rh){
17     robotHW = rh;
18     lastWriteTime = ros::Time::now();
19     lastReadTime = ros::Time::now();
20 }
21
22 hardware_interface::RobotHW *control_loop::RobotHWAdapter::getRobotHW()
23 {
24     return robotHW;
25 }
```

code/control_loop/include/ros_controller_manager_adapter.h

```
1  #ifndef ROS_CONTROLLER_MANAGER_ADAPTER
2  #define ROS_CONTROLLER_MANAGER_ADAPTER
```

```
3
4  #include <controller_manager/controller_manager.h>
5  #include <icontroller.h>
6  #include <ros/ros.h>
7  #include <memory>
8
9  using namespace controller_manager;
10 using namespace std;
11
12 namespace control_loop{
13
14 class RosControllerManagerAdapter : public IController{
15
16 private:
17
18     ros::Time lastUpdateTime = ros::Time::now();
19     ControllerManager controllerManager;
20
21     // IController interface
22 public:
23
24     RosControllerManagerAdapter(hardware_interface::RobotHW *robot_hw,
25                                 const ros::NodeHandle& nh=ros::NodeHandle());
26
27     ControllerManager& getControllerManager();
28
29     virtual bool prepareRead(chrono::steady_clock::time_point);
30
31     virtual bool update(chrono::steady_clock::time_point);
32
33 };
34
35 }
36 #endif
```

code/control_loop/src/ros_controller_manager_adapter.cpp

```
1  #include <ros_controller_manager_adapter.h>
2
3  using namespace control_loop;
4
5  RosControllerManagerAdapter::RosControllerManagerAdapter(hardware_interface::RobotHW *
       robot_hw, const ros::NodeHandle &nh)
6      : controllerManager(robot_hw, nh) {
7
8  }
9
10 ControllerManager &RosControllerManagerAdapter::getControllerManager()
11 {
12     return controllerManager;
13 }
14
15 bool RosControllerManagerAdapter::prepareRead(std::chrono::steady_clock::time_point){
       return true;}
16
17 bool RosControllerManagerAdapter::update(std::chrono::steady_clock::time_point){
18     ros::Time now = ros::Time::now();
19     ros::Duration duration = now - lastUpdateTime;
20
21     lastUpdateTime = now;
```

```
22
23      controllerManager.update(now, duration);
24
25      return true;
26  }
```

code/control_loop/include/ros_control_loop.h

```
1  #ifndef ROS_CONTROL_LOOP_H
2  #define ROS_CONTROL_LOOP_H
3
4  #include <control_loop.h>
5  #include <ihardware.h>
6  #include <icontroller.h>
7  #include <ros/ros.h>
8  #include <ros_controller_manager_adapter.h>
9  #include <hardware_interface/robot_hw.h>
10 #include <controller_manager/controller_manager.h>
11 #include <chrono>
12 #include <robot_hw_adapter.h>
13 #include <memory>
14
15 namespace control_loop {
16
17 class RosControlLoop : public ControlLoop {
18
19 public:
20
21     RosControlLoop(std::shared_ptr<hardware_interface::RobotHW> rh, const ros::NodeHandle
        & nh=ros::NodeHandle());
22
23     // ControlLoop interface
24 protected:
25
26     void onMiss(chrono::steady_clock::time_point desire, chrono::steady_clock::time_point
         realization);
27
28     void onMiss(ros::Time desire, ros::Time realization, ros::Duration delay);
29 };
30
31 }
32 #endif
```

code/control_loop/src/ros_control_loop.cpp

```
1  #include <ros_control_loop.h>
2
3  control_loop::RosControlLoop::RosControlLoop(std::shared_ptr<hardware_interface::RobotHW>
       rh, const ros::NodeHandle &nh):
4      control_loop::ControlLoop(std::make_shared<RobotHWAdapter>(rh.get())){
5
6      this->loadController("ControllerManager", std::make_shared<
           RosControllerManagerAdapter>(rh.get(),nh));
7      cout.setf(ios::fixed);
8  }
9
10 void control_loop::RosControlLoop::onMiss(chrono::steady_clock::time_point desire, chrono
       ::steady_clock::time_point realization){
11     auto rosNow = ros::Time::now();
```

```
12
13      auto timeDiff = std::chrono::duration_cast<std::chrono::nanoseconds>(realization-
            desire);
14
15      auto rosTimeDiff = ros::Duration(int(round(timeDiff.count()/1e9)),timeDiff.count()%
            long(1e9));
16
17      onMiss(rosNow - rosTimeDiff,rosNow, rosTimeDiff);
18  }
19
20  void control_loop::RosControlLoop::onMiss(ros::Time desire, ros::Time realization, ros::
        Duration delay){
21      std::cout << "ON MISS (ROS):"
22                << "\tDelay="<< delay.sec*1000 + delay.nsec/1e6 << "ms"
23                << "\tPeriod="<< (1/getFrequency())*1e3 << "ms"
24                << "\tDesire=" << double(desire.sec + desire.nsec/1e9)
25                << "\tRealization=" << double(realization.sec + realization.nsec/1e9)
26                << std::endl;
27  }
```

B.3 posvel_controllers

code/posvel_controllers/include/forward_joint_trajectory_point_controller.h

```
#ifndef FORWARD_TRAJECTORY_POINT_CONTROLLER_FORWARD_JOINT_GROUP_TRAJECTORY_POINT_CONTROLLER_H
#define FORWARD_TRAJECTORY_POINT_CONTROLLER_FORWARD_JOINT_GROUP_TRAJECTORY_POINT_CONTROLLER_H

#include <vector>
#include <string>

#include <ros/node_handle.h>
#include <hardware_interface/joint_command_interface.h>
#include <controller_interface/controller.h>
#include <realtime_tools/realtime_buffer.h>
#include <trajectory_msgs/JointTrajectoryPoint.h>

namespace forward_trajectory_point_controller
{

template<typename T, typename HandleType = typename T::ResourceHandleType>
class ForwardJointTrajectoryPointController: public controller_interface::Controller<T>
{
public:
    ForwardJointTrajectoryPointController() {}
    ~ForwardJointTrajectoryPointController() {sub_command_.shutdown();}

    struct PosVelAccEff{
        double position = 0;
        double velocity = 0;
        double acceleration = 0;
        double effort = 0;

        PosVelAccEff(){}

        PosVelAccEff(double pos, double vel, double acc, double eff):position(pos),
            velocity(vel),acceleration(acc),effort(eff){}
    };

    bool init(T* hw, ros::NodeHandle &n)
    {
        // List of controlled joints
        std::string param_name = "joints";
        if(!n.getParam(param_name, joint_names_))
        {
            ROS_ERROR_STREAM("Failed to getParam '" << param_name << "' (namespace: " <<
                n.getNamespace() << ").");
            return false;
        }
        n_joints_ = joint_names_.size();

        if(n_joints_ == 0){
            ROS_ERROR_STREAM("List of joint names is empty.");
            return false;
        }
        for(unsigned int i=0; i<n_joints_; i++)
        {
            try
            {
```

```
53                  joints_.push_back(hw->getHandle(joint_names_[i]));
54              }
55              catch (const hardware_interface::HardwareInterfaceException& e)
56              {
57                  ROS_ERROR_STREAM("Exception thrown: " << e.what());
58                  return false;
59              }
60          }
61
62          commands_buffer_.writeFromNonRT(std::vector<PosVelAccEff>(n_joints_, PosVelAccEff
                ()));
63
64          sub_command_ = n.subscribe<trajectory_msgs::JointTrajectoryPoint>("command", 1, &
                ForwardJointTrajectoryPointController::commandCB, this);
65          return true;
66      }
67
68  //    void starting(const ros::Time& time);
69  //    virtual void update(const ros::Time &, const ros::Duration &) = 0;
70
71      std::vector< std::string > joint_names_;
72      std::vector< HandleType > joints_;
73      realtime_tools::RealtimeBuffer<std::vector<PosVelAccEff> > commands_buffer_;
74      unsigned int n_joints_;
75
76  protected:
77      ros::Subscriber sub_command_;
78
79      virtual void commandCB(const trajectory_msgs::JointTrajectoryPointConstPtr&) = 0;
80
81  };
82
83  }
84
85  #endif
```

code/posvel_controllers/include/joint_group_posvel_controller.h

```
1  #ifndef POSVEL_CONTROLLERS_JOINT_GROUP_POSVEL_CONTROLLER_H
2  #define POSVEL_CONTROLLERS_JOINT_GROUP_POSVEL_CONTROLLER_H
3
4  #include <forward_joint_trajectory_point_controller.h>
5  #include <hardware_interface/posvel_command_interface.h>
6
7
8  namespace posvel_controllers
9  {
10
11 class JointGroupPosVelController : public forward_trajectory_point_controller::
       ForwardJointTrajectoryPointController <hardware_interface::PosVelJointInterface> {
12
13 public:
14     JointGroupPosVelController() {}
15     ~JointGroupPosVelController() {sub_command_.shutdown();}
16
17     void update(const ros::Time &, const ros::Duration &);
18
19 private:
20     void commandCB(const trajectory_msgs::JointTrajectoryPointConstPtr &msg);
21 };
```

```
22
23
24 }
25
26 #endif
```

code/posvel_controllers/src/joint_group_posvel_controller.cpp

```
1  #include <pluginlib/class_list_macros.h>
2  #include <joint_group_posvel_controller.h>
3
4  void posvel_controllers::JointGroupPosVelController::update(const ros::Time &, const ros
       ::Duration &) {
5      std::vector<PosVelAccEff> & commands = *commands_buffer_.readFromRT();
6      for(unsigned int i=0; i<n_joints_; i++)
7      {  joints_[i].setCommand(commands[i].position,commands[i].velocity);  }
8  }
9
10 void posvel_controllers::JointGroupPosVelController::commandCB(const trajectory_msgs::
       JointTrajectoryPointConstPtr &msg){
11
12     if(msg->positions.size() != n_joints_ || msg->velocities.size() != n_joints_)
13     {
14         ROS_ERROR_STREAM("Dimension of command positions (" << msg->positions.size() << "
               ) or velocities(" << msg->velocities.size() <<
15                          ") do not match number of joints (" << n_joints_ << ")! Not
                               executing!");
16         return;
17     }
18
19     std::vector<PosVelAccEff> values;
20
21     for(unsigned int i = 0; i < n_joints_; i++)
22         values.push_back(PosVelAccEff(msg->positions[i],msg->velocities[i],0,0));
23
24     commands_buffer_.writeFromNonRT(values);
25 }
26
27 PLUGINLIB_EXPORT_CLASS(posvel_controllers::JointGroupPosVelController,
       controller_interface::ControllerBase);
```

code/posvel_controllers/include/posvel_joint_interface_adapter.h

```
1  #ifndef JOINT_TRAJECTORY_CONTROLLER_POSVEL_JOINT_INTERFACE_ADAPTER_H
2  #define JOINT_TRAJECTORY_CONTROLLER_POSVEL_JOINT_INTERFACE_ADAPTER_H
3
4  #include <joint_trajectory_controller/hardware_interface_adapter.h>
5  #include <hardware_interface/posvel_command_interface.h>
6
7  template <class State>
8  class HardwareInterfaceAdapter<hardware_interface::PosVelJointInterface, State>
9  {
10 public:
11     HardwareInterfaceAdapter() : joint_handles_ptr_(0) {}
12
13     bool init(std::vector<hardware_interface::PosVelJointHandle>& joint_handles, ros::
           NodeHandle& /*controller_nh*/)
14     {
15         // Store pointer to joint handles
```

```
16          joint_handles_ptr_ = &joint_handles;
17
18          return true;
19      }
20
21      void starting(const ros::Time& /*time*/)
22      {
23          if (!joint_handles_ptr_) {return;}
24
25          // Semantic zero for commands
26          for (unsigned int i = 0; i < joint_handles_ptr_->size(); ++i)
27          {
28              (*joint_handles_ptr_)[i].setCommand((*joint_handles_ptr_)[i].getPosition(),0)
                    ;
29          }
30      }
31
32      void stopping(const ros::Time& /*time*/) {}
33
34      void updateCommand(const ros::Time&     /*time*/,
35                         const ros::Duration& /*period*/,
36                         const State&         desired_state,
37                         const State&         /*state_error*/)
38      {
39          // Forward desired position and velocity to command
40          const unsigned int n_joints = joint_handles_ptr_->size();
41          for (unsigned int i = 0; i < n_joints; ++i) {
42              (*joint_handles_ptr_)[i].setCommand(desired_state.position[i],desired_state.
                    velocity[i]);
43          }
44      }
45
46  private:
47      std::vector<hardware_interface::PosVelJointHandle>* joint_handles_ptr_;
48  };
49
50
51  #endif
```

code/posvel_controllers/include/posvel_joint_trajectory_controller.h

```
1   #ifndef JOINT_TRAJECTORY_POSVEL_CONTROLLER_H
2   #define JOINT_TRAJECTORY_POSVEL_CONTROLLER_H
3
4   #include <joint_trajectory_controller/joint_trajectory_controller.h>
5   #include <hardware_interface/posvel_command_interface.h>
6   #include <trajectory_interface/quintic_spline_segment.h>
7   #include <posvel_joint_interface_adapter.h>
8
9   namespace posvel_controllers {
10
11  typedef joint_trajectory_controller::JointTrajectoryController<trajectory_interface::
        QuinticSplineSegment<double>,
12                                                                 hardware_interface::
                                                                       PosVelJointInterface
                                                                       >
13          JointTrajectoryController;
14
15  }
16
```

```
17
18
19 #endif
```

code/posvel_controllers/src/posvel_joint_trajectory_controller.cpp

```
1 // Pluginlib
2 #include <pluginlib/class_list_macros.h>
3 #include <posvel_joint_trajectory_controller.h>
4
5 PLUGINLIB_EXPORT_CLASS(posvel_controllers::JointTrajectoryController,
      controller_interface::ControllerBase);
```

B.4 Programa de Registro de Latências

code/dxl_tests/rt_test.cpp

```cpp
#include <iostream>
#include <cstdio>
#include <chrono>
#include <pthread.h>
#include <sched.h>
#include <sys/mman.h>
#include <cstring>
#include <vector>
#include <thread>

void help() {
    std::cout << "\nrt_test [sleep_time] [n_iterations] [rt_option]" << "\n\n";

    std::cout << "sleep_time: default 1000 [us]" << std::endl;

    std::cout << "n_iterations: default 1000" << std::endl;

    std::cout << "rt_option: '-rt' or '-nonrt'" << std::endl;
}

int main(int argc, char** argv){

    long sleep_time = 1000; //us
    long n_iterations = 1000;
    bool real_time = false;
    bool show_help = false;

    if(argc >= 2){

        if(std::string(argv[1]).find("help")!=std::string::npos)
            show_help = true;
        else {
            sleep_time = std::stol(argv[1]);

            if(argc >= 4)
                n_iterations = std::stol(argv[2]);

            if(argc >= 4){
                if(strcmp(argv[3],"-rt") == 0)
                    real_time = true;
                else if(strcmp(argv[3],"-nonrt") == 0)
                    real_time = false;
                else
                    show_help = true;
            }else if(argc > 4)
                show_help = true;
        }
    }

    if(show_help){
        help();
        exit(-1);
    }

    if(real_time){
        mlockall(MCL_FUTURE);
```

```
57
58          struct sched_param param;
59          //param.__sched_priority = 98;
60          param.__sched_priority = 31;
61
62          //if(pthread_setschedparam(pthread_self(), SCHED_FIFO, &param)){
63          if(pthread_setschedparam(pthread_self(), SCHED_RR, &param)){
64              std::cout << "No permission to set real_time scheduler" << std::endl;
65              exit(-1);
66          }
67
68      }
69
70      std::vector<std::chrono::microseconds> timestamps(static_cast<unsigned long>(
            n_iterations));
71
72      std::chrono::microseconds sleep_duration(sleep_time);
73
74      std::chrono::steady_clock::time_point start_time = std::chrono::steady_clock::now()+
            std::chrono::seconds(1);
75      std::chrono::steady_clock::time_point next_time = start_time + sleep_duration;
76      std::chrono::steady_clock::time_point prev_time = start_time;
77
78      std::this_thread::sleep_until(start_time);
79
80      for (ulong i = 0; i < timestamps.size(); i++){
81
82          std::this_thread::sleep_until(next_time);
83          timestamps[i] = std::chrono::duration_cast<std::chrono::microseconds>(std::chrono
                ::steady_clock::now()-prev_time);
84          prev_time = next_time;
85          next_time += sleep_duration;
86      }
87
88      if(real_time)
89          munlockall();
90
91      std::cout << "expected_period," << sleep_time << std::endl;
92      std::cout << "iteration,period" << std::endl;
93
94      for(ulong i = 0; i < timestamps.size(); i++)
95          std::cout << i << "," << timestamps[i].count() << std::endl;
96
97      return 0;
98
99  }
```

APÊNDICE C – Modelo URDF do Robô de Teste

Para gerar o modelo URDF do robô humanoide de testes utilizou-se uma linguagem auxiliar chamada *xacro*[1], que permite construir arquivos URDF mais curtos e fáceis de compreender por meio da criação de *macros* que substituem expressões XML maiores. Ressalta-se que esta definição foi majoritariamente reaproveitada do projeto *ros-bioloid*[2]. As principais modificações foram a retirada das pernas e modificação da cabeça do modelo original, além da adição da IMU.

code/bioloid_example/urdf/bioloid_torso_std_part.xacro

```
<?xml version="1.0"?>
<!-- Standard Bioloid parts -->
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Torso -->
    <link name="torso_link">
        <visual>
            <xacro:default_geometry cad_file="BPF-CHEST-F"/>
            <origin rpy="${pi/2} 0 0"/>
        </visual>
        <visual>
            <xacro:default_geometry cad_file="BPF-CHEST-B"/>
            <origin rpy="${pi/2} 0 0"/>
        </visual>

        <visual>
            <xacro:default_geometry cad_file="CM-5"/>
            <origin xyz="0 0.032 -0.015" rpy="0 ${pi} 0"/>
        </visual>

        <!-- Sensor 100 -->
        <!--
        <visual>
            <xacro:default_geometry cad_file="AX-S1"/>
            <origin xyz="0 0 0.025" rpy="${-pi/2} 0 ${pi}"/>
        </visual>
        -->

        <!-- Motor 1 -->
        <visual>
            <xacro:default_geometry cad_file="AX-12"/>
            <origin xyz="-0.032 0 0" rpy="${pi/2} 0 ${-pi/2}"/>
        </visual>
        <visual>
            <xacro:default_geometry cad_file="FP04-F3"/>
            <origin xyz="-0.032 0.019 -0.0145" rpy="${pi} ${-pi/2} 0"/>
```

[1] <http://wiki.ros.org/xacro>
[2] <https://github.com/dxydas/ros-bioloid>

```
37          </visual>
38          <!-- Motor 2 -->
39          <visual>
40              <xacro:default_geometry cad_file="AX-12"/>
41              <origin xyz="0.032 0 0" rpy="${pi/2} 0 ${pi/2}"/>
42          </visual>
43          <visual>
44              <xacro:default_geometry cad_file="FP04-F3"/>
45              <origin xyz="0.032 0.019 -0.0145" rpy="${pi} ${-pi/2} 0"/>
46          </visual>
47          <visual>
48              <xacro:default_geometry cad_file="FP04-F8"/>
49              <origin xyz="0 0.0255 -0.0465" rpy="${-pi/2} 0 ${pi/2}"/>
50          </visual>
51          <visual>
52              <xacro:default_geometry cad_file="FP04-F8"/>
53              <origin xyz="0 0.0255 -0.0843" rpy="${pi/2} 0 ${pi/2}"/>
54          </visual>
55          <!-- Motor 7 -->
56          <visual>
57              <xacro:default_geometry cad_file="AX-12"/>
58              <origin xyz="-0.033 0.016 -0.0655" rpy="0 ${pi} 0"/>
59          </visual>
60          <!-- Motor 8 -->
61          <visual>
62              <xacro:default_geometry cad_file="AX-12"/>
63              <origin xyz="0.033 0.016 -0.0655" rpy="0 ${pi} 0"/>
64          </visual>
65          <collision name="head_collision">
66              <origin xyz="0 0.0122 0.035"/>
67              <geometry>
68                  <box size="0.033 0.054 0.044"/>
69              </geometry>
70          </collision>
71          <collision name="torso_collision_top">
72              <origin xyz="0 0.019 -0.015"/>
73              <geometry>
74                  <box size="0.103 0.0905 0.0635"/>
75              </geometry>
76          </collision>
77          <collision name="torso_collision_bottom">
78              <origin xyz="0 0.00625 -0.0648"/>
79              <geometry>
80                  <box size="0.099 0.065 0.043"/>
81              </geometry>
82          </collision>
83          <inertial>
84            <origin xyz="0 0 0"/>
85            <mass value="1.0"/>
86            <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001"
                  />
87          </inertial>
88      </link>
89
90  </robot>
```

code/bioloid_example/urdf/bioloid_without_legs.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="bioloid_without_legs">
```

```
3
4       <!-- Pi is now predefined -->
5       <!-- <xacro:property name="pi" value="3.14159"/> -->
6       <xacro:property name="cad_dir" value="file://$(find bioloid_master)/../../assets/cad"
            />
7       <!-- Typically, STL/DAE model units are in mm, but RViz units are in m, hence models
            must be scaled down by 1000. -->
8       <xacro:macro name="default_geometry" params="cad_file suffix:=_FCD-Conv_BDR-Col
            extension:=dae">
9           <geometry>
10              <mesh filename="${cad_dir}/${cad_file}${suffix}.${extension}" scale="0.001
                    0.001 0.001"/>
11          </geometry>
12      </xacro:macro>
13      <xacro:macro name="z_rot">
14          <axis xyz="0 0 1"/>
15      </xacro:macro>
16
17      <!-- Select one and only one of the following include files -->
18      <!-- 1) Standard parts of Bioloid Comprehensive humanoid with 18 AX-12+ servos, which
             have been replaced in custom version -->
19      <!--    When using this URDF, ensure a transform from odom to base_link is being
            published -->
20  <!--     <xacro:include filename="$(find bioloid_master)/urdf/bioloid_std_parts.xacro"/>
        -->
21          <xacro:include filename="bioloid_torso_std_part.xacro"/>
22  <!-- 2) Custom parts of modified Bioloid Comprehensive humanoid, with CM-5 replaced by PC
         and custom sensors -->
23      <!--    When using this URDF, ensure a transform from odom to imu_link is being
            published -->
24  <!--
25      <xacro:include filename="$(find bioloid_master)/urdf/bioloid_custom_parts.xacro"/>
26  -->
27
28      <!-- Import gazebo_ros_control plugin -->
29      <xacro:include filename="bioloid_without_legs_gazebo.xacro"/>
30
31      <!-- <xacro:macro name="shoulder_swing_joint" params="prefix reflect x"> -->
32      <xacro:macro name="shoulder_swing_joint" params="prefix reflect">
33          <joint name="${prefix}_shoulder_swing_joint" type="revolute">
34              <parent link="torso_link"/>
35              <!-- <origin xyz="${reflect*(-0.032)} 0 0" rpy="0 ${reflect*(-pi/2)} ${x*pi}"
                    /> -->
36              <origin xyz="${reflect*(-0.032)} 0 0" rpy="${reflect*(-pi/2)} 0 ${reflect*(pi
                    /2)}"/>
37              <xacro:z_rot/>
38              <child link="${prefix}_shoulder_swing_link"/>
39              <limit lower="${-pi}" upper="${pi}" effort="10" velocity="1.0"/>
40          </joint>
41      </xacro:macro>
42      <xacro:macro name="shoulder_lateral_joint" params="prefix reflect">
43          <joint name="${prefix}_shoulder_lateral_joint" type="revolute">
44              <parent link="${prefix}_shoulder_swing_link"/>
45              <!-- <origin xyz="${reflect*(-0.014)} 0 ${reflect*0.044}" rpy="${-pi/2} 0 0"/
                    > -->
46              <origin xyz="${reflect*(-0.014)} 0 ${reflect*0.044}" rpy="${-pi/2} ${pi/2} 0"
                    />
47              <xacro:z_rot/>
48              <child link="${prefix}_upper_arm_link"/>
49              <limit lower="${-pi}" upper="${pi}" effort="10" velocity="1.0"/>
```

```
50          </joint>
51      </xacro:macro>
52      <xacro:macro name="elbow_joint" params="prefix reflect">
53          <joint name="${prefix}_elbow_joint" type="revolute">
54              <parent link="${prefix}_upper_arm_link"/>
55              <origin xyz="${reflect*-0.0675} 0 0" rpy="0 0 0"/>
56              <xacro:z_rot/>
57              <child link="${prefix}_lower_arm_link"/>
58              <limit lower="${-1.77}" upper="${1.77}" effort="10" velocity="1.0"/>
59          </joint>
60      </xacro:macro>
61      <xacro:macro name="hand_joint" params="prefix">
62          <joint name="${prefix}_hand_joint" type="fixed">
63              <parent link="${prefix}_lower_arm_link"/>
64              <origin xyz="0 0 0" rpy="0 0 0"/>
65              <child link="${prefix}_hand_link"/>
66          </joint>
67      </xacro:macro>
68
69      <!-- World -->
70      <!-- Used for fixing robot to Gazebo 'base_link' -->
71      <link name="world"/>
72
73      <joint name="world_to_odom" type="fixed">
74          <parent link="world"/>
75          <child link="odom"/>
76          <origin xyz="0 0 0.3" rpy="0 0 0"/>
77        </joint>
78
79      <link name="odom"/>
80
81        <joint name="odom_to_imu" type="fixed">
82          <parent link="odom"/>
83          <child link="imu"/>
84          <origin xyz="0 0 0" rpy="0 0 0"/>
85        </joint>
86
87      <link name="imu"/>
88
89        <joint name="imu_to_base_link" type="fixed">
90          <parent link="imu"/>
91          <child link="base_link"/>
92          <origin xyz="0 0 0" rpy="0 0 ${pi/2}"/>
93        </joint>
94
95  <!--
96      <joint name="imu_to_base" type="fixed">
97          <parent link="imu"/>
98          <child link="base_link"/>
99          <origin xyz="0 0 0" rpy="0 0 0"/>
100       </joint>
101 -->
102
103
104     <!-- Base -->
105     <joint name="base_to_torso_joint" type="fixed">
106         <parent link="base_link"/>
107         <origin xyz="0 0 0" rpy="0 0 ${pi/2}"/>
108         <child link="torso_link"/>
109     </joint>
```

```
110
111     <link name="base_link">
112     </link>
113
114     <!-- Neck Joint-->
115   <joint
116     name="head_yaw_joint"
117     type="revolute">
118     <origin
119       xyz="0 0 0.015"
120       rpy="0 0 0" />
121     <parent
122       link="base_link" />
123     <child
124       link="neck_link" />
125     <limit lower="${-pi}" upper="${pi}" effort="10" velocity="1.0"/>
126     <axis
127       xyz="0 0 1" />
128     <limit
129       effort="0"
130       velocity="0" />
131   </joint>
132
133   <!-- Neck Link-->
134     <link
135     name="neck_link">
136     <visual>
137       <origin
138         xyz="0 0 0"
139         rpy="0 0 0" />
140       <geometry>
141         <mesh
142           filename="package://bioloid_example/mesh/neck_link.STL" />
143       </geometry>
144       <material
145         name="">
146         <color
147           rgba="0.1 0.1 0.1 1" />
148       </material>
149     </visual>
150   </link>
151
152     <!-- Head Link-->
153   <link
154     name="head_link">
155     <visual>
156       <origin
157         xyz="0 0 0"
158         rpy="0 0 0" />
159       <geometry>
160         <mesh
161           filename="package://bioloid_example/mesh/head_link.STL" />
162       </geometry>
163       <material
164         name="">
165         <color
166           rgba="0.5 0.5 0.5 1" />
167       </material>
168     </visual>
169   </link>
```

```
170
171   <!-- Head Joint-->
172   <joint
173     name="head_pitch_joint"
174     type="revolute">
175     <origin
176       xyz="0.008 0 0.053"
177       rpy="0 0 0" />
178     <parent
179       link="neck_link" />
180     <child
181       link="head_link" />
182       <limit lower="-1.3" upper="1.4" effort="10" velocity="1.0"/>
183     <axis
184       xyz="0 1 0" />
185     <limit
186       effort="0"
187       velocity="0" />
188   </joint>
189
190
191     <!-- Torso to right shoulder -->
192     <xacro:shoulder_swing_joint prefix="right" reflect="1"/>
193     <link name="right_shoulder_swing_link">
194         <visual>
195             <xacro:default_geometry cad_file="FP04-F1"/>
196             <origin xyz="-0.014 0 0.044" rpy="0 ${pi} ${-pi/2}"/>
197         </visual>
198         <collision name="right_shoulder_collision">
199             <origin xyz="-0.006 0 0.0375"/>
200             <geometry>
201                 <box size="0.039 0.052 0.036"/>
202             </geometry>
203         </collision>
204         <inertial>
205             <origin xyz="0 0 0"/>
206             <mass value="1.0"/>
207             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
208         </inertial>
209     </link>
210
211     <!-- Torso to left shoulder -->
212     <xacro:shoulder_swing_joint prefix="left" reflect="-1"/>
213     <link name="left_shoulder_swing_link">
214         <visual>
215             <xacro:default_geometry cad_file="FP04-F1"/>
216             <origin xyz="0.014 0 -0.044" rpy="0 0 ${pi/2}"/>
217         </visual>
218         <collision name="left_shoulder_collision">
219             <origin xyz="0.006 0 -0.0375"/>
220             <geometry>
221                 <box size="0.039 0.052 0.036"/>
222             </geometry>
223         </collision>
224         <inertial>
225             <origin xyz="0 0 0"/>
226             <mass value="1.0"/>
227             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
```

```
228         </inertial>
229     </link>
230 
231     <!-- Right shoulder to upper arm -->
232     <xacro:shoulder_lateral_joint prefix="right" reflect="1"/>
233     <link name="right_upper_arm_link">
234         <visual>
235             <xacro:default_geometry cad_file="AX-12"/>
236             <origin xyz="0 0 0" rpy="${pi} 0 ${pi/2}"/>
237         </visual>
238         <visual>
239             <xacro:default_geometry cad_file="BPF-BU"/>
240             <origin xyz="0 0 0.0185" rpy="${pi/2} 0 0"/>
241         </visual>
242         <visual>
243             <xacro:default_geometry cad_file="BPF-WA"/>
244             <origin xyz="0 0 0.019" rpy="${pi/2} 0 0"/>
245         </visual>
246         <visual>
247             <xacro:default_geometry cad_file="FP04-F3"/>
248             <origin xyz="-0.0415 0 0" rpy="0 0 ${-pi/2}"/>
249         </visual>
250         <visual>
251             <xacro:default_geometry cad_file="FP04-F2"/>
252             <origin xyz="-0.0675 0 0" rpy="0 ${pi/2} 0"/>
253         </visual>
254         <collision name="right_upper arm_collision_proximal">
255             <origin xyz="-0.0148 0 0"/>
256             <geometry>
257                 <box size="0.054 0.033 0.044"/>
258             </geometry>
259         </collision>
260         <collision name="right_upper arm_collision_distal">
261             <origin xyz="-0.0555 0 0"/>
262             <geometry>
263                 <box size="0.047 0.033 0.052"/>
264             </geometry>
265         </collision>
266         <inertial>
267             <origin xyz="0 0 0"/>
268             <mass value="1.0"/>
269             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
270         </inertial>
271     </link>
272 
273     <!-- Left shoulder to upper arm -->
274     <xacro:shoulder_lateral_joint prefix="left" reflect="-1"/>
275     <link name="left_upper_arm_link">
276         <visual>
277             <xacro:default_geometry cad_file="AX-12"/>
278             <origin xyz="0 0 0" rpy="0 0 ${pi/2}"/>
279         </visual>
280         <visual>
281             <xacro:default_geometry cad_file="BPF-BU"/>
282             <origin xyz="0 0 -0.0185" rpy="${-pi/2} 0 0"/>
283         </visual>
284         <visual>
285             <xacro:default_geometry cad_file="BPF-WA"/>
286             <origin xyz="0 0 -0.019" rpy="${-pi/2} 0 0"/>
```

```
287         </visual>
288         <visual>
289             <xacro:default_geometry cad_file="FP04-F3"/>
290             <origin xyz="0.0415 0 0" rpy="0 0 ${pi/2}"/>
291         </visual>
292         <visual>
293             <xacro:default_geometry cad_file="FP04-F2"/>
294             <origin xyz="0.0675 0 0" rpy="0 ${-pi/2} 0"/>
295         </visual>
296         <collision name="left_upper arm_collision_proximal">
297             <origin xyz="0.0148 0 0"/>
298             <geometry>
299                 <box size="0.054 0.033 0.044"/>
300             </geometry>
301         </collision>
302         <collision name="left_upper arm_collision_distal">
303             <origin xyz="0.0555 0 0"/>
304             <geometry>
305                 <box size="0.047 0.033 0.052"/>
306             </geometry>
307         </collision>
308         <inertial>
309             <origin xyz="0 0 0"/>
310             <mass value="1.0"/>
311             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
312         </inertial>
313     </link>
314
315     <!-- Right upper arm to lower arm -->
316     <xacro:elbow_joint prefix="right" reflect="1"/>
317     <link name="right_lower_arm_link">
318         <!-- Motor 5 -->
319         <visual>
320             <xacro:default_geometry cad_file="AX-12"/>
321             <origin xyz="0 0 0" rpy="${pi} 0 ${pi/2}"/>
322         </visual>
323         <visual>
324             <xacro:default_geometry cad_file="BPF-BU"/>
325             <origin xyz="0 0 0.0185" rpy="${pi/2} 0 0"/>
326         </visual>
327         <visual>
328             <xacro:default_geometry cad_file="BPF-WA"/>
329             <origin xyz="0 0 0.019" rpy="${pi/2} 0 0"/>
330         </visual>
331         <visual>
332             <xacro:default_geometry cad_file="FP04-F3"/>
333             <origin xyz="-0.0415 0 0" rpy="0 0 ${-pi/2}"/>
334         </visual>
335         <collision name="right_lower_arm_collision">
336             <origin xyz="-0.0135 0 0"/>
337             <geometry>
338                 <box size="0.051 0.033 0.044"/>
339             </geometry>
340         </collision>
341         <inertial>
342             <origin xyz="0 0 0"/>
343             <mass value="1.0"/>
344             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
```

```
345         </inertial>
346     </link>
347 
348     <!-- Left upper arm to lower arm -->
349     <xacro:elbow_joint prefix="left" reflect="-1"/>
350     <link name="left_lower_arm_link">
351         <!-- Motor 6 -->
352         <visual>
353             <xacro:default_geometry cad_file="AX-12"/>
354             <origin xyz="0 0 0" rpy="0 0 ${pi/2}"/>
355         </visual>
356         <visual>
357             <xacro:default_geometry cad_file="BPF-BU"/>
358             <origin xyz="0 0 -0.0185" rpy="${-pi/2} 0 0"/>
359         </visual>
360         <visual>
361             <xacro:default_geometry cad_file="BPF-WA"/>
362             <origin xyz="0 0 -0.019" rpy="${-pi/2} 0 0"/>
363         </visual>
364         <visual>
365             <xacro:default_geometry cad_file="FP04-F3"/>
366             <origin xyz="0.0415 0 0" rpy="0 0 ${pi/2}"/>
367         </visual>
368         <collision name="left_lower_arm_collision">
369             <origin xyz="0.017 0 0"/>
370             <geometry>
371                 <box size="0.058 0.033 0.044"/>
372             </geometry>
373         </collision>
374         <inertial>
375             <origin xyz="0 0 0"/>
376             <mass value="1.0"/>
377             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
378         </inertial>
379     </link>
380 
381     <!-- Right lower arm to hand -->
382     <xacro:hand_joint prefix="right"/>
383     <link name="right_hand_link">
384         <visual>
385             <xacro:default_geometry cad_file="FP04-F11"/>
386             <origin xyz="-0.0415 0 0" rpy="${-pi} ${pi/2} 0"/>
387         </visual>
388         <collision name="right_palm_collision">
389             <origin xyz="-0.039 0 0"/>
390             <geometry>
391                 <box size="0.0131 0.0265 0.044"/>
392             </geometry>
393         </collision>
394         <collision name="right_fingers_collision">
395             <origin xyz="-0.0701 -0.0077 0"/>
396             <geometry>
397                 <box size="0.0575 0.0175 0.035"/>
398             </geometry>
399         </collision>
400         <inertial>
401             <origin xyz="0 0 0"/>
402             <mass value="1.0"/>
403             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
```

```
                0.001"/>
404         </inertial>
405     </link>
406 
407     <!-- Left lower arm to hand -->
408     <xacro:hand_joint prefix="left"/>
409     <link name="left_hand_link">
410         <visual>
411             <xacro:default_geometry cad_file="FP04-F11"/>
412             <origin xyz="0.0415 0 0" rpy="0 ${pi/2} 0"/>
413         </visual>
414         <collision name="left_palm_collision">
415             <origin xyz="0.039 0 0"/>
416             <geometry>
417                 <box size="0.0131 0.0265 0.044"/>
418             </geometry>
419         </collision>
420         <collision name="left_fingers_collision">
421             <origin xyz="0.0701 0.0077 0"/>
422             <geometry>
423                 <box size="0.0575 0.0175 0.035"/>
424             </geometry>
425         </collision>
426         <inertial>
427             <origin xyz="0 0 0"/>
428             <mass value="1.0"/>
429             <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="
                    0.001"/>
430         </inertial>
431     </link>
432 
433 </robot>
```