Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Salvador

30 de abril 2022

Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

Universidade Federal da Bahia - UFBA

Escola Politécnica

Programa de Pós-Graduação em Engenharia Elétrica

Orientador: Wagner Oliveira

Coorientador: Paulo César

Salvador

30 de abril 2022

Nestor Dias Pereira Neto
Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS/ Nestor Dias Pereira Neto. – Salvador, 30 de abril 2022-
44p. : il. (algumas color.) ; 30 cm.

Orientador: Wagner Oliveira

Dissertação (Mestrado) – Universidade Federal da Bahia - UFBA
Escola Politécnica
Programa de Pós-Graduação em Engenharia Elétrica, 30 de abril 2022.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

Nestor Dias Pereira Neto

Desenvolvimento de um co-processador de vídeo em FPGA para integração com o Robot Operating System - ROS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica.

Trabalho aprovado. Salvador, 24 de novembro de 2012:

Wagner Oliveira
Orientador

Professor
Convidado 1

Professor
Convidado 2

Salvador
30 de abril 2022

Resumo

Segundo a.o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex, abntex, editoração de texto.

Abstract

oihbqptipbõq4tnpot4photnj4yojnj4ynojp

Keywords: latex. abntex. text editoration.

Sumário

1 Introdução

Nos últimos anos novas técnicas para construção de robôs tem estado em muita evidência, em especial áreas como robótica móvel e robótica colaborativa têm chamado bastante atenção dos pesquisadores. Uma das principais características dessas áreas é a exigência de um alto grau de percepção do ambiente que rodeia o robô, além de uma execução mais precisa em seus movimentos, isto devido ao fato de que as atividades desempenhadas por estes robôs estão exigindo um nível cada vez maior de interação com as atividades desempenhadas por seres humanos.

Este grau de precisão requerido no desenvolvimento de novos robôs exigem sistemas robóticos cada vez mais complexos que precisam de processadores igualmente mais poderosos, consequentemente demandando um maior consumo de energia. Este aumento de consumo, provoca uma verdadeira disputa entre poder de processamento e consumo, no momento do levantamento dos requisitos de um novo projeto, o que pode vir a ser um problema principalmente em sistemas que fazem uso de baterias.

O FPGA é uma excelente alternativa para resolver este impasse, por oferecer um aumento do poder de processamento associado a um baixo consumo de energia. O potencial que os FPGAs possuem para melhorar o desempenho de sistemas computacionais já é bem conhecido há algum tempo. Myer-Baese (2014) descreve algumas vantagens dos FPGAs modernos para uso em processamento digitais de sinais, como as cadeias de fast-carry usadas para implementar MACs de alta velocidade e o paralelismo tipicamente encontrado em designs implementados em FPGA.

Por essas características o FPGA necessita de frequências menores de trabalho para alcançar desempenho equivalente ou superior às soluções baseadas unicamente em processadores, diminuindo a dissipação térmica, e consequentemente, necessitando um consumo de energia consideravelmente menor. Todas estas características oferecidas pelo hardware configurável o tornam um recurso bastante interessante para o uso em projetos de robótica.

Entretanto, apesar de oferecer grandes vantagens, as facilidades de desenvolvimento encontradas em aplicações que fazem uso de softwares não estão disponíveis na mesma proporção no mundo do hardware configurável. A maior dificuldade no desenvolvimento de soluções que fazem uso do FPGA tornam os seus projetos mais longos e requerem a necessidade de mão de obra extremamente especializada, aumentando consideravelmente o custo final de projetos com FPGA. Isso faz com que o seu uso em projetos de robótica seja pouco usado ou até mesmo desencorajado.

Atualmente o *framework* ROS está se consolidando como o padrão na criação de

novas plataformas robóticas, tanto no desenvolvimento de manipuladores colaborativos quanto na robótica móvel. O objetivo do ROS é facilitar a elaboração de novos robôs, através de um conjunto completo de ferramentas para desenvolvimento, como *drivers* para sensores e atuadores, bibliotecas e principalmente reuso de código. Agrupar inúmeros "blocos" de softwares usados em robótica, fornecer drivers para hardwares específicos (sensores e atuadores), gerenciar troca de mensagens entre os nós que fazem parte do sistema, são as função do ROS.

Estas características fazem com que o ROS seja reconhecido com um pseudo sistema operacional (PYO et al., 2017). Dessa maneira o ROS se tornou muito ágil no desenvolvimento de novas aplicações para robótica. Usando aplicações já desenvolvidas e testadas por outros desenvolvedores, podem criar novos sistemas completos apenas gerenciando estas aplicações na estrutura interna do ROS. Essa abordagem fez com que o número de pacotes para o ROS cresça em uma taxa muito rápida, desde o ano de seu lançamento, em 2007, até 2012 o ROS aumentou de 1 para 3699 pacotes (YAMASHINA et al., 2015).

Aproveitar as facilidades de desenvolvimento proporcionadas pelo ROS em conjunto com o alto poder de processamento e baixo consumo que o FPGA oferece, seria um cenário ideal no desenvolvimento de novas aplicações com robôs. Para isso, precisamos estabelecer uma conexão com uma taxa de transferência de dados alta o suficiente para não influenciar de forma negativa no tempo de processamento e, que torne relativamente fácil seu uso por desenvolvedores especializados em robótica, mas sem grande experiência em FPGA. Portanto, este trabalho tem como objetivo estabelecer uma comunicação de alto desempenho entre ROS e um FPGA para que se possa aproveitar o melhor das duas tecnologias em projetos de robótica.

explicar a parte da comunicação entre o computador/ros e o SoC melhorar a comunicação, comunicação eficiente, pacote pronto e de fácil integração com qualquer sistema ros, O mais genérico possível para se enquadrar a qualquer projeto é que o desenvolvedor tenha interesse em incluir um FPGA ao sistema

- **Como estabelecer a comunicação entre o ROS e um sistema de processamento auxiliar embarcado em um FPGA?**

Este problema é o que o trabalho vai tentar resolver, possibilitando assim, o uso de aceleração por hardware através do fpga, ser incluída no desenvolvimento de novos projetos de robótica. Projetistas especializados em robótica poderão aproveitar dos benefícios do uso do hardware dedicado em seus projetos e profissionais que trabalham com descrição de hardware poderão desenvolver novas soluções para problemas de robótica de forma modularizada.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver uma solução para estabelecer comunicação entre *Field Programmable Gate Array - FPGA*, configurado como um co-processador de vídeo.

1.1.2 Objetivos Específicos

- Estudar teoria dos assuntos relevantes ao projeto: Verilog HDL, embedded linx, Cyclone V, TCP/IP Stack, ROS;
- Estudar conceito de programação de redes usando sockets em liguagem C++ e detalhes dos protocolos da rede TCP/IP usada para comunicação interna dos nós e serviços ROS;
- Implementar distribuição embedded linux para processador ARM embarcado no SoC Cyclone V da Intel;
- Estabelecer comunicação entre o ROS e o Cyclone V, através da tecnologia Gigabit Ethernet;
- Desenvolver aplicação em Verilog para testar comunicação;
- Avaliar a performance da rede entre o computador e o protótipo após a inclusão do FPGA ao sistema.

1.2 Organização

No Capítulo 1 é apresentada a introdução do texto, com uma breve contextualização do tema, além do problema ao qual o trabalho se propõe a resolver. Neste capítulo também são apresentados a justificativa e os objetivos gerais e específicos. Na sequência o texto é dividido em três partes, são elas: Referencial teórico, Desenvolvimento e Resultados.

No Referencial teórico temos dois capítulos onde são apresentadas as tecnologias utilizadas no trabalho, no Capítulo 2 é discorrido sobre o System on Chip - SoC, utilizado para o desenvolvimento do trabalho e no Capítulo 3 é falado sobre o Robot Operating System - ROS.

Na segunda parte deste trabalho, chamada de Desenvolvimento, são explicadas as etapas do desenvolvimento. A arquitetura do sistema é explicada no Capítulo 4, na sequência, no Capítulo 5 é explicado o Cliente e o servidor é descrito no Capítulo 6

Na última parte deste documento são discutidos os Resultados alcançados, que são apresentados no Capítulo 7. Já O Capítulo 8 é apresentada a conclusão desta pesquisa e no Capítulo 9 são apresentadas algumas sugestões de trabalhos futuros

Parte I

Referenciais teóricos

2 Cyclone V: System on a Chip - SoC

2.1 Field Programmable Gate Array - FPGA

2.2 Hard processor ARM

2.2.1 Embedded Linux

2.3 Kit de desenvolvimento DE10-nano

3 Robot Operating System - ROS

> "O Robot Operating System (ROS) é um conjunto de bibliotecas de software e ferramentas que te auxiliam na construção de aplicações em robótica. De drivers ao estado da arte de algoritmos, e com poderosas ferramentas de desenvolvimento, ROS tem o que você precisa para seu próximo projeto de robótica. E tudo é open source (ROS, 2011)"

O ROS foi idealizado com o objetivo de ser um ambiente completo, de código aberto, para elaboração de sistemas robóticos. Largamente aceito e amplamente utilizado atualmente, os desenvolvedores se beneficiam da alta qualidade de código proporcionado pelo grande número de usuários e plataformas que aproveitam-se do ROS em seus projetos (ROS, 2018). Uma ampla variedade de sensores e atuadores empregados na robótica também seguem essa tendência e oferecem suporte ao ROS através de seus drivers. O ROS fornece abstração de hardware, controle de baixo nível para dispositivos, funcionalidades e bibliotecas de uso comum, passagem de mensagens entre processos e gerenciamento de pacotes (MAHTANI et al., 2016). Por esses motivos, o ROS é conhecido como um meta sistema operacional para robôs. Neste capítulo será apresentado o ROS e as vantagens do seu uso na concepção de novas plataformas robóticas.

A cada ano o ROS vem se consolidando como o framework padrão para o desenvolvimento de novos projetos de robótica, apesar de já possuir este status, o ROS possui uma história relativamente curta. O ROS teve início no Laboratório de Inteligência Artificial de Standord e em 2007 a companhia Willow Garage assumiu o seu desenvolvimento. A Willow Garage é uma empresa que desenvolve robôs pessoais e de serviços. Ela também é responsável pelo desenvolvimento de suporte a *Point Cloud Library (PCL)*, que é uma biblioteca de software largamente usada para processamento de nuvens de pontos. Em Janeiro de 2010 a primeira versão do ROS foi lançada, desde então muitas outras versões foram lançadas. O ROS está sob as licenças BSD 3-Clause e Apache 2.0, que permite qualquer um modificar, reusar e distribuir códigos ROS (PYO et al., 2017). Atualmente o ROS se encontra com uma versão estável do ROS2 e o ROS uma tem seu fim marcada para maio de 2015.

3.1 Um sistema operacional para robôs

De forma simplificada um sistema operacional tem como objetivo gerenciar os recursos do sistema computacional, ele atua como uma ponte entre esses recursos e o seu usuário, ou seja, o sistema operacional se faz necessário para disponibilizar às aplicações os recursos funcionais do sistema de forma padronizada, se tornando uma verdadeira camada

de abstração entre os programas e o hardware. Windows, Ubuntu, para computadores pessoais e Android para smartphones, são exemplos de sistemas operacionais populares.

A comunidade da robótica em todo mundo tem feito grande progresso nos últimos anos. Hardware confiáveis e com menor custo têm sido ofertados em um nível nunca encontrado no passado, desde robôs móveis terrestres, passando por drones e até mesmo robôs humanóides estão disponíveis no mercado com relativa facilidade. O que pode ser até mais impressionante, a comunidade também tem desenvolvido algoritmos que permitem que estes robôs possuam um nível crescente de autonomia. Apesar desse rápido progresso, o desenvolvimento de robôs ainda representam um desafio para os desenvolvedores de softwares e grande parte desse desafio se deve a falta de padronização de um software específico para robótica, ou até mesmo um sistema operacional dedicado para robôs, como podemos encontrar em outros nichos, como os PCs e smartphones e o ROS tenta preencher essa lacuna.

O nome ROS vem da abreviação de Robot Operating System, mas seria o ROS um sistema operacional para robôs? Ele fornece abstração de hardware, controle de baixo nível para dispositivos, implementações de funcionalidades de uso comum, troca de mensagens entre processos, até um sistema de gerenciamento de pacotes. Além destas características o ROS está equipado com bibliotecas e ferramentas para escrever, compilar e rodar seus códigos (ROS, 2018).

Apesar de todos os atributos que o caracterizam como um sistema operacional, o ROS não é um sistema operacional convencional, por ainda precisar rodar em um outro sistema operacional previamente instalado, o que faz com que o ROS seja conhecido como um meta-sistema operacional. Antes de ter o ROS em execução no robô é necessário instalar um sistema operacional, como por exemplo o Ubuntu. Com a distribuição Linux rodando é possível executar a instalação completa do ROS, sendo assim, todos os recursos fornecidos por um sistema operacional convencional podem ser utilizados pelo ROS, como sistema de gerenciamento de processos, sistema de arquivos, interface do usuário, compiladores entre outros.

Para complementar esses recursos básicos do sistema operacional o ROS fornece funcionalidades específicas para o uso na robótica, tal como libraries para transmissão e recepção de dados para uma variedade de hardwares comumente utilizados em sistemas robóticos. Esse tipo de software é conhecido como middleware ou framework. Como pode ser visto na Figura 1, o ROS é o sistema auxiliar para controlar atuadores e sensores, com um nível de abstração de hardware dando suporte para o desenvolvimento de novas aplicações de robótica em sistemas operacionais convencionais.

Figura 1 – ROS um meta-sistema operacional

Fonte: Pyo et al. (2017)

3.2 Vantagens do ROS

Até agora descrevemos o ROS como uma ferramenta ideal para o desenvolvimento de novos projetos de robótica, apesar disto, aprender a usar um novo framework é uma atividade árdua, principalmente um tão abrangente e complexo como o ROS. Esse tipo de questão deve ser levada em consideração na escolha das ferramentas no inicio de um novo projeto.

O objetivo do ROS não é ser um framework com o maior número de recursos, seu principal objetivo é oferecer o máximo de reuso de softwares usados no desenvolvimento de novos robôs. O ROS é um framework de computação distribuída, isso que dizer que os seus processos podem ser projetados individualmente e podem ser integrados ao sistema livremente e em tempo de execução. Estes processos podem ser agrupados em pacotes, facilitando o compartilhamento e a sua distribuição. Outra iniciativa para incentivar a colaboração e o compartilhamento de código são os repositórios oficiais do ROS, que estão disponíveis de forma livre.

A seguir serão listadas algumas características que incentivam a colaboração da comunidade e são responsáveis pelo sucesso do ROS:

- **Recursos Nativos:** O ROS oferece, de forma nativa, muitos recursos prontos, testados e validados pela comunidade. Podemos citar como exemplos o *Simultaneous Localization and Mapping (SLAM)* e o *Adaptive Monte Carlo Localization (AMCL)* que são usados para navegação autônoma de plataformas robóticas móveis, outro pacote oferecido pelo ROS é o MoveIt,pacote usado para planejamento de movimento de manipuladores. Estes recursos podem ser usados sem nenhum problema e são

altamente configuráveis, podendo ser adaptados em vários modelos de robôs e atender a inúmeras aplicações.

- **Ferramentas de desenvolvimento:** O ROS é disponibilizado com uma grande quantidade de ferramentas para debugging, visualização, incluindo ferramentas para simulação. Algumas das ferramentas open source mais poderosas para visualização, debugging e simulação, respectivamente, rqt_gui, RViz e Gazebo, são nativas do ambiente ROS.

- **Suporte sensores e atuadores:** Muitos dos sensores e atuadores usados na robótica já são suportados pelo ROS e o número de dispositivos compatíveis aumenta a cada ano. Algumas companhias se beneficiam pelo fato de muitos desses dispositivos fazerem uso de hardware aberto e o software existente também pode ser reutilizado a custo zero. Fazendo com que o processo de desenvolvimento de novos hardware periféricos usados na robótica também acelere. Sensores como Velodyne-LIDAR, Laser scanners e atuadores como os servos Dynamixel, podem ser integrados ao ROS sem nenhum impedimento.

- **Múltiplas linguagens:** O framework ROS pode ser programado em algumas linguagens de programação modernas. Pdemos escrever código eficientes em C ou C++ e outra aplicação ser escrita em python. O ROS possui bibliotecas clientes para C/C++, Python, Java e Lisp. Este tipo de flexibilidade não é comum em outros frameworks.

3.2.1 Computacão distribuida

Grande parte de robôs modernos possuem diferentes unidades de processamentos, que podem estar localizados em diferentes computadores. Desde sensores microprocessados até mesmo controles específicos para motores, podem possuir suas próprias unidades computacionais. Mesmo possuindo apenas um computador, dividir o processamento em processos individuais específicos para cada função e fazer com que eles trabalhem em conjunto para se afim de resolverem um problema maior é uma excelente abordagem para a arquitetura de robôs modernos. Além disso, múltiplos robôs podem trabalhar de forma colaborativa dividindo atividades entre eles, ou até mesmo interagindo com seres humanos que poderiam enviar comandos através de um computador ou celular. Em todos estes casos é necessário a comunicação entre processos, que podem ou não estarem no mesmo computador.

O ROS é um Inter-process communication framework e de fato ele usa uma rede TCP/IP para realizar essa comunicação entre processos. Ele usa sockets TCP/IP para transportar dados entre os processos. Esta abordagem proporciona grande flexibilidade na troca de mensagem, processos podem se comunicar com outros mesmo se eles não

estiverem na mesma máquina, ou até mesmo em robôs separados, bastando para isso, todas as máquinas que compartilham processos estarem na mesma rede. Isso faz com que até mesmo processos rodando na internet possam participar da comunicação e realizar uma parte do processamento de um robô.

3.2.2 Reuso de Software

O uso do ROS pode diminuir a necessidade de implementar algoritmos que já foram testados e validados por outros pesquisadores. Algoritmos de navegação, planejamento de rotas, mapeamento entre outro, são usados em diferentes projetos, e o ROS permite que estes algoritmos sejam reaproveitados de duas maneiras possíveis:

- **Pacotes padrões:** São pacotes de softwares de importantes algoritmos usados na robótica ou mesmo drivers de dispositivos comuns na robótica, que já foram implementados e testados.
- **Interface de troca de mensagens:** A interface utilizada pelo ROS vem se tornando um padrão na comunicação entre processos em robôs.

Nos repositórios oficiais do ROS estão disponíveis centenas de pacotes públicos que utilizam a interface de troca de mensagens padronizada do ROS possibilitando uma redução significativa do esforço necessário para o desenvolvimento de uma lógica para integrar estes pacotes ao seu sistema. Logo desenvolvedores que usam o ROS podem concentrar mais tempo e esforços no desenvolvimento de suas novas ideias, aproveitando algoritmos consolidados dos repositórios oficiais do ROS, sem a necessidade de reescrevê-los para adaptá-los ao seu projeto.

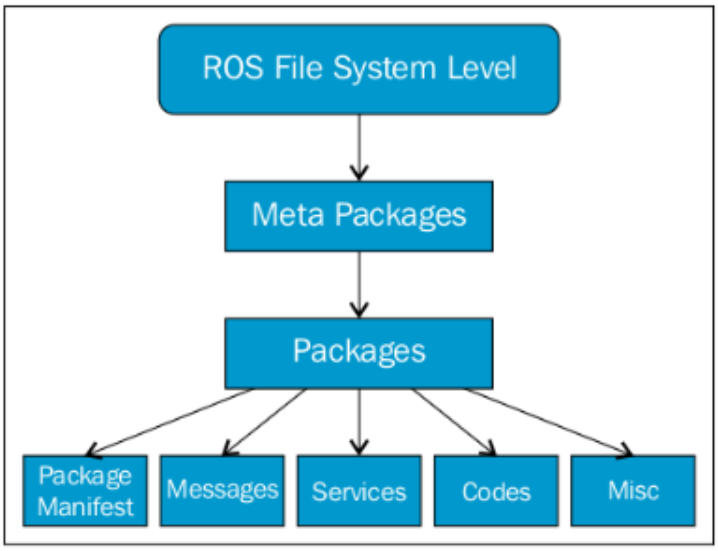
3.3 Sistemas de aquivos

Como era de se esperar de um sistema operacional, o ROS também possui um sistema de arquivos padronizado. É de extrema importância para o desenvolvimento de novas aplicações conhecer a organização dessa estrutura de arquivos. A Figura 2 apresenta um diagrama de blocos do sistema de arquivo do ROS.

A seguir são apresentados a definição de cada bloco da estrutura de arquivos do ROS:

- **Pacotes:** Pacotes são a unidade de software principal do ROS. Um pacote pode conter nós, dependências, bibliotecas, datasets, arquivos de configuração, ou qualquer coisa que seja útil organizar de forma agrupada (ROS, 2019b).

Figura 2 – Sistema de arquivos ROS



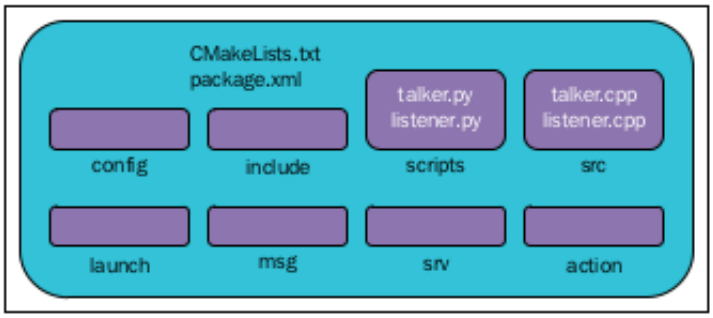
Fonte: Joseph (2015)

- **Manifesto do pacote:** O arquivo package.xml é conhecido como manifesto do pacote, ele fornece os metadados a respeito do pacotes, incluindo nome, versão, descrição, licença, dependências e outras informações. Os padrões do package.xml são definidos no REP-0127.
- **Metapackages:** Metapackages são pacotes especializados que serve apenas como representante de um grupo de outros pacotes relacionados entre si.
- **Meta packages manifest:** O manifesto de um metapackage é semelhante ao manifesto de um pacote comum, a diferença entre eles é que no manifesto devemos incluir as dependências encontradas no mesmo repositório do meta package
- **Message (msg) types:** É o arquivo de descrição de um tipo de mensagem, é armazenado no diretório my_package/msg/MyMessageType.msg e define toda a estrutura de dados enviados a partir deste tipo de mensagem.
- **Service (srv) types:** É o arquivo de descrição de um tipo de mensagem, é armazenado no diretório my_package/srv/MyServiceType.msg e define toda a estrutura de dados enviados a partir deste tipo de serviço.
- **Repositórios:** Pacotes ROS são compartilhados usando algum tipo de Version Control System (VCS), como o gti. Cada repositório pode conter apenas um pacote ou um metapackage

3.3.1 Pacotes ROS

A unidade básica para configuração do software no ROS é conhecida como pacote, isso quer dizer que todas as aplicações desenvolvidas para ROS são estruturadas como um pacote. Na Figura XX podemos ter uma visão da estrutura típica de uma pacote ROS

Figura 3 – Estrutura típica de um pacote ROS ROS



Fonte: Joseph (2015)

Nos pacotes estão contidos um ou mais nós, como são chamadas as unidades de processamento no ROS, ou podem incluir arquivos de configuração para a execução de nós de outros pacotes. Existem milhares de pacotes ROS oficiais e uma quantidade ainda maior de pacotes desenvolvidos por seus usuários.

Cada metapackage possui um arquivo chamado package.xml, este arquivo é responsável por reunir informações importantes sobre o pacote, nele podemos encontrar o nome do pacote, seu autor, dependências e licença de uso. O sistema de compilação do ROS é o Catkin, ele usa o CMake e o arquivo CMakeLists.txt que deve estar dentro da pasta de cada pacote com as suas instruções de compilação.

Os pacotes são as menores unidades que podem ser compiladas no ROS, são também a maneira com que podemos organizar o software para ser lançado. No caso dos pacotes oficiais do ROS, por exemplo, existe um pacote Debian, que são os pacotes utilizados pelo Ubuntu, para cada pacote ROS. Apesar do conceito ser semelhante, e mesmo que ao instalar um pacote Debian, você possa incluí-lo à sua lista de pacotes ao ROS instalado no sistema, os pacotes debian e ROS não são equivalentes.

A principal função dos pacotes é ser uma maneira funcional, de fácil configuração e um caminho descomplicado para possibilitar o reuso de software. De maneira geral o pacotes ROS devem conter funcionalidades suficientes para serem úteis, mas não muito para torná-los muito grande e confuso, tornando seu uso difícil por outro software

3.3.2 Mensagens ROS

Os nós do ROS podem publicar apenas dados de tipos predeterminados. Estes tipos são definidos usando uma linguagem de descrição de mensagens, conhecida como ROS messages. Esta simples descrição permite que as ferramentas do ROS gerem automaticamente o código fonte para mensagens para todas as linguagens aceitas no ROS. As descrições das mensagens são armazenadas no arquivo .msg localizada no subdiretório msg/ dentro de um pacote ROS.

Existem dois segmentos em um arquivo .msg, são eles: Fields e constants. Filds são os dados que serão enviados dentro da mensagens. Constants define os valores, ou tipo de dados, que poderão ser usados em cada campo. Os tipos de mensagens são referenciados a partir do nome do seu respectivo pacote, como ilustração, podemos usar o arquivo de descrição geometry_msgs/msg/Twist.msg, que será referencido como geometry_msgs/Twist.

O ROS possui um grande número de mensagens predefinidas, o que não impede que o programador escreva seus próprios arquivos .msg com a descrição de uma mensagem específica para atender a sua aplicação. A Tabela 1 a seguir lista os tipos de dados padrão do ROS que podem ser usados na criação de novas mensagens:

Tabela 1 – Estrutura típica de um pacote ROS

Primitive type	**Serialization**	**C++**	**Python**
bool(1)	unsigned8-bitint	uint8_t(2)	bool
int8	signed8-bitint	int8_t	int
uint8	unsigned8-bitint	uint8_t	int
int16	signed16-bitint	int16_t	int
uint16	unsigned16-bitint	uint16_t	int
int32	signed32-bitint	int32_t	int
uint32	unsigned32-bitint	uint32_t	int
int64	signed64-bitint	int64_t	long
uint64	unsigned64-bitint	uint64_t	long
float32	32-bitIEEEfloat	float	float
float64	64-bitIEEEfloat	double	float
string	asciistring(4)	std::string	string
time	Secs/nsecs signed 32bit ints	ros::Time	rospy.Time
duration	S ecs/nsecs signed 32bit ints	ros::Duration	rospy.Duration

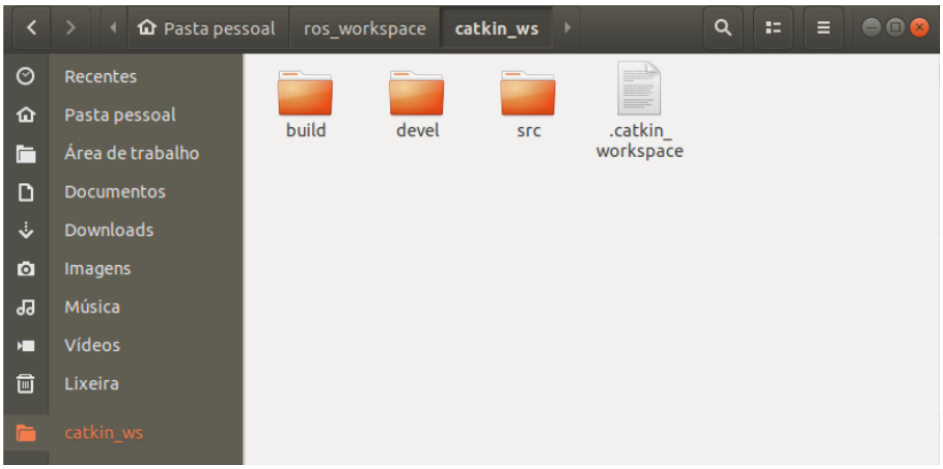
Fonte: Martinez e Fernández (2013)

3.3.3 Workspace ROS

De forma geral, o workspace é uma pasta no computador que contém todos os pacotes que estão sendo usados no desenvolvimento de uma nova aplicação. Estes pacotes contêm os arquivos fontes e o workspace fornece um local para que os pacotes sejam

compilados. O workspace se torna ainda mais útil quando é necessário compilar vários pacotes ao mesmo tempo, todos os pacotes poderão estar contidos no mesmo workspace centralizando todo processo de desenvolvimento. Não existe um diretor específico para que o workspace seja criado, logo, ele pode ser criado no local de preferência do desenvolvedor e sua equipe. Um workspace típico é mostrado na Figura 4.

Figura 4 – Estrutura típica de workspace ROS



Fonte: do Autor

Cada pasta dentro do workspace é um diferente espaço diferente, cada um com uma função específica:

- **O espaço das fontes:** No espaço das fontes, localizada na pasta src do workspace, estão localizados todos os pacotes do projetos. O arquivo mais importante dessa área do workspace é o CMakeLists.txt. Este arquivo é gerado na primeira vez em que o workspace é compilado
- **O espaço de compilação:** Localizado na pasta build,é neste local que são armazenadas as informações de cache, configurações e outros arquivos intermediário para que os pacotes sejam compilados corretamente.
- **O espaço de desenvolvimento:** Na pasta de que se encontra o espaço de desenvolvimento, aqui são armazenados os programas compilados. Assim você pode testar seus programas sem a necessidade de instalá-los no sistema.

Outra função importante do ROS que pode ser usada através de um workspace é o overlays. Se você tiver um pacote instalado em seu sistema, mas quiser testar uma versão mais atual do mesmo pacote, não é necessário instalar a versão mais atual, você pode baixar o código fonte do pacote dentro do seu workspace, após compilar o workspace o ROS entende que deverá usar a versão presente no workspace e não a versão instalada no sistema.

3.4 Sistemas computacional ROS Graph level

3.4.1 ROS node

No ROS os executáveis são chamados de nós, eles podem se comunicar com outros processos por meio dos tópicos, serviços ou pelo servidor de parâmetros. Os nós proporcionam grande modularidade aos sistemas robóticos que usam o ROS, isso faz com que o desenvolvimento destes sistemas se tornem bem mais simples.

Ao ser executado o nó possui um nome único no sistema. Através desse nome o nó pode se comunicar com outros nós. O ROS permite que os nós sejam escritos em diferentes linguagens de programação, as bibliotecas que fornecem a interface do ROS com uma linguagem específica é chamada de biblioteca cliente, as mais populares são: roscpp, para a linguagem C++ e a rospy para a linguagem Python.

Uma característica bastante interessante dos nós ROS é a possibilidade de parâmetros serem modificados no momento em que o nó é iniciado. Esta característica proporciona maior flexibilidade aos nós, já que o uso de parâmetros oferece a possibilidade de o código ser reconfigurado sem a necessidade de recompilar o código fonte, sendo assim podemos adaptar o nó a diferentes cenários sem conhecer detalhes de sua implementação.

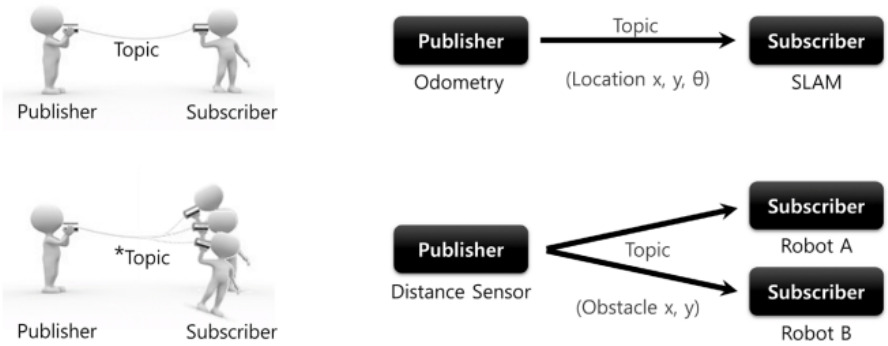
A powerful feature of ROS nodes is the possibility of changing parameters while you start the node. This feature gives us the power to change the node name, topic names, and parameter names. We use this to reconfigure the node without recompiling the code so that we can use the node in different scenes.

3.4.2 ROS TOPIC

Os tópicos são a maneira com que os nós enviam dados no ROS. Mesmo sem uma conexão direta entre dois nós, os tópicos podem ser transmitidos, isso faz com que a geração e o consumo de dados sejam desacoplados. Um tópico pode ser lido por vários nós e de maneira igual, também pode ser publicado por vários nós, mas não é uma boa prática um mesmo tópico ser publicado por nós distintos, isso pode causar conflitos nas informações enviadas. A Figura 5 ilustra o processo de comunicação com tópico.

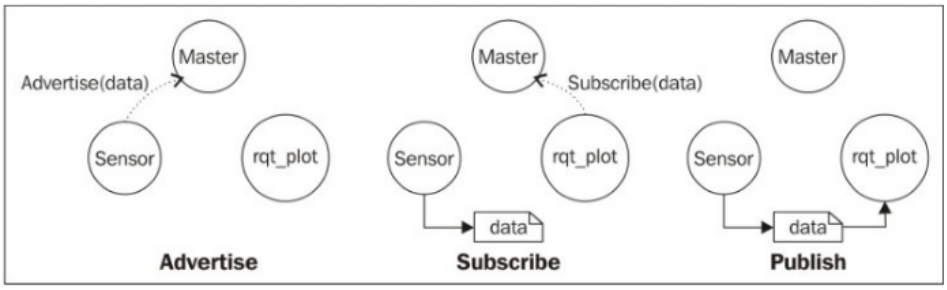
As mensagens ROS determinam os tipos de dados que poderão ser transportados através dos tópicos, isso faz com que os tópicos sejam fortemente tipados, consequentemente os mesmo tipo de mensagem deve ser obrigatoriamente o mesmo, para o nó que publica e para o nó que ler um determinado tópico. Sendo assim uma comunicação entre nós, por meio de tópicos apenas ocorrerá se ambos, nó de leitura e nó de publicação, estiverem registrados no ROS master com tópicos de mesmo nome e usando a mesma mensagem.

Figura 5 – Comunicação a partir de tópicos ROS



Fonte: Pyo et al. (2017)

Figura 6 – Gerenciamento de comunicação através do ROS master



Fonte: Mahtani et al. (2016)

3.4.3 ROS master

3.4.4 ROS Parameters

Parte II

Desenvolvimento

4 Arquitetura do sistema

Para conseguirmos estabelecer a comunicação entre o computador e o SoC precisaremos efetuar programação de sockets e bibliotecas específicas para trabalho em redes, desenvolver um pacote ROS para disponibilizar os dados recebidos através da interface de rede para os outros pacotes ROS do sistema robótico, além de um programa rodando no HPS do SoC para estabelecer esta comunicação entre a interface de rede da placa De10-nano e a aplicação sendo executada no FPGA. Já a aplicação que estará embarcada no FPGA contido no SoC deverá ser descrita por alguma linguagem de descrição de hardware, como por exemplo, verilog ou VHDL.

Todas essas etapas descritas anteriormente são necessárias para a construção completa do sistema proposto, o que torna o desenvolvimento da solução completa um desafio devido às diferentes ferramentas de software e hardware necessárias para sua conclusão. Tendo em vista este problema, a solução foi idealizada para conter o maior grau de modularidade possível, ou seja, cada uma dessas etapas será tratada com um projeto independente, apenas tendo cuidado para garantir a correta comunicação entre cada uma delas.

A grande vantagem que esse abordagem traz ao projeto é a possibilidade futura, de tanto a continuação do desenvolvimento como da manutenção do sistema, serem realizados por profissionais com background nas diferentes áreas envolvidas, sem a necessidade de se envolver no desenvolvimento de outros módulos. Sendo assim, um profissional especialista em descrição de hardware poderia se dedicar apenas à concepção da solução embarcada no FPGA, sem a necessidade possuir conhecimento em programação de redes.

4.1 Modelo cliente-servidor

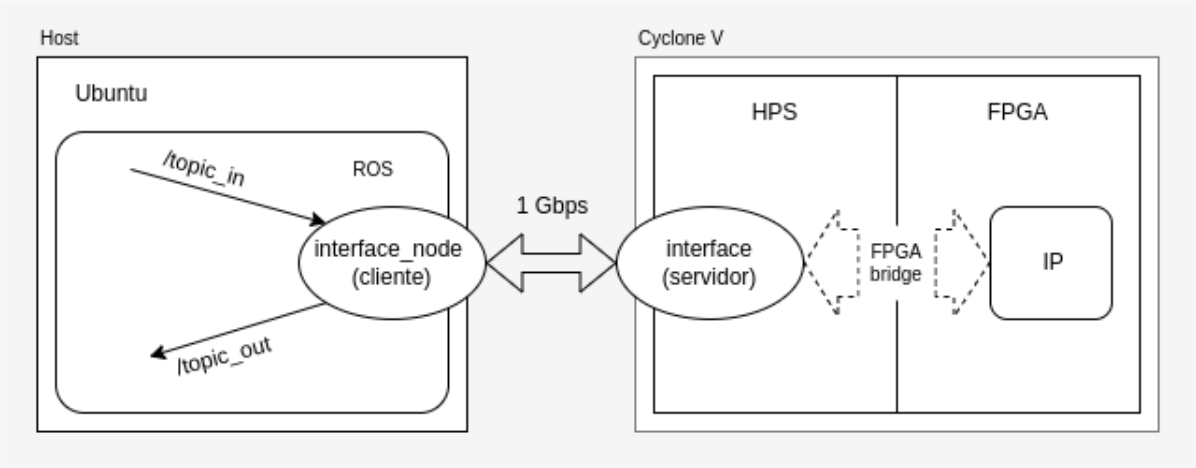
A comunicação entre o host, rodando o ROS, e a placa DE10-nano será estabelecida através de uma rede gigabit ethernet ponto a ponto, ou seja, o host e o SoC estarão conectados diretamente entre si. Desta maneira é possível obter o melhor desempenho da rede, alcançando as maiores taxas de transmissão de dados. Com o meio de comunicação definido é preciso definir também a arquitetura da comunicação, uma boa alternativa é o modelo cliente-servidor.

O modelo cliente-servidor é caracterizado por possuir uma estrutura que permite dividir o trabalho computacional entre os participantes da comunicação, isto é, entre o servidor, que é o encarregado de disponibilizar os recursos e serviços, e o cliente, que realiza as solicitações para os serviços disponíveis. Desta maneira tanto o cliente quanto

o servidor foram tratados como módulos independentes durante o desenvolvimento do trabalho. O uso do modelo cliente-servidor contribui de forma significativa para que o sistema alcance o máximo de modularização, essa abordagem facilita, entre outras coisas, a depuração e manutenção do código, o que proporciona mais agilidade e simplicidade no processo de desenvolvimento da solução.

Na Figura 7 podemos ter uma visão global do sistema, nela podemos ver cada etapa da comunicação.No lado do host, está instalado o ROS, nele também é onde o cliente será executado, assim sendo o cliente fica responsável por ler o tópico de entrada, fornecido por outro nó do sistema, realizar uma solicitação ao servidor enviando os dados já lidos. O servidor, por sua vez, aceita a solicitação do cliente, recebe os dados e os envia à aplicação embarcada no FPGA que os devolve após seu processamento. Para completar o ciclo o servidor retorna os dados processados ao cliente, que por sua vez, disponibiliza os dados já processados através do tópico de saída.

Figura 7 – Arquitetura geral



Fonte: do autor

4.2 Biblioteca de comunicação - libinterfacesocket

Para manter o padrão do desenvolvimentos dos códigos tanto do cliente quanto do servidor, foi desenvolvida uma classe, que fornece os métodos para a abertura da comunicação, além de métodos para envio e recebimento das mensagens através da rede gigabit ethernet. Essa classe foi desenvolvida como um módulo a parte e compilada como uma biblioteca estática, sendo assim, a partir do momento em que os métodos de comunicação estiverem testados e validado tanto o código do cliente quanto o do servidor poderão fazer uso desta biblioteca, eliminando assim a necessidade de reescrever uma parte do código código. Outra vantagem nessa abordagem é que ao manter o código desassociado

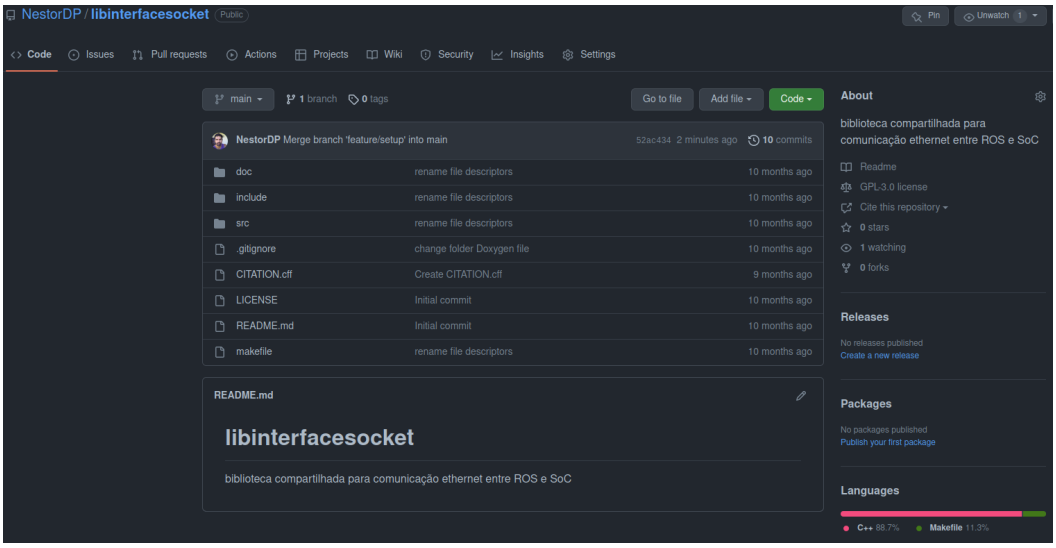
tanto do servidor como do cliente, nós possibilita fazer alterações ou correções de bugs, sem necessariamente realizar alterações nos códigos do servidor ou do cliente.

A programação da biblioteca foi realizada com base em sockets. Sockets são um caminho para conectar processos em uma rede de computadores. A conexão através de sockets entre nós em uma rede independe do protocolo. Um nó da rede ouve uma determinada porta para um IP específico esperando por o pedido de conexão do segundo nó, assim a conexão entre dois processos é estabelecida. O servidor é o nó que aguarda o pedido ser enviado pelo cliente.

A programação de sockets em C++ possibilita um alto nível de otimização da comunicação entre os processos, principalmente por se tratar de um modelo cliente-servidor onde só existirá a comunicação entre o servidor e apenas um cliente. Após implementar a comunicação entre o servidor e o cliente, poderá ser testadas novas técnicas de para otimizar o desempenho da rede possibilitando o aumento da taxa de transferência de dados entre o servidor e o cliente.

O código fonte da biblioteca pode ser encontrado no repositório no github (NETO, 2021a), que pode ser visto na figura 8, onde podemos observar a estrutura de arquivos da blibioteca. Vale frizar que, a libinterfacesocket possui um makefile para realizar o processo de compilação de forma automática. Assim podemos de forma simplificada compilar e instalar a biblioteca tanto no sistema do host onde será executado o cliente, quanto no sistema do HPS embarcado no SoC, onde o servidor estará rodando.

Figura 8 – Repositório libinterfacesocket



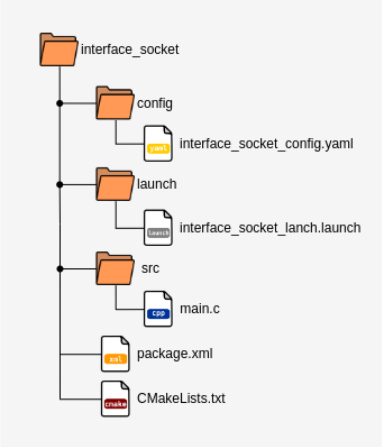
Fonte: do autor

5 Pacote ROS (cliente)

Como já foi mencionado anteriormente o cliente é um pacote ROS, sendo assim, deve-se levar em consideração, os conceitos de programação do framework ROS em seu desenvolvimento. As técnicas específicas de programação ROS usadas no cliente serão descritas com detalhes neste capítulo, além da explicação do seu funcionamento interno e do uso da libinterfacesocket.

No desenvolvimento de aplicações com ROS deve ser respeitada a estrutura de diretórios de um pacote ROS. Pacotes são a maneira com que os softwares são organizados no ROS, eles podem conter desde nós, que são a unidades de processamento do ROS, até mesmo bibliotecas ou módulos de softwares de terceiros. Os pacotes devem seguir uma estrutura padrão, por este motivo o código fonte do cliente foi organizado como é mostrado na Imagem 9 abaixo.

Figura 9 – Estrutura de diretórios pacote cliente



Fonte: do autor

A seguir será apresentada uma breve descrição de cada item do pacote:

config/interface_socket_config.yaml: Arquivo com o qual o usuário pode mudar alguns parâmetros de configuração da comunicação, como IP do servidor ou alguns outros parâmetros do tópico que será lido. Essa mudança pode ocorrer sem a necessidade de recompilar o código fonte, isso pode dar flexibilidade ao usuário do pacote durante o desenvolvimento de uma nova aplicação, que poderá ser configurada sem alterações no código fonte do nó.

launch/interface_socket.launch: Arquivo responsável chamar a execução do nó, além de carregar os parâmetros presentes no arquivo config.yaml no servidor de parâmetro do ROS.

Figura 10 – Configurações do cliente

```
server_ip: "127.0.0.1"
port: 4242

fps: 15
encoding: "rgb8"
height: 720
width: 1280
step: 3840
header.frame_id: "camera_link"
```

Fonte: Elaborado pelo autor

src/main.cpp: Código fonte do executável, ou seja, o código fonte do nó presente neste pacote.

package.xml: Manifesto do pacote é o arquivo que define as propriedades do pacote, como por exemplo, nome do pacote, autor, e dependências. Deve estar presente em todos os pacotes ROS (ROS, 2019a).

CmakeLists.txt: Contém instruções e diretivas para configuração do processo de compilação do pacote.

O pacote cliente disponibiliza apenas um executável, ou seja, apenas um nó. Assim que o no é executado ele realiza a requisição para estabelecer a conexão com o servidor, com a conexão estabelecida o cliente pode dar início à sua função, ler dados de um tópico e enviá-los ao servidor. Antes de serem enviados ao servidor os dados precisam ser preparados, informações importantes para o funcionamento interno do ROS, como campos dos cabeçalho, ou timestamp das mensagens, não serão enviados ao servidor. Ou seja, apenas os dados brutos são enviados, contribuindo para a diminuição de dados trafegando na rede. o

Figura 11 – Função Callback para o tópico lido pelo cliente

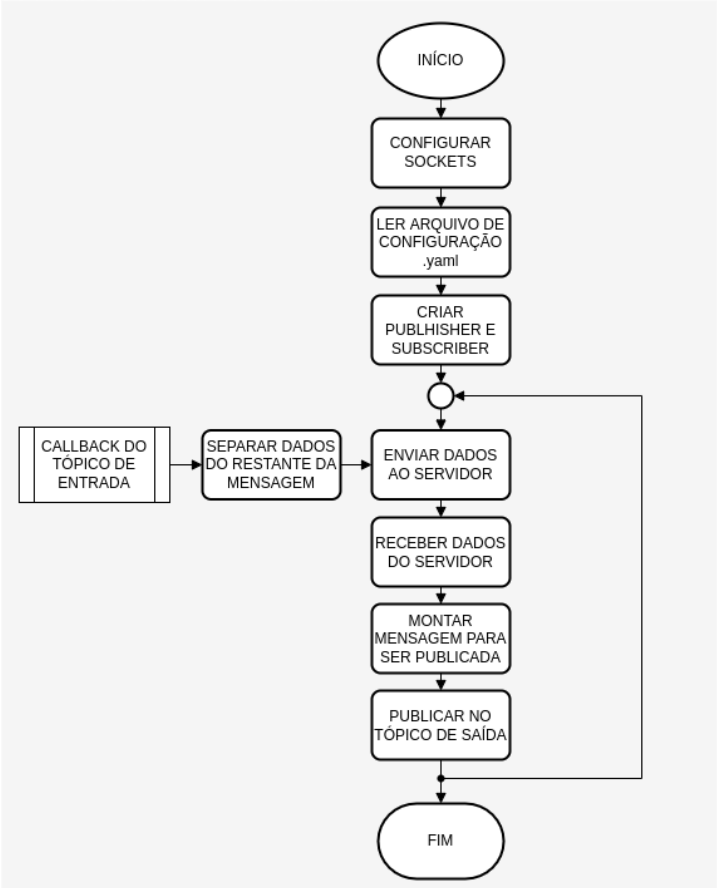
```
// Variavel que armazena apenas os dados contidos no
// topico lido pleo cliente
std::vector<uint8_t> dados_out(MSG_LEN);

void image_rawCallback
                (const sensor_msgs::Image::ConstPtr & image){
        // Apenas o campo data he armazenado na variavel
        dados_out = image->data;
}
```

Fonte: Elaborado pelo autor

Após os dados serem enviados ao servidor, o cliente deve ser capaz de recebê-los assim que o servidor os devolve. Neste ponto os dados já foram processados pelos circuitos do FPGA, mas como apenas os dados foram brutos foram enviados ao servidor, o cliente precisa montar novamente a mensagem, assim, o cliente poderá disponibilizar os dados processados a partir do tópico de saída, para que os outros nós do sistema possam utilizar esses dados em suas funcionalidades. O processo de funcionamento do cliente, descrito, pode ser inteiramente visualizado através do fluxograma. O processo descrito anteriormente é relativamente simples, como pode ser visto no fluxograma simplificado da Figura 12

Figura 12 – Fluxograma pacote cliente



Fonte: do autor

5.1 Outras alternativas

Se seu objetivo for apenas estabelecer a comunicação entre dois dispositivos, já existe um pacote ROS pronto que atenderá a sua necessidade. O ROS serial é uma pacote que tem como principal objetivo permitir a comunicação entre o ROS e outro dispositivo que possua uma porta serial ou uma interface de rede (FERGUSON, 2018). Sendo assim o ROS serial seria uma opção genérica, que não depende do dispositivo de hardware. Além

do ROS serial, foi encontrado um trabalho em que os autores estabeleceram a comunicação entre o ROS e um FPGA.

A grande diferença deste trabalhos para o proposto é a busca por No trabalho de Yamashina et al. (2005), são demonstradas três técnica para realizar a conexão entre o FPGA e o ROS, que se diferem da proposta por essa pesquisa

Existem alternativas prontas para fazê-lo, como por exemplo o ROS serial, pacote ROS desenvolvido para permitir comunicação entre o ROS e outros dispositivos que possuem uma porta serial ou uma interface de rede (FERGUSON, 2018). A escolha por desenvolver um novo pacote para executar a mesma função se fez necessário pela necessidade de alta taxa de transferência de dados entre o ROS e o SoC para que seja aceitável a utilização do SoC com a finalidade de acelerar o processamento por hardware.

Desenvolvendo um novo pacote de comunicação podemos extrair o máximo de desempenho da rede, como por xemplo, escolhendo o melhor protocolo (UDP ou TCP) e transferindo apenas os dados para o processamento da informação. Todos os códigos do cliente podem ser encontrados no repositório disponível em (NETO, 2021b).

6 Servidor

No modelo cliente-servidor, o servidor é o elemento responsável por executar uma ação somente após um pedido realizado pelo cliente, sendo assim, o servidor deve estar sempre preparado para responder à uma solicitação de serviço. Deste modo, além da capacidade de se comunicar com o cliente, o servidor deve oferecer algum serviço de interesse do cliente.

Neste trabalho o programa servidor disponibiliza ao cliente a interface com o FPGA presente no SoC, logo, ao receber o pedido do cliente, o servidor deve ser capaz de receber os dados vindos do cliente à unidade de processamento embarcada no FPGA e em seguida devolver esses dados já processados ao cliente. Entretanto, antes mesmo de iniciar o desenvolvimento do código do servidor, devemos compilar uma distribuição linux e configurar o processo de boot desse sistema no processador ARM presente no SoC. Estas etapas no desenvolvimento do servidor serão descritas neste capítulo.

6.1 Processo de BOOT

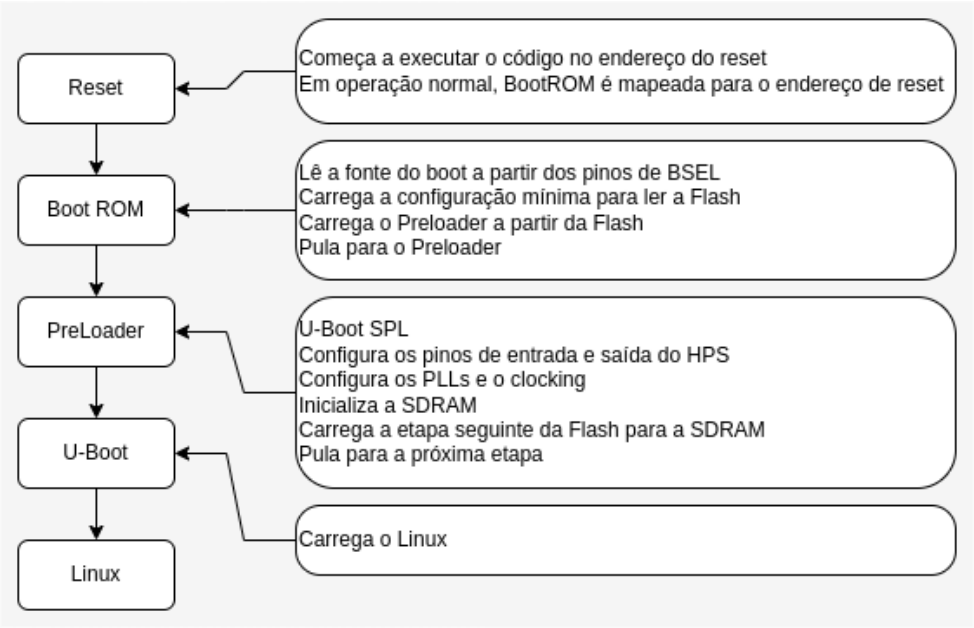
O fluxo de boot do linux na placa é resumido na Figura 13. O primeiro elemento de software é o *Boot ROM* que está gravado de fábrica internamente no dispositivo. Os arquivos de *PreLoader*, *U-boot* e do sistemas de aquivos do linux são salvos em um cartão de memória micro SD. O *Secondary Program Loader - SPL*, conhecido como *PreLoader*, é executado a partir da Boot ROM. Ele é responsável por configurar o sistema para que o *bootloader* (U-boot) possa ser executado. A Intel fornece uma ferramenta chamada BSP editor que, a partir de arquivos que descrevem o hardware, podem gerar o PreLoader para o projeto específico (ROCKETBOARDS.ORG, 2015).

A etapa seguinte ao *PreLoader* é o *bootloader*. Nessa fase do boot todas as questões de baixo nível do SoC, como por exemplo, os clocks, pinos e SDRAM, já foram inicializados e estão prontos. O objetivo do bootloader, é obter essas informações do sistema e fazer com que ele funcione até o ponto onde o linux possa ser iniciado. Outra função importante do U-boot em um SoC Intel é programar o FPGA (ROCKETBOARDS.ORG, 2015).

6.2 Distribuição Linux rsyocto

Com todos os arquivos necessários para o boot do sistema já gerados a partir das ferramentas de desenvolvimento da Intel, poderemos escolher nosso sistema operacional. O sistema escolhido foi uma distribuição linux desenvolvida exclusivamente para SoCs da

Figura 13 – Boot linux embarcado



Fonte:(ROCKETBOARDS.ORG, 2015)

Intel com fpga integrado, como pode ser observado na descrição da distribuição disponível em (ROBIN, 2022):

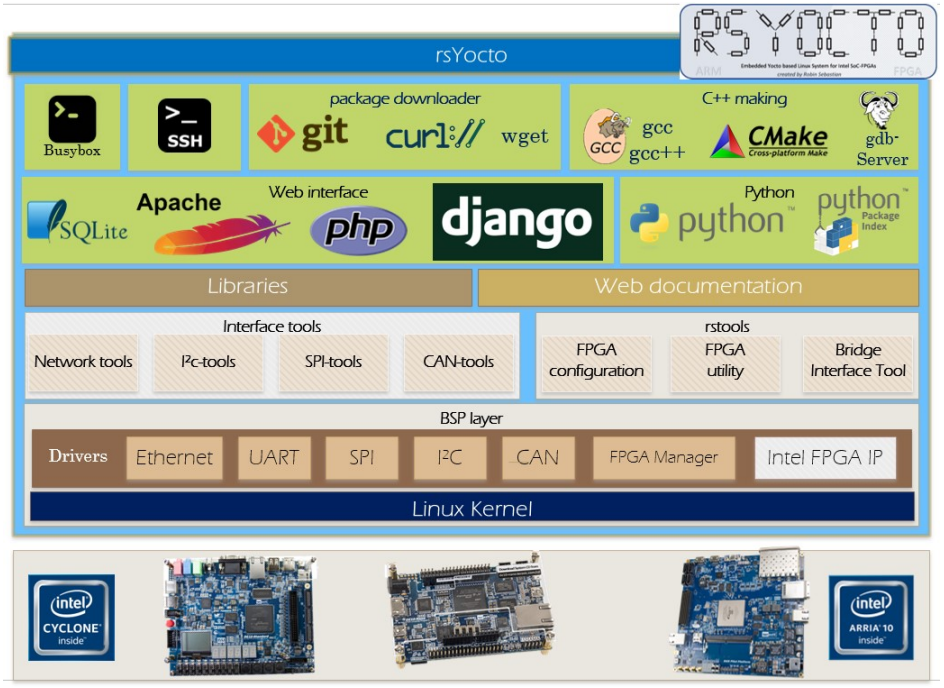
> "**rsyocto** is an open source Embedded Linux Distribution designed with the Yocto Project and with a custom build flow to be optimized for Intel SoC-FPGAs (Intel Cyclone V and Intel Arria 10 SX SoC-FPGA with an ARM Cortex-A9) to achieve the best customization for the strong requirements of modern embedded SoC-FPGA applications."

O rsyocto faz uso do Kernel Linux **linux-socfpga 5.11** (ALTERA-OPENSOURCE, 2022) e possui um conjunto de ferramentas necessárias para ajudar a simplificar o uso e desenvolvimento de aplicações desenvolvidas para os SoCs com FPGA integrado da Intel. Além destas ferramentas o rsyocto dispõe de drivers para todos os periféricos de comunicação integrados SoC, como por exemplo, a interfaces I2C e CAN, e para todas as interfaces entres o HPS e o FPGA. O rsyocto oferece um conjunto de simples aplicações que podem ser executadas em linha de comando, que possibilitam executar novas configurações no FPGA, usando o FPGA Manager, até mesmo ler e escrever a interface ARM AXI-Bridge que permite interagir com o FPGA.

Essas aplicações simplificam a comunicação com o FPGA a partir de simples comandos que podem ser executados a partir do terminal, através de todas as interfaces disponíveis, estes comandos são: **lwhps2fpga**, que permite ações de leitura e escrita no barramento Lightweight HPS-to-FPGA-Bridge, o **hps2fpga** para leitura e escrita no barramento HPS-to-FPGA-Bridge, o **hps2sdram**, que proporciona acesso à interface da

SDRAM e os **gpi** e **gpo** para acesso aos sinais de uso geral. Na Figura 14 podemos ter uma visão geral de todas as ferramentas disponíveis na distribuição linux *rsyocto*.

Figura 14 – Overview do rsyocto



Fonte:(ROBIN, 2022)

A seguir são listadas algumas características da distribuição linux rsyocto que fazem dela uma excelente escolha para ser utilizada neste trabalho:

- Embedded Linux specially developed for Intel SoC-FPGAs
- Linux Kernel 5.11 (Source)
- Full usage of the Dual-Core ARM (ARMv7-A) Cortex-A9 with
- FPGA Fabric configuration during the boot and with a single Linux command
- All Bridge Interfaces between the HPS and FPGA are enabled and ready for use!
- Tools to interact with the FPGA Fabric via the ARM AXI HPS-to-FPGA bridges
- HPS Hard IP components (I²C-,SPI-, CAN-BUS or UART) are routed to FPGA
- USB Host support with test tools (e.g. lsusb)
- Full Linux Network stack with dynamic and static iPv4 is supported
- OpenSSH-Server starts automatically during boot

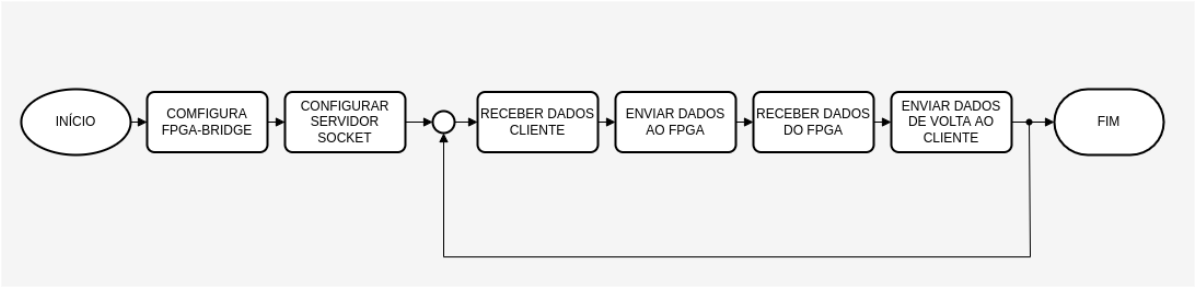
- gcccompiler 9.3.0; glibc and glib-2.0(The GNU C Library) cmake 3.16.5
- Python 3.8 Python3-dev and Python-dev
- git 2.31, wget 1.20.3, curl 7.69.1
- opkg package manager enables to add packages from different Linux Distributions.
- Changing the running FPGA-Configuration of the FPGA-Fabric

6.3 Interface socket server

Com o linux já devidamente configurado, ele já pode ser inicializado no processador do SoC, assim podemos iniciar o desenvolvimento do servidor. No primeiro momento devemos realizar o download do código fonte da biblioteca de comunicação, a libinterfacesocket, que já foi detalhada anteriormente, o código fonte pode ser baixado em (NETO, 2021c). Com os arquivos já no sistema de diretórios do SoC devemos realizar a compilação e a instalação da libinterfacesocket, só assim o servidor terá acesso a suas funcionalidades.

Logo que é inicializado o servidor mantém uma porta aberta para estabelecer uma conexão com o cliente e assim que a conexão é estabelecida, o cliente já pode começar a enviar os dados. Ao receber os dados do cliente, o servidor envia-os ao FPGA que os processa e os devolve ao servidor para que ele possa reenviar os dados já processados ao cliente. Um fluxograma simplificado desse processo pode ser visualizado na Figura 15.

Figura 15 – Fluxograma simplificado do servidor



Fonte: Elaborado pelo autor

O acesso do servidor ao FPGA é feito através de mapeamento de endereços do linux, os SoCs Intel possuem uma arquitetura de memória mapeada. Desta forma, além da biblioteca desenvolvida durante a pesquisa, um outro arquivo cabeçalho é necessário para facilitar o acesso aos endereços de memória do HPS. Quando uma instância do processador ARM é incluída no projeto do QSys Designer, é gerado um arquivo.sopcinfo no momento em que o projeto é compilado. Podemos usar esse arquivo como entrada da ferramenta *sopc-create-header-files* presente no SoC EDS para gerar um novo arquivo com extensão *.h*, que lista os endereços base de todos os módulos (IP) incluídos no FPGA.

Portanto, o arquivo cabeçalho gerado disponibiliza o *offset* do endereço em que cada periférico está localizado para cada uma das FPGA-bridges disponíveis. Deste modo o servidor pode enviar ou receber dados ao periférico conhecendo o endereço da FPGA-bridge em que este periférico está conectado e o seu nome, sem a necessidade de se conhecer exatamente o seu endereço. Na Figura 16 é apresentado uma parte do arquivo cabeçalho gerado pela ferramenta sopc-create-reader-files, neste trecho são realizadas algumas definições que facilitam o acesso ao IP dobro, que já está instanciado no FPGA. Este módulo IP foi usado para testar o processo completo de comunicação entre o tópico ROS até o FPGA.

Figura 16 – Definição de endereço presente no arquivo hps_0.h

```
/*
 * Macros for device 'Dobro_0', class 'Dobro'
 * The macros are prefixed with 'DOBRO_0_'.
 * The prefix is the slave descriptor.
 */
#define DOBRO_0_COMPONENT_TYPE Dobro
#define DOBRO_0_COMPONENT_NAME Dobro_0
#define DOBRO_0_BASE 0x38
#define DOBRO_0_SPAN 4
#define DOBRO_0_END 0x3b
```

Fonte: Elaborado pelo autor

Para interagir com FPGA usando o recurso de memória mapeada, a primeira coisa que devemos fazer é abrir o dispositivo de memória do sistema. Em distribuições linux o acesso à memória física do sistema é realizado através do dispositivo /dev/mem, esse dispositivo funciona como um arquivo caracteres que representa uma imagem da memória principal do computador (KERRISK, 2021). Por se comportar como um arquivo no sistema, podemos no programa abri-lo como abriríamos qualquer arquivo de caracteres, com a funções *open*, como pode ser visto na linha 2 da Figura 17.

Em seguida podemos usar o descritor de arquivo gerado como retorno da função open, para criar um ponteiro, que dará acesso à memória principal do sistema. Para gerarmos este ponteiro, podemos usar a função mmap, ela cria um mapeamento virtual no espaço de endereço passado como argumento da função. Nos dois primeiros argumentos da função mmap, como pode ser visto na linha 10 da Figura 17, podemos definir o espaço de endereços para o mapeamento. No código apresentado este espaço se inicial no endereço zero, NULL, e termina na constante HPS_TO_FPGA_AXI_SPAN (0x3C000000), que guarda o valor do alcance máximo do espaço de endereço das interfaces entre o HPS e o FPGA. O ultimo argumento da função mmap é o endereço base da Lightweight HPS-to-FPGA-Bridge (0xC0000000), representado pela constante HPS_TO_FPGA_AXI_BASE.

Figura 17 – Memória mapeada

```
// Open up the /dev/mem device
devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
if(devmem_fd < 0) {
        perror("devmem open");
        exit(EXIT_FAILURE);
}

// mmap() the entire address space of the Lightweight
// bridge so we can access our custom module
lw_bridge_map = (uint32_t *)mmap(
                                NULL,
                                HPS_TO_FPGA_AXI_SPAN,
                                PROT_READ|PROT_WRITE,
                                MAP_SHARED,
                                devmem_fd,
                                HPS_TO_FPGA_AXI_BASE );

if(lw_bridge_map == MAP_FAILED) {
        perror("devmem mmap");
        close(devmem_fd);
        exit(EXIT_FAILURE);
}
```

Fonte: Elaborado pelo autor

Agora que já existe um ponteiro para a região de memória onde se encontra a interface de comunicação com o FPGA, podemos enviar e receber dados para o IP instanciado no FPGA. Para isto, vamos usar este ponteiro para o endereço da HPS-to-FPGA-Bridge, somado com o offset presente no arquivo hps_0.h. Consequentemente, será possível interagir com o IP em um programa escrito em linguagem C, da mesma maneira que interagimos com um ponteiro comum. O bloco de código da Figura 18 ilustra este processo de escrita e leitura de dados em um IP através de um ponteiro

Figura 18 – Memória mapeada

```
// Set the dobro_map to the correct offset within the RAM
uint32_t * dobro_map = 0;
dobro_map = (uint32_t*)(lw_bridge_map + DOBRO_0_BASE);

*dobro_map = 56;
valor = *dobro_map;
```

Fonte: Elaborado pelo autor

Parte III

Resultados

7 Resultados Alcançados

Para validar a comunicação foram idealizados dois ensaios: o primeiro com o objetivo de testar o fluxo completo de troca de dados entre um nó ROS e uma aplicação embarcada no FPGA; o segundo ensaio teve o objetivo de testar um fluxo grande de dados trafegando entre o ROS e o SoC.

No primeiro ensaio foi descrito um circuito simples para calcular o dobro de um número, em seguida a *HPS-to-FPGA Lightweight* bridge foi utilizada para realizar a comunicação entre o servidor rodando no HPS e o FPGA. No nó cliente, rodando no ROS, foram configurados dois tópicos, um para o número a ser enviado e outro para receber o resultado calculado no FPGA, da mesma maneira que mostra o esquema da Figura

Com o objetivo de testar a sobrecarga de dados na rede, para o segundo ensaio foi realizado o envio de uma stream de vídeo com resolução de 1280x720 em 15 fps. Ao receber os dados o servidor os reenvia ao cliente que o republica em um tópico. Neste ensaio os dados não foram processados no FPGA.

Os resultados alcançados para os dois ensaios foram considerados satisfatórios. Apesar de, na transferência das imagens, ter sido observado um pequeno *delay* entre a imagem enviada e a recebida de volta.

8 Conclusão

O objetivo de estabelecer uma comunicação com alta taxa de transferência entre o framework de robótica ROS e um SoC com FPGA integrado foi alcançado, possibilitando o uso de aceleração por hardware de maneira mais ágil em projetos de robótica.

9 Estudos funturos

Por se tratar de um trabalho interdisciplinar que abrange áreas como programação de rede, compilação e configuração de linux embarcado, desenvolvimento com o ROS e com SoC, trabalhos futuros podem dar ênfase a alguma dessas áreas específicas, otimizando alguns pontos do projeto para se alcançar maiores taxas de transferência, tornar o sistema ainda mais genérico e fácil de ser usados por outros grupos de trabalho que tenham o interesse de adicionar processamento por hardware em seus projetos de robótica.

Referências

ALTERA-OPENSOURCE. *linux-socfpga*. 2022. Disponível em: <https://github.com/altera-opensource/linux-socfpga>. Citado na página 34.

FERGUSON, M. *Rosserial*. 2018. Rosserial. Disponível em: <http://wiki.ros.org/rosserial>. Acesso em: 20 julho 2021. Citado 2 vezes nas páginas 31 e 32.

JOSEPH, L. *Mastering ROS for Robotics Programming*. 1. ed. Birmingham: Packt Publishing Ltd, 2015. Citado 2 vezes nas páginas 19 e 20.

KERRISK, M. *Linux Programmer's Manual - mem, kmem, port - system memory, kernel memory and system ports*. [S.l.], 2021. Disponível em: <https://man7.org/linux/man-pages/man4/mem.4.html>. Acesso em: 20 julho 2022. Citado na página 37.

MAHTANI, A. et al. *Effective Robotics Programming with ROS*. 3. ed. Birmingham: Packt Publishing Ltd, 2016. Citado 2 vezes nas páginas 14 e 24.

MARTINEZ, A.; FERNáNDEZ, E. *Learning ROS for Robotics Programming*. 1. ed. [S.l.]: Packt Publishing, 2013. Citado na página 21.

MYER-BAESE, U. *Digital Signal Processing with Field Programmable Gate Arrays*. 4. ed. Nova York: Springer, 2014. Citado na página 8.

NETO, N. P. *Biblioteca projeto interfacesocket*. 2021. Disponível em: <https://github.com/NestorDP/libinterfacesocket>. Citado na página 28.

NETO, N. P. *interface_socketr*. 2021. Disponível em: <https://github.com/NestorDP/interface_socket>. Citado na página 32.

NETO, N. P. *interface_socket_server*. 2021. Disponível em: <https://github.com/NestorDP/interface_socket_server>. Citado na página 36.

PYO, Y. et al. *ROS Robot Programming*. Nova York: ROBOTIS, 2017. Citado 4 vezes nas páginas 9, 14, 16 e 24.

ROBIN, S. *rsyocto*. 2022. Disponível em: <https://github.com/robseb/rsyocto>. Citado 2 vezes nas páginas 34 e 35.

ROCKETBOARDS.ORG. *Embedded Linux Beginners Guide*. 2015. Disponível em: <https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide>. Acesso em: 5 outubro 2021. Citado 2 vezes nas páginas 33 e 34.

ROS. *ROS - Robot Operating System*. 2011. Open Source Robotics Fundation. Disponível em: <https://www.ros.org/>. Acesso em: 22 outubro 2021. Citado na página 14.

ROS. *Introduction*. 2018. Open Source Robotics Fundation. Disponível em: <http://wiki.ros.org/ROS/Introduction>. Acesso em: 20 julho 2021. Citado 2 vezes nas páginas 14 e 15.

ROS. *catkin/package.xml*. 2019. Disponível em: <http://wiki.ros.org/catkin/package.xml>. Acesso em: 21 setembro 2021. Citado na página 30.

ROS. *Pckages*. 2019. Open Source Robotics Fundation. Disponível em: <http://wiki.ros.org/Packages>. Acesso em: 20 julho 2021. Citado na página 18.

YAMASHINA, K. et al. Proposal of ros-compliant fpga component for low-power robotic systems: case study on image processing application. *2nd International Workshop on FPGAs for Software Programmers (FSP 2015)*, p. 62–67, 2015. Disponível em: <https://arxiv.org/pdf/1508.07123.pdf>. Citado na página 9.