Nestor Dias Pereira Neto

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

Salvador

Outubro 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 Luiz Alves de Oliveira

Coorientador: Paulo César Machado de Abreu Farias

Salvador

Outubro 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, Outubro 2022-
55p. : il. (algumas color.) ; 30 cm.

Orientador: Wagner Luiz Alves de Oliveira

Dissertação (Mestrado) – Universidade Federal da Bahia - UFBA
Escola Politécnica
Programa de Pós-Graduação em Engenharia Elétrica, Outubro 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, xx de Outubro de 2022:

Wagner Luiz Alves de Oliveira
Orientador

Paulo César Machado de Abreu Farias
Coorientador

Professor
Convidado 1

Professor
Convidado 2

Salvador
Outubro 2022

Resumo

Os novos projetos em robótica têm exigido cada vez mais poder de processamento, consequentemente, uma maior eficiência energética, principalmente nas aplicações que fazem uso de baterias. Dessa maneira o uso do FPGA pode contribuir com ganho de poder de processamento associado ao baixo consumo. Neste trabalho foi elaborado um método para estabelecer a comunicação entre o ROS e um FPGA embarcado em um SoC da família Cyclone V. Por meio de um sistema servidor-cliente, através de um link Gigabit ethernet, foi possível estabelecer a comunicação entre os elementos do sistema através do desenvolvimento de um software servidor rodando no HPS do SoC, além de um pacote ROS para interfacear os outros pacotes do sistema ROS e a porta ethernet. Os testes de desempenho foram realizados no kit de desenvolvimento DE10-Nano e alcançaram um resultado considerado aceitável.

Palavras-chave: ROS, SoC, FPGA, Cyclone V.

Abstract

oihbqptipbõq4tnpot4photnj4yojnj4ynojp

Keywords: latex. abntex. text editoration.

Sumário

1 Introdução

Nos últimos anos novas técnicas para construção de robôs t se destacado, em especial áreas como robótica móvel e robótica colaborativa. 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 aquelas desempenhadas por seres humanos.

Este grau de precisão requerido no desenvolvimento de novos robôs exige 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, já na fase de 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 projetos implementados em FPGA.

Por essas características, FPGAs necessitam 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. As maiores dificuldades no desenvolvimento de soluções baseadas em FPGAs referem-se ao longo tempo de projeto e à necessidade de mão-de-obra extremamente especializada, o que aumenta consideravelmente o custo final de tais projetos. Desta forma, o uso de FPGAs em projetos de robótica acaba sendo 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 software usados em robótica, fornecer driver para componentes de hardwares específicos (sensores e atuadores), gerenciar troca de mensagens entre os nós que fazem parte do sistema, são as funções 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, pode-se 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 a uma taxa muito rápida, desde o ano de seu lançamento, em 2007, até 2020, o ROS aumentou de 1 para 2647 pacotes (KOLAK et al., 2020).

Aproveitar as facilidades de desenvolvimento proporcionadas pelo ROS em conjunto com o alto poder de processamento e baixo consumo que FPGAs oferecem, 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, adicionalmente, tornar 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.

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

Este problema é o que este trabalho busca resolver, possibilitando assim, o uso de aceleração por hardware através do FPGA, para inclusão de tal dispositivo no desenvolvimento de novos projetos de robótica. Projetistas especializados em robótica poderão aproveitar dos benefícios do uso do hardware dedicado em seus projetos, enquanto 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 conceitos 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 o desempenho 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: 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. No Capítulo 5 é explicado o Cliente, enquanto o Servidor é descrito no Capítulo 6

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

Parte I

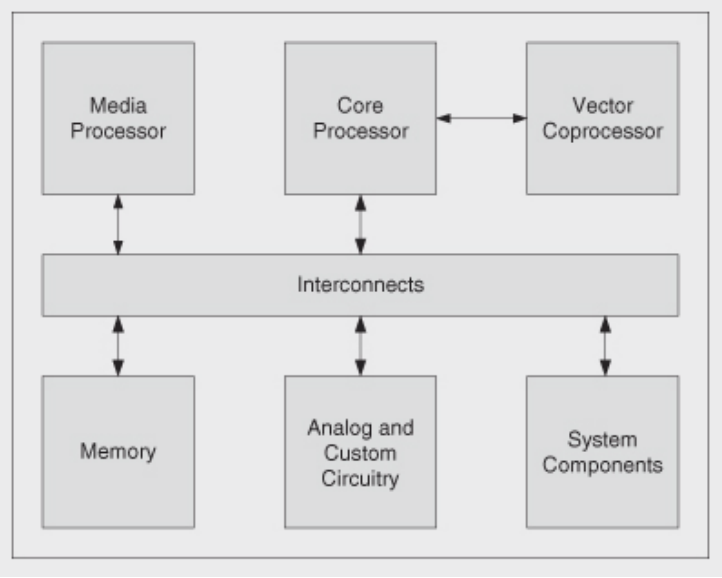
Referenciais Teóricos

2 Cyclone V SoC-FPGA

SoC é um acrônimo de *System-on-a-Chip* ou apenas *System on Chip*, o qual é um sistema computacional inteiro contido em apenas um circuito integrado. Sendo assim, um SoC pode combinar diferentes elementos, em diferentes configurações, para formar um sistema completo. Portanto um SoC não se restringe à apenas um processador: além da CPU, ele integra memória, periféricos de controle (por exemplo, barramento de comunicação USB), unidade de processamento gráfico e outros periféricos específicos para uma determinada aplicação.

Para Flynn e Luk (2011), um system-on-chip é uma arquitetura que estabelece uma montagem de processadores, memórias, e interconexões feitos sobre medida para uma determinada aplicação. A figura 1 ilustra um SoC com alguns elementos básicos, que pode incluir múltiplos processadores de diferentes arquiteturas, interconectados com uma ou mais memórias, o que pode oferecer meios de se criar sistemas reconfiguráveis, adicionando maior flexibilidade a um SoC. Em muitas ocasiões um SoC pode ser equipado com componentes analógicos, como conversores analógico-digitais.

Figura 1 – Esquema SoC básico

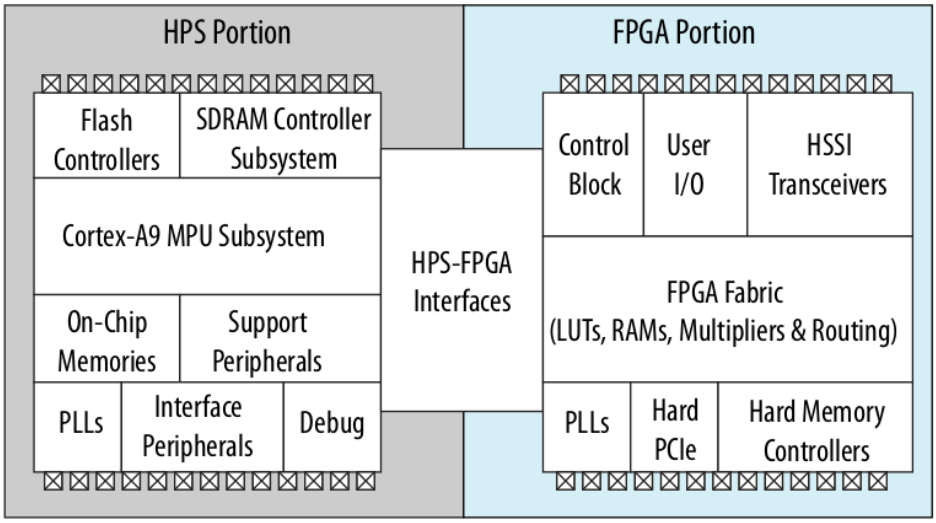


Fonte: Flynn e Luk (2011)

Dentre os dispositivos classificados como SoC, a Intel fornece uma linha de produtos classificados como SoC-FPGA, os quais se caracterizam por possuir um rede de FPGA integrados a um Processador ARM. Uma família de produtos que possuem essa característica é a Cyclone V SoC-FPGA. Estes componentes são constituídos por um *Hardware Processor System* (HPS) que possui processadores ARM Cortex-A9 de um ou

dois núcleos e um FPGA. A Figura 2 oferece uma visão global da integração entre os dois componetes principais do SoC-FPGA da família Cyclone V.

Figura 2 – Diagrama de blocos simplificado Cyclone V



Fonte: Altera (2018)

Lançada iniciante pela Altera que foi adquirida em 2015 pela Intel (MAAN; BAKER, 2015), a família Cyclone V SoC-FPGA alia o poder de processamento do processador Cortex-A9 à versatilidade e flexibilidade do FPGA, o que faz desta linha de componentes possuir um alto poder de processamento aliado a um baixo consumo. Os caminhos de interconexão entre o HPS e o FPGA do Cyclone V proporciona altas taxas de transferência, o que seria inviável em sistemas com dois chips. Esta estratégia de integração em um único circuito integrado oferece (INTEL, 2022):

- Largura de banda de pico de mais de 100 Gbps;
- Coerência de dados integrada;
- Significativa economia de energia do sistema, eliminando caminhos de E/S externos entre o processador e o FPGA.

Este nível de integração entre os dois componentes que formam o SoC FPGA provê uma solução com baixa potência dissipada, aliado a um reduzido custo e espaço da placa necessária para a montagem do sistema. Ou seja, o uso dos SoC-FPGA, com processador ARM combinado com FPGA, interligados através de um *backbone* de alta largura de banda e baixa potência, disponíveis nos produtos da linha da família Cyclone V da Intel, possibilita o desenvolvimento de aplicações com excelente desempenho, alta flexibilidade, além de baixo custo e baixo consumo de energia.

2.1 ARM Cortex-A9

O processador Cortex A9 é uma linha de processadores ARM otimizados para alcançarem maior desempenho aliado a um baixo consumo. Esta linha de processadores ARM é uma das mais utilizadas, nas mais diversas aplicações. O Cortex A9 possui uma estrutura interna configurável, o que oferece flexibilidade ideal para o desenvolvimento de um novo SoC. Na Tabela 1 são listadas algumas das configurações básicas do processador Cortex A9.

Tabela 1 – Confi básicas do ARM Cortex A9

Arquitetura	Armv7-A
Multicore	1-4 cores
Suporte ISA	Armv7-A
	Thumb-2 ou Thumb
	Tecnologia de segurança TrustZone
	Jazelle DBX e Tecnologia RCT
	Extensão DSP
	Neon (Opcional)
	Ponto Flutuante (Opcional)
Unidade de Gerenciamento de Memória (MMU)	Armv7 MMU
Depuração	CoreSight
Caracteristicas	Dual-issue, partially out-of-order pipeline
	Arquitetura do sistema flexível com caches configuráveis
	Sistema de coerência com ACP port
	Desempenho 50% maior do que o processador Cortex-A8 configurado como *single-core*

Fonte: ARM (2022b)

2.1.1 Cortex-A9 MPCore Processor

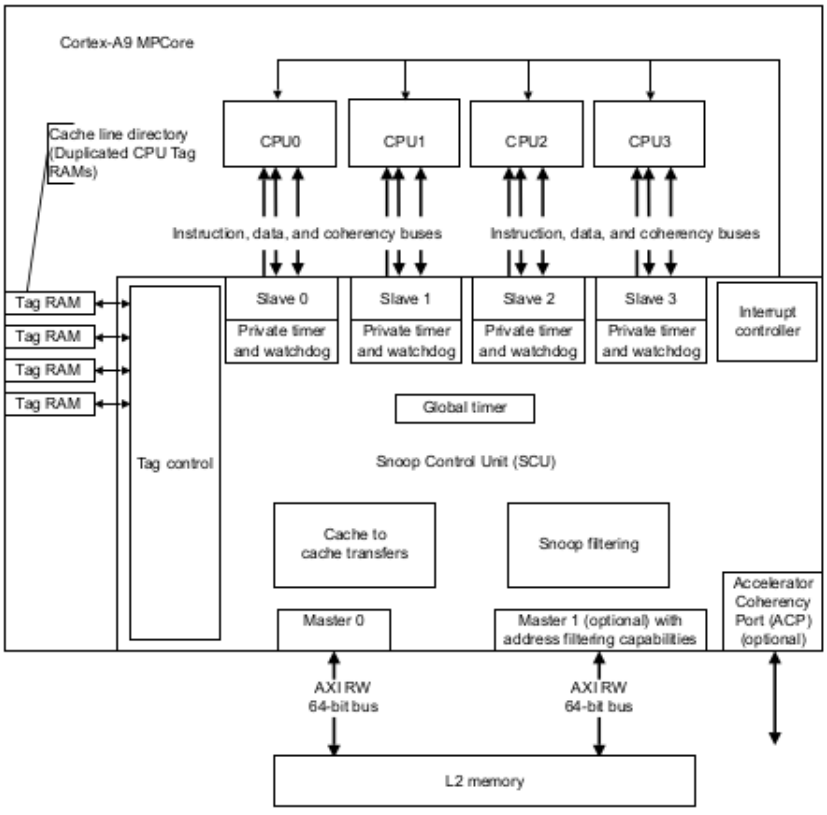
O Cortex-A9 MPCore processor é formado por três partes. Estas partes são componentes configuráveis que proporcionam a flexibilidade necessária para implementação de sistemas dedicados sob medida para determinada aplicação. Os componentes do Cortex-A9 MPCore são (ARM, 2016):

- De um a quatro processadores Cortex-A9, que podem ser agrupados em um cluster, e um *Snoop Control Unit* (SCU) que pode ser usado para assegurar a coerência do cluster;

- Um conjunto de periféricos de memória mapeada, incluindo timer global, watchdog e timer privativo para cada processador contido no cluster; e
- Um controlador de interrupção integrado que é uma implementação do Generic Interrupt Controller.

É possível implementar apenas um processador no cluster do Cortex-A9 MPCore, para isso, o processador deve ser implementado com suas próprias configurações de hardware. Mesmo contendo apenas um processador um SCU ainda estaré disponível, bem como ACP como um opcional. A Figura 3 apresenta um diagrama que representa o Cortex-A9 MPCore processor.

Figura 3 – Cortex-A9 MPCore Processor



Fonte: Arm (2016)

2.2 Field-Programmable Gate Array - FPGA

Field-Programmable Gate Arrays, ou simplesmente FPGA, são dispositivos semicondutores construídos através de uma matriz de blocos lógicos configuráveis, os CLBs (XILINX, 2022). Estes blocos lógicos são interconectados a partir de conexões programáveis,

o que permite ao projetista conectar esses blocos em configurações capazes de executar qualquer tarefa, desde simples portas lógicas até complexas funções.

Os FPGAs foram batizados como dispositivos programáveis em campo devido a capacidade de terem seu hardware reconfigurado pelo usuário, para executar uma função desejada, mesmo após seu processo de fabricação. Esta característica permite que atualizações e soluções de possíveis erros de projetos sejam executadas em campo. São estas características que diferem os FPGAs dos circuitos integrados de aplicação específicas, também conhecidos como ASICs - *Application Specific Integrated Circuits*, que como o nome já sugere, são circuitos desenvolvidos e fabricados para desempenharem apenas uma tarefa específica.

Extremamente versáteis, os FPGAs permitem que desenvolvedores testem inúmeras aplicações mesmo depois que o próprio FPGA e todo o hardware auxiliar já estarem montados em suas placas. Se por algum motivo uma nova configuração for necessária, novos arquivos de configuração podem ser transferidos para o FPGA, fornecendo ao dispositivo novas funcionalidades ou mesmo correção de defeitos. FPGAs são poderosas ferramentas de prototipagem devido à sua flexibilidade, mesmo depois do hardware já estar montado. Aplicações comerciais normalmente usam FPGAs em produtos finais quando a necessidade de computação paralela e requerimentos dinâmicos são um requisito do sistema (ARM, 2022a).

2.3 Hardware Processos System

Como foi mencionado no início deste capítulo, a família de dispositivos Cyclone V system-on-a-chip (SoC) é formada por duas partições, uma malha de FPGA e um processador Arm Cortex-A9, que pode estar organizado em dois ou apenas um núcleo. Esta última partição é chamada de *Hard Processor System* (HPS).

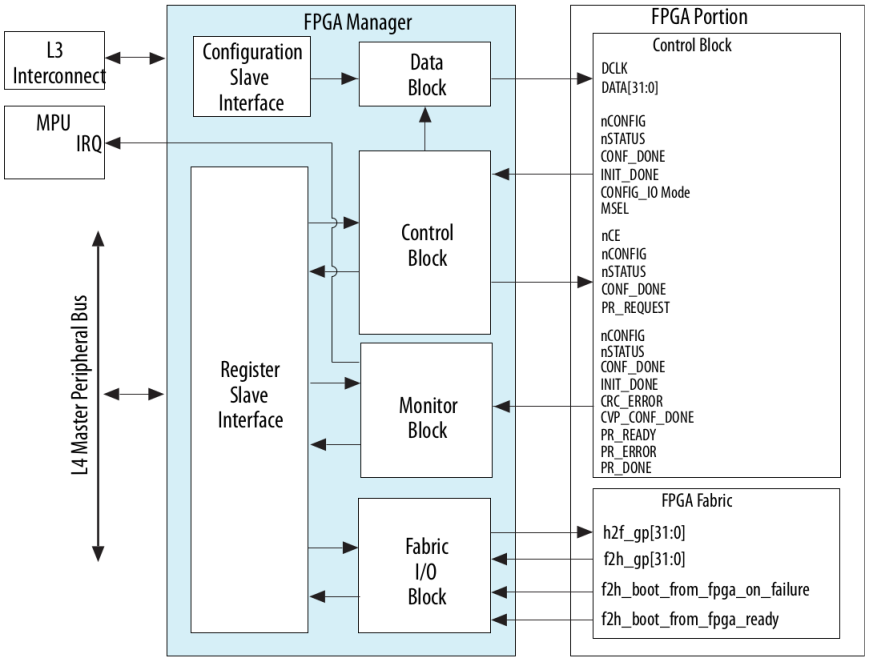
Os principais módulos do HPS são:

- Microprocessador Unit - MPU, subsistema com um ou dois ARM Cortex-A9 (MPCore Processor)
- Controladores de memória Flash;
- Controladores de SDRAM;
- Sistema de interconexão;
- On-chip memória; e
- Phase-locked loops (PLLs).

2.3.1 FPGA Manager

O *FPGA Manager* é o bloco do *Hard Processor System* responsável por gerenciar e monitorar a malha de FPGA presente no SoC. Através do *FPGA Manager* o projetista é capaz de configurar o FPGA, monitora seu estado, além de enviar dados entre o HPS e o FPGA.

Figura 4 – Diagrama de blocos FPGA Manager



Fonte: Altera (2018)

O *FPGA manager* possui as seguintes funcionalidades:

- Capacidade de realizar configuração parcial ou completa de toda a malha do FPGA;
- Enviar 32 sinais de uso geral do HPS para o FPGA;
- Receber 32 sinais de uso geral do FPGA para o HPS;
- Monitorar o status de configuração e potência do FPGA;
- Gerar interrupções para o HPS a partir das mudanças de status do FPGA; e
- Capacidade de resetar o FPGA.

Na Figura 4 podemos ver os principais blocos que formam o sistema do *FPGA Manager*, O *Registrer Slave Interface* se conecta ao L4 *Master Peripheral Bus* fornecendo acesso ao registro de *status* da malha do FPGA. Já o *Configuration slave interface* se

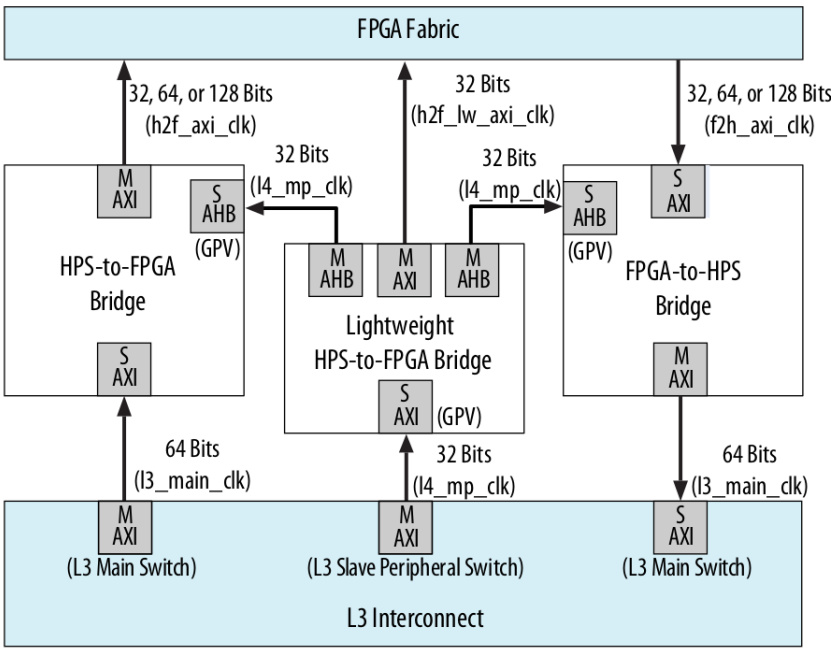
conecta ao *L3 interconnect* da MPU fornecendo um caminho para que o FPGA possa ser reconfigurado a partir do HPS.

Do lado da malha do FPGA, o *FPGA Manager* possui dois blocos, o *fabric I/O* e o *monitor*. No bloco *fabric I/O* os registradores *General-purpose input register* (gpi), *General-purpose output register* (gpo) e *Boot handshaking input register* (misci) possibilitam a comunicação entre o HPS e a malha do FPGA. Estes registradores só são acessíveis quando o FPGA está no modo usuário, modo ao qual o FPGA pode ser reconfigurado a partir do HPS. O bloco *monitor*, como pode ser deduzido pelo seu nome, fornece uma interface para o monitoramento da malha do FPGA, este bloco monitora sinais relacionados com a configuração do FPGA

2.3.2 HPS-FPGA Bridge

O HPS pode se comunicar com a malha do FPGA a partir de barramentos de alto desempenho, que podem alcançar larguras de banda superiores a 100Gbps. Estes barramentos, chamados de HPS-FPGA Bridge, usam o protocolo *Advanced Microcontroller Bus Architecture* (AMBA) *Advanced eXtensible Interface* (AXI), o qual foi desenvolvido pela própria ARM, constituindo-se num padrão de conexão dos blocos internos dos dispositivos ARM. A Figura 5 exibe o diagrama de blocos das conexões de todas a pontes presentes do Cyclone V.

Figura 5 – Diagrama de blocos HPS-FPGA Bridge



Fonte: Altera (2018)

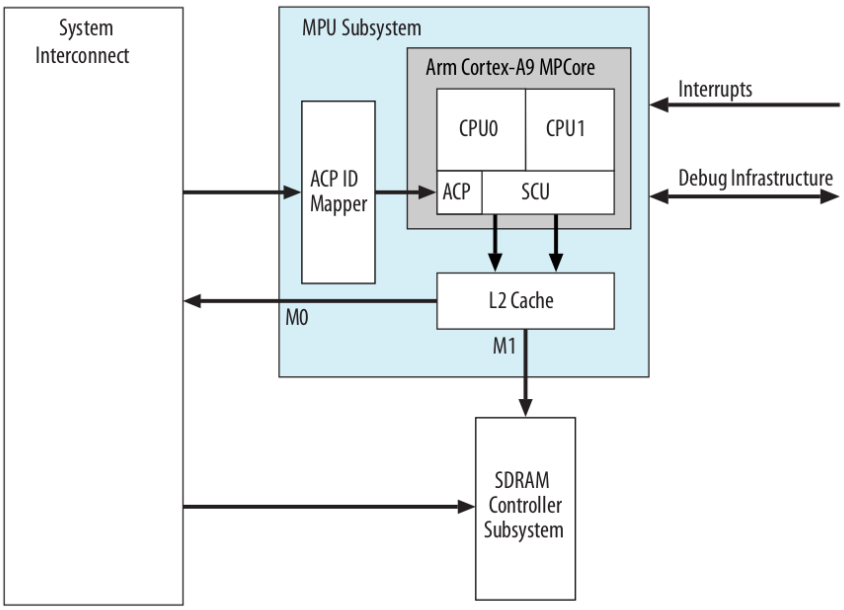
O HPS possui três pontes, são elas: *FPGA-to-HPS Bridge*, *HPS-to-FPGA Bridge*

e a *Lightweight HPS-to-FPGA Bridge*. Estas pontes permitem a comunicação entre o HPS e o FPGA e vice-versa, ou seja, quando o projetista incorpora periféricos ao seu projeto no FPGA, o processador pode acessar esses periféricos através de um barramento de alta velocidade. Como foi dito, essa comunicação também funciona no sentido inverso, do FPGA para o HPS, significando que um componente implementado no FPGA pode acessar regiões de memória e/ou periféricos do HPS.

2.3.3 Cortex-A9 Microprocessor Unit Subsystem

Os SoCs da família Cyclone V incluem um *Hard Processor System* formado por um Arm Cortex-A9 MPCore, com processador de uso geral de 32 bits com um ou dois núcleos, um L2 cache, um *Accelerator Coherency Port* (ACP) e módulo de depuração. Tanto o processador como outros módulos no HPS podem acessar blocos lógicos instaciados no FPGA através do *HPS-to-FPGA bridge*. Na Figura 6 é apresentado um diagrama de blocos simplificado do MPU.

Figura 6 – Cortex-A9 Microprocessor Unit Subsystem com bloco do sistema de interconexão



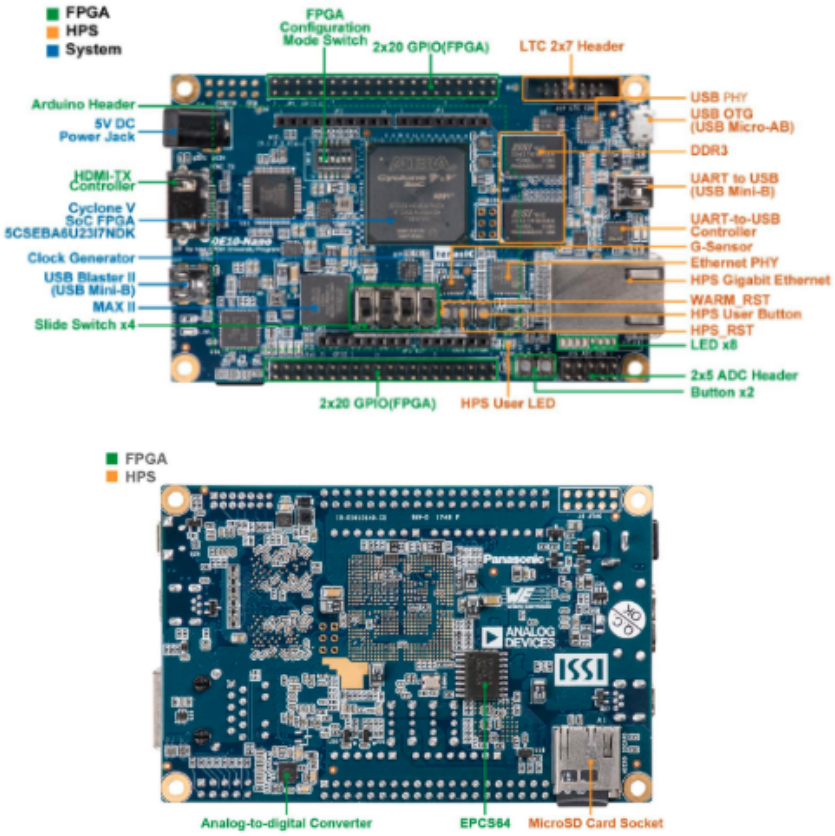
Fonte: Altera (2018)

2.4 Kit de desenvolvimento DE10-nano

Para o desenvolvimento deste trabalho foi escolhido o kit de desenvolvimento DE10-Nano da Terasic. Construído com base no SoC Cyclone® V SE 5CSEBA6U23I7

integrado com um processador ARM Cortex-A9 dual-core. O kit disponibiliza toda a estrutura básica de hardware necessário para que usuários possam aproveitar todo o poder do hardware reconfigurável oferecido pelo FPGA, combinado com um processador de alto desempenho. Além do SoC Intel, a placa DE10-Nano (Figura 7) oferece excelentes recursos, como memória DDR3 de alta velocidade, conversor analógico-digital, interface USB e interface de rede gigabit ethernet, os quais dão ao kit grande flexibilidade no desenvolvimento de novas aplicações.

Figura 7 – Kit de desenvolvimento DE10-Nano



Fonte:(TERASIC, 2020)

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 esta posição, 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 a sucederam. 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 ROS1 tem seu fim marcada para maio de 2025.

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 e 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. Dispositivos de 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 software 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. Neste contexto, 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 e compiladores, dentre outros.

Para complementar esses recursos básicos do sistema operacional, o ROS fornece funcionalidades específicas para o uso na robótica, tal como bibliotecas para transmissão e recepção de dados para uma variedade de dispositivos de hardwares comumente utilizados em sistemas robóticos. Esse tipo de software é conhecido como *middleware* ou *framework*. Como pode ser visto na Figura 8, 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 8 – 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 software 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 problema algum 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 a 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 e atuadores como os servos Dynamixel, podem ser integrados ao ROS sem impedimento algum.

- **Múltiplas linguagens:** O framework ROS pode ser programado em algumas linguagens de programação modernas. Podemos escrever código eficiente 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 processamento, que podem estar localizadas em diferentes computadores. Desde sensores microprocessados até mesmo controles específicos para motores, tais dispositivos 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 na resolução de 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 tal que todas as máquinas que compartilham estejam 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 e mapeamento, dentre outros, são usados em diferentes projetos, e o ROS permite que estes algoritmos sejam reaproveitados de duas maneiras possíveis:

- **Pacotes padrão:** São pacotes de software 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, o que possibilitando uma redução significativa do esforço necessário para o desenvolvimento de uma lógica para integrar estes pacotes ao seu sistema. Assin, 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.

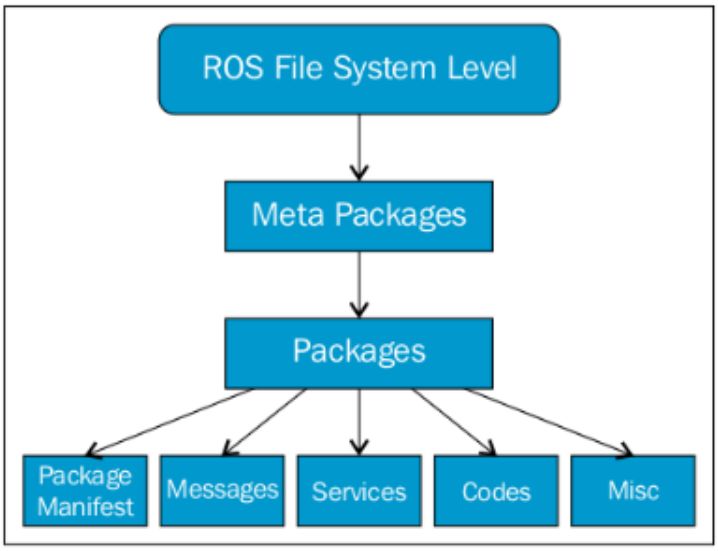
3.3 Sistema de aquivos

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 9 apresenta um diagrama de blocos do sistema de arquivo do ROS.

A seguir são apresentadas as definições 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 9 – Sistema de arquivos ROS



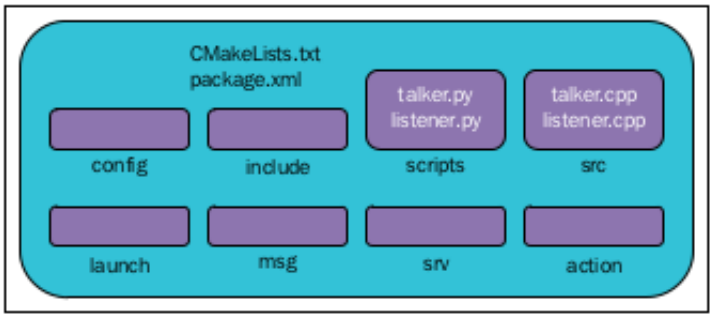
Fonte: Joseph (2015)

- **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.
- **Manifesto do *metapackage*:** 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
- **Arquivo .msg:** É o arquivo de descrição de um tipo de mensagem, é armazenado no diretório meu_pacote/msg/Mensagem.msg e define toda a estrutura de dados enviados a partir deste tipo de mensagem.
- **Arquivo .srv:** É o arquivo de descrição de um tipo de mensagem, é armazenado no diretório meu_pacote/srv/Servico.msg e define toda a estrutura de dados enviados a partir deste tipo de serviço.
- **Repositórios:** Pacotes ROS são compartilhados usando algum tipo de sistema de controle de versão, 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 10 podemos ter uma visão da estrutura típica de um pacote ROS

Figura 10 – Estrutura típica de um pacote ROS



Fonte: Joseph (2015)

Nos pacotes estão contidos um ou mais nós, como são chamadas as unidades de processamento no ROS, ou ainda 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, o qual 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, o qual 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, os pacotes ROS devem conter funcionalidades suficientes para serem úteis, mas não muito para torná-los muito grande e confusos, dificultando seu uso 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 mensagens ROS. 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 localizado no subdiretório msg/ dentro de um pacote ROS.

Existem dois segmentos em um arquivo .msg, são eles: campos e constantes. Campos são os dados que serão enviados dentro da mensagens. Constantes 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 2 lista os tipos de dados padrão do ROS que podem ser usados na criação de novas mensagens:

Tabela 2 – 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

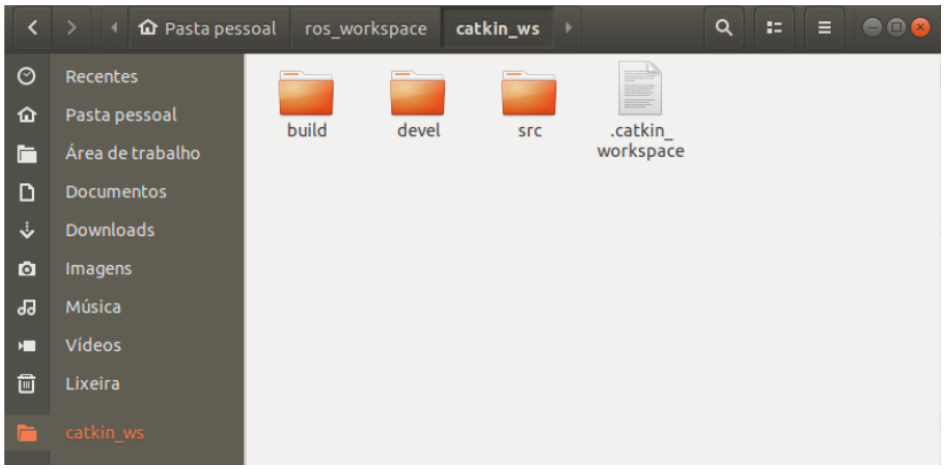
Fonte: Martinez e Fernández (2013)

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* é bastante 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 diretório 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 11.

Figura 11 – 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 dos fontes:** No espaço dos 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 intermediários para que os pacotes sejam compilados corretamente.
- **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 Componentes ROS

3.4.1 Nós ROS

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 torne bem mais simples.

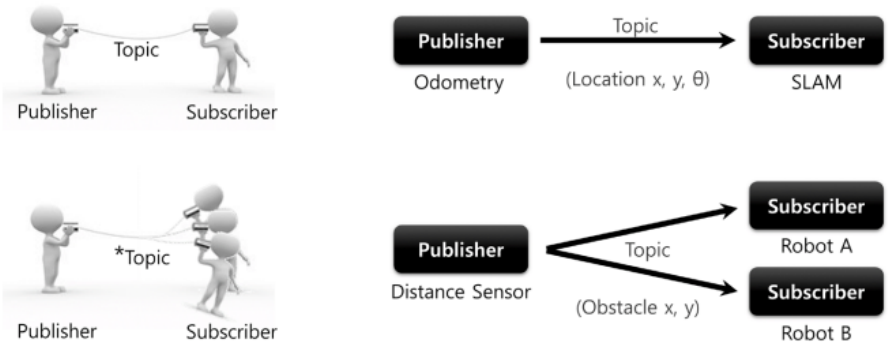
Ao ser executado, o nó possui um nome único no sistema, o que possibilita a sua comunicação com os demais nós. O ROS permite que os nós sejam escritos em diferentes linguagens de programação-as bibliotecas que fornecem a interface do ROS com uma linguagem específica são chamadas de clientes. As mais 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.

3.4.2 Tópicos ROS

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, fazendo 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, o que pode causar conflitos nas informações enviadas. A Figura 12 ilustra o processo de comunicação com tópico.

Figura 12 – Comunicação a partir de tópicos ROS



Fonte: Pyo et al. (2017)

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.

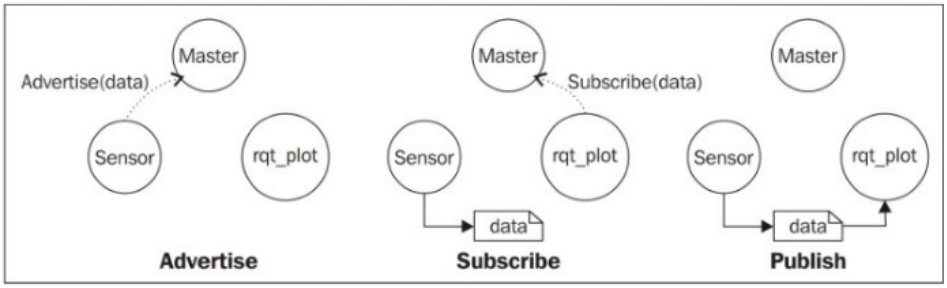
3.4.3 ROS *master*

O ROS *master* é o responsável por registrar nomes dos elementos que fazem parte do sistema, entre eles estão os nós, tópicos, serviços, action, tipos de mensagens, ele também é o encarregado por fornecer o servidor de parâmetros. Por gerenciar as informações das conexões entre as trocas de mensagens entre os nós, o *master* é o primeiro elemento que deve ser executado no ROS. Afunção do *master* é permitir que um determinado nó encontre outro, uma vez localizados os nós poderão se comunicar através de uma conexão per-to-per.

No momento em que um nós é executado no sistema, ele registra seu nome no *master*. Sendo assim, o ROS *master* possui os detalhes de todos os nós rodando no sistema. No momento em que qualquer detalhes de um nós muda, ele gera um callback para atualizar as novas informações no *master*. Quando um nó inicia a publicação de um tópico ele informará todos os detalhes ao *master*, desde do nome até o tipo de dados que serão enviado, em seguida, o *master* verifica se existe algum outro nó lendo o mesmo tópico, se algum nó estão querendo fazer a leitura deste tópico o ROS *master* irá compartilhar detalhes do nó que está publicando com o nós que quer ler o tópico.

Na Figura /reffig:rosMastering podemos ver como se da a interação entre o *master* e os nós que farão a comunicação.

Figura 13 – Gerenciamento de comunicação através do ROS master



Fonte: Mahtani et al. (2016)

3.4.4 ROS Parameters

Parâmetros são variáveis globais que podem ser usadas por nós. Os parâmetros são criados com valores padrões, que podem ser lidos ou escritos por algum processo dentro do sistema ROS. O principal objetivo dos parâmetros é fornecer ao sistema a capacidade de se adaptar a cenários distintos de maneira ágil. Por exemplo, o desenvolvedor pode criar um nó para leitura de uma câmera USB, dando à esse nós parâmetros para que usuários desse nós possam configurar o frame rate em que os dados serão publicados, o nome da porta USB em que a câmera está conectada, entre outros, sempre ficando a critério do desenvolvedor. Em casos especiais os parâmetros poderão ser atualizados em tempo de execução, uma função muito útil principalmente em sistemas dinâmicos, que opera em constante mudança.

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, tanto da 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 de possuir conhecimento em programação de redes.

4.1 Modelo cliente-servidor

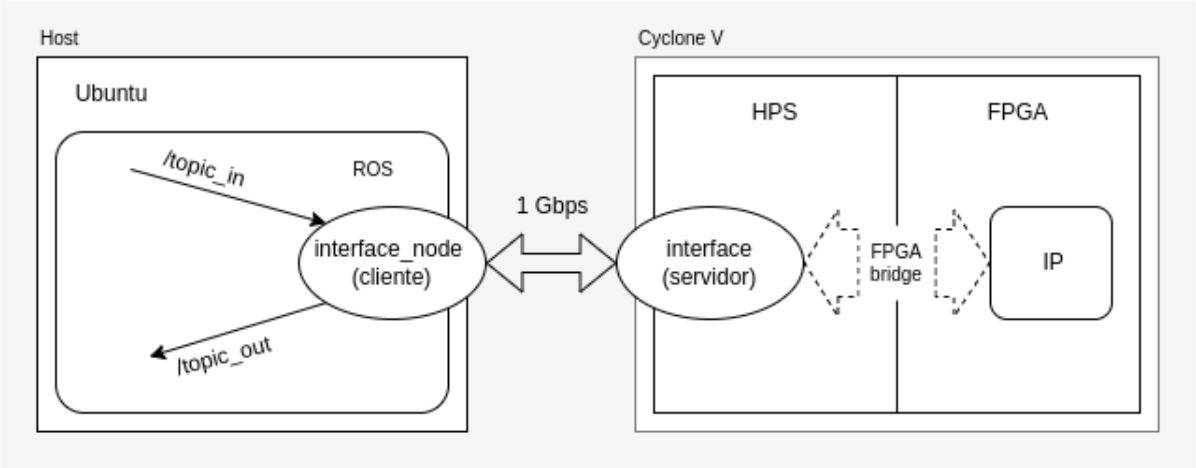
A comunicação entre o *host*, rodando o ROS, e a placa DE10-nano será estabelecida através de uma rede gigabit ethernet ponto a ponto, ou seja, o host e o SoC estarão conectados diretamente entre si. Desta maneira é possível obter o melhor desempenho da rede, alcançando as maiores taxas de transmissão 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, o que facilita, entre outras coisas, a depuração e manutenção do código, proporcionando mais agilidade e simplicidade no processo de desenvolvimento da solução.

Na Figura 14 podemos ter uma visão global do sistema, com destaque a 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 14 – 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 à parte e compilada como uma biblioteca estática. sDesta forma, uma vez testados e validados seus métodos de comunicação, tanto o código do cliente quanto o do servidor poderão fazer uso desta biblioteca, eliminando assim a necessidade de reescrever uma parte do código. Outra vantagem nessa abordagem é que, ao manter o código desassociado tanto do servidor

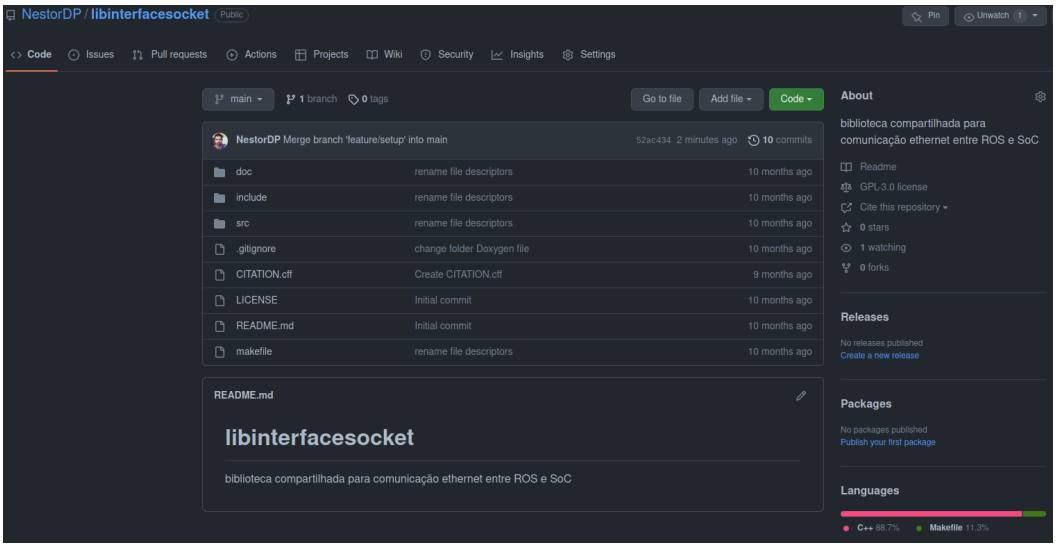
como do cliente, nos possibilita fazer alterações ou correções de erros, sem necessariamente realizar alterações nos códigos do servidor ou do cliente.

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 um 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, poderão ser testadas novas técnicas 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 15, 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 15 – Repositório libinterfacesocket



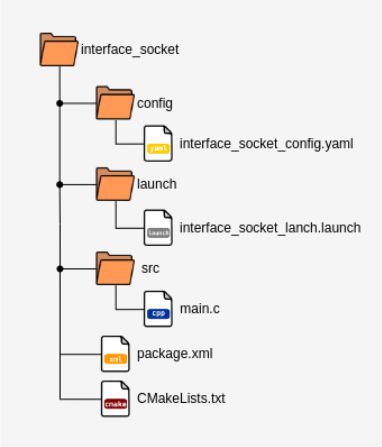
Fonte: do autor

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 16.

Figura 16 – 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, como pode ser visto na Figura 17. Essa mudança pode ocorrer sem a necessidade de recompilar o código fonte, o que 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 17 – 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.

Figura 18 – Função Callback para o tópico lido pelo cliente

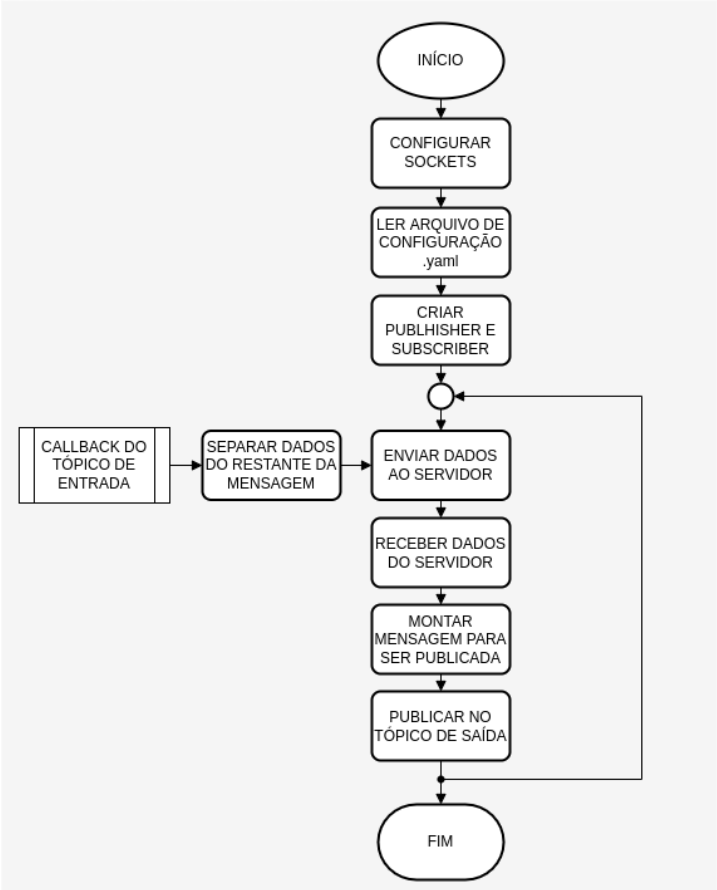
```
// Variavel que armazena apenas os dados contidos no
// topico lido pleo cliente
std::vector<uint8_t> dados_out(MSG_LEN);

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 atender à requisição. Neste ponto os dados já foram processados pelos circuitos do FPGA, mas como apenas os dados brutos foram enviados ao servidor, o cliente precisa montar novamente a mensagem, disponibilizando 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 descrito anteriormente é relativamente simples, como pode ser visto no fluxograma simplificado da Figura 19

Figura 19 – 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 técnicas para realizar a conexão entre o FPGA e o ROS, que se diferem da proposta por essa pesquisa. 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. Assim, 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

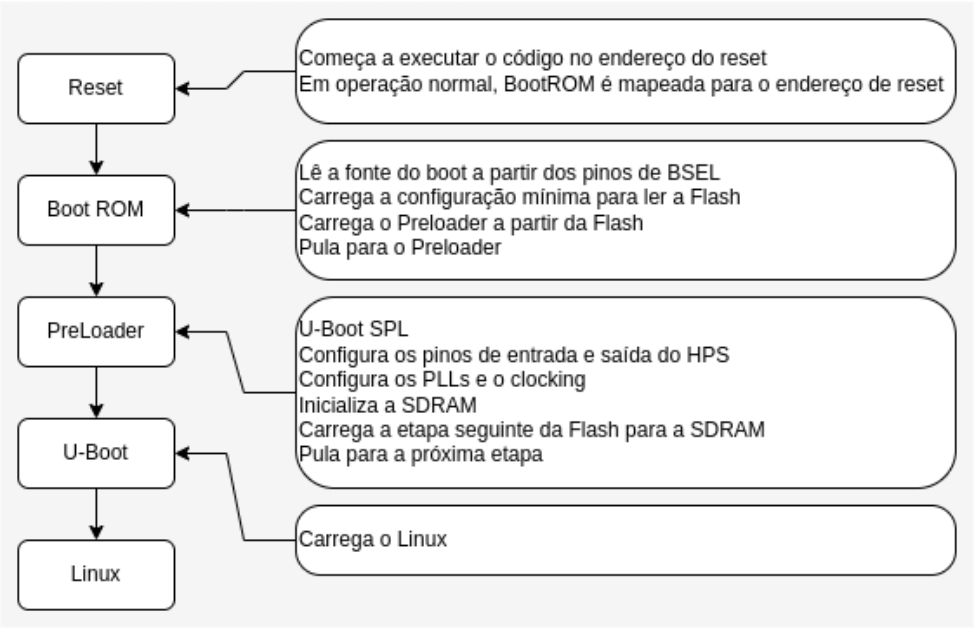
O fluxo de boot do Linux na placa é resumido na Figura 20. O primeiro elemento de software é o *Boot ROM* que está gravado de fábrica internamente no dispositivo. Os arquivos de *PreLoader*, *U-boot* e do sistemas de aquivo do linux são salvos em um cartão de memória micro SD. O *Secondary Program Loader - 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, pode 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

Figura 20 – Boot linux embarcado



Fonte:(ROCKETBOARDS.ORG, 2015)

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

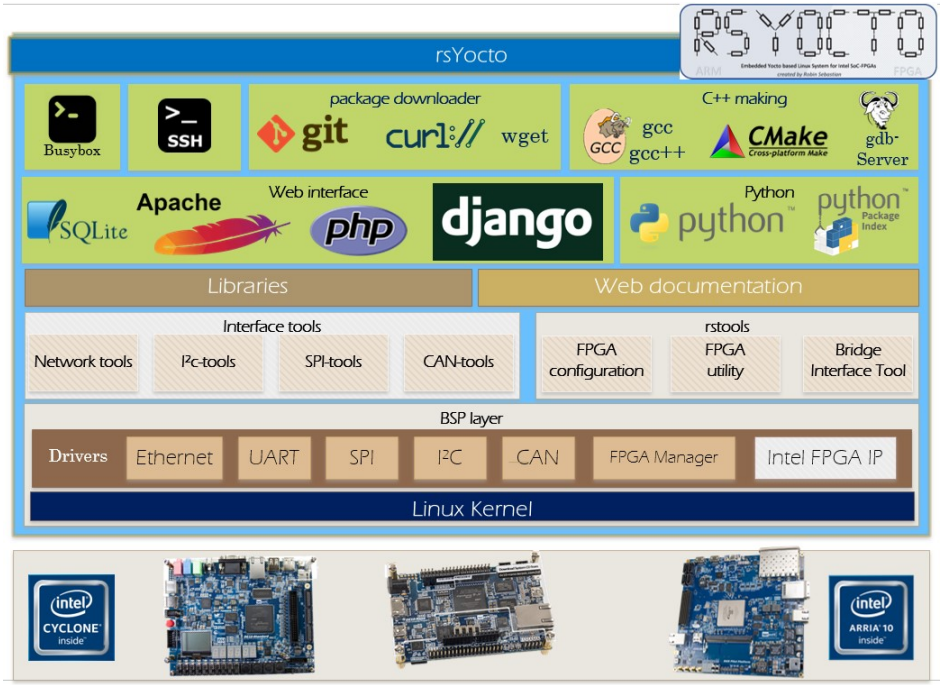
> "**rsyocto** is an open source Embedded Linux Distribution designed with the Yocto Project and with a custom build flow to be optimized for Intel SoC-FPGAs (Intel Cyclone V and Intel Arria 10 SX SoC-FPGA with an ARM Cortex-A9) to achieve the best customization for the strong requirements of modern 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 entre 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 21 podemos ter uma visão geral de todas as ferramentas disponíveis na distribuição linux *rsyocto*.

Figura 21 – Visão geral da distribuição Linux *rsyocto*



Fonte:(ROBIN, 2022)

A seguir são listadas algumas características da distribuição linux rsyocto que fizeram dela uma excelente escolha para uso neste trabalho:

- Linux embarcado especialmente desenvolvido para SoC-FPGAda Intel;
- Linux Kernel 5.11;
- Acesso total ao Dual-Core ARM (ARMv7-A) Cortex-A9;
- Configuração da malha FPGA durante o *boot* e com comandos Linux simples;
- Todas as interfaces entre o HPS e o FPGA estão habilitadas e prontas para uso;
- Ferramentas para interagir com a malha do FPGA a partir das *ARM AXI HPS-toFPGA bridges*;
- Componentes Hard IP (I²C-,SPI-, CAN-BUS or UART) do HPS estão roteados para o FPGA;
- Suporte para o USB *Host* com ferramentas de teste (lsusb);
- Interface de rede com suporte ao iPv4 dinâmico e estático;

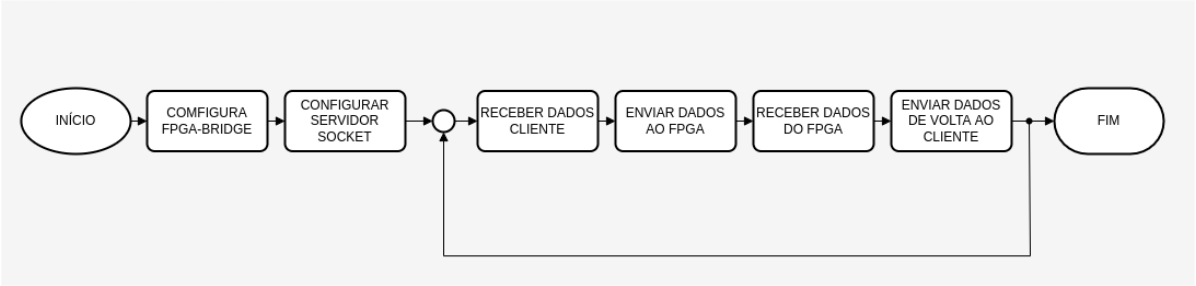
- OpenSSH-Server inicia automaticamente durante o *boot*;
- Compilador gcc 9.3.0; glibc e glib-2.0 da *GNU C Library*, cmake 3.16.5
- Python 3.8 Python3-dev e Python-dev;
- git 2.31, wget 1.20.3, curl 7.69.1;
- Gerenciador de pacotes opkg configurado para pacotes de diferentes distribuições Linux; e
- Checagem e execução do *FPGA-Configuration* da malha do FPGA.

6.3 Servidor: *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, para que o servidor tenha acesso às 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 22.

Figura 22 – Fluxograma simplificado do servidor



Fonte: Elaborado pelo autor

O acesso do servidor ao FPGA é feito através de mapeamento de endereços do linux, uma vez que os SoCs da 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 Platform 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 23 é apresentada 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 23 – 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 de texto 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 texto, com a funções *open*, como pode ser visto na linha 2 da Figura 24.

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*, a qual 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 24, 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 último argumento da função *mmap* é o endereço base da *Lightweight HPS-to-FPGA-Bridge* (0xC0000000), representado pela constante HPS_TO_FPGA_AXI_BASE.

Figura 24 – 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 25 ilustra este processo de escrita e leitura de dados em um IP através de um ponteiro.

Figura 25 – 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

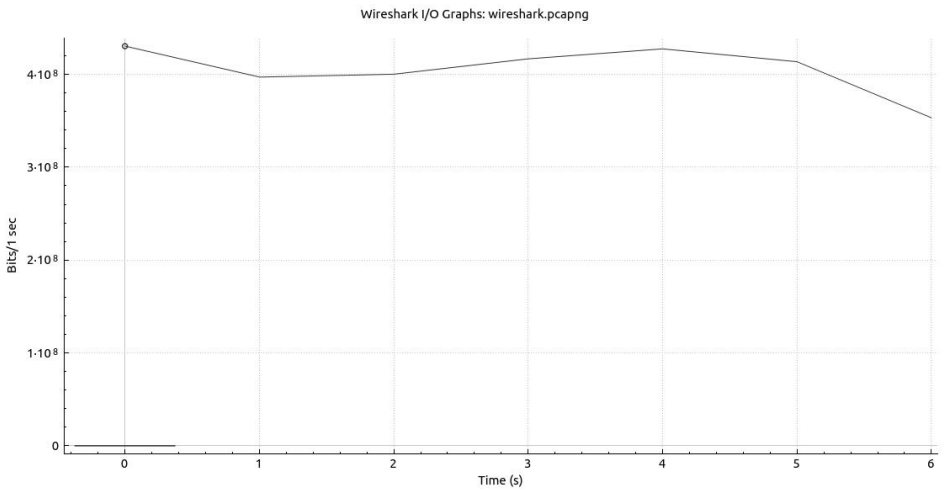
7 Resultados Alcançados

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.

O objetivo do primeiro ensaio era verificar a comunicação completa entre o nó ROS e um IP sendo executado no FPGA. Para isso, foi projetado um circuito simples que calcula o dobro de um número, este circuito foi instanciado no FPGA, 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 pelo FPGA, da mesma maneira que mostra o esquema da Figura 14. A comunicação foi estabelecida e o sistema se comportou como esperado.

O segundo ensaio tinha a intenção de testar a sobrecarga de dados na rede e avaliar o comportamento da mesma. Neste teste foi realizado o envio de uma *stream* de vídeo com resolução de 1280x720 em 15 fps capturada a partir de uma webcam dentro do ambiente ROS, ou seja, um nó ROS acessa a webcam e disponibiliza a *stream* de vídeo em um tópico sendo publicado e uma frequência de 15Hz. A partir deste ensaio foi possível realizar a estimativa de largura de banda utilizada e o *delay* gerado na comunicação. No gráfico da Figura 26 podemos ver a velocidade da comunicação, em bits por segundo, em função do tempo de transmissão. As velocidades alcançadas superaram a barreira dos 400 Mbps.

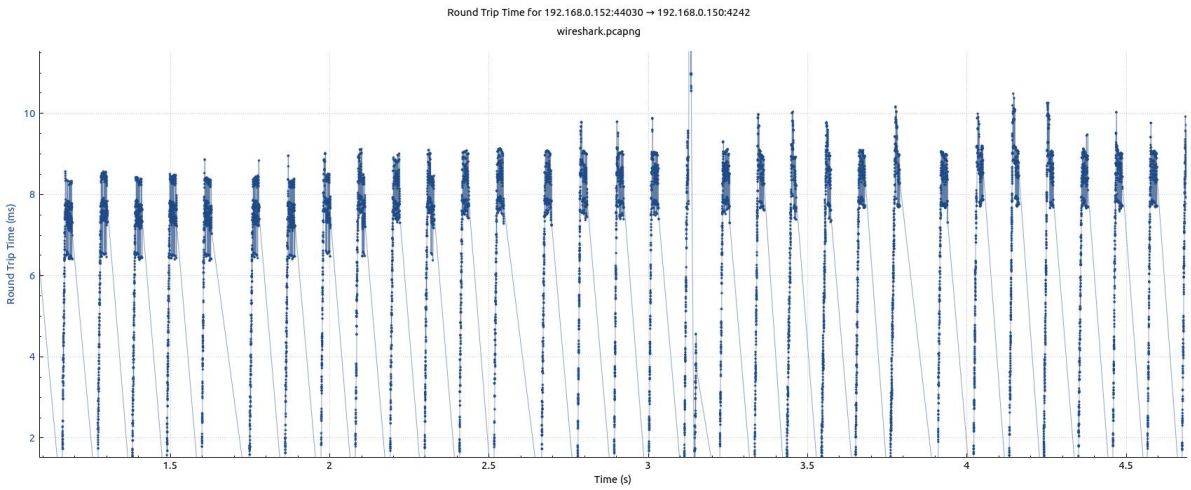
Figura 26 – Gráfico velocidade de transmissão



Fonte: Elaborado pelo autor

Os atrasos gerados podem ser vistos na Figura 27, onde é possível observar o *Round Trip Time* (RTT), que é o atraso total de um pacote TCP, ou seja, o tempo que um pacote leva para ir da origem ao destino e retornar à origem. Os atrasos máximos ficaram por volta de 10ms.

Figura 27 – Atrasos gerados durante a comunicação



Fonte: Elaborado pelo autor

8 Conclusão

Os resultados alcançados para os dois ensaios foram considerados satisfatórios, apesar de, na transferência das imagens, ter sido observado um pequeno *delay*, menor do que 10 ms. Um atraso desse nível não impossibilitaria, por exemplo, o uso do sistema em um robó móvel que usasse o sistema para realizar, através do FPGA, cálculos de odometria visual ou mesmo planejamento de rotas.

Portanto, 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 futuros

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 usado por outros grupos de trabalho que tenham o interesse de adicionar processamento por hardware em seus projetos de robótica. Como por exemplo, os métodos disponibilizados pela biblioteca de comunicação podem oferecer versões que façam uso do protocolo UDP, o que poderia diminuir consideravelmente o *delay* observador durante os ensaios.

Referências

ALTERA. *Cyclone V Hard Processor System: Technical Reference Manual*. [S.l.], 2018. Citado 4 vezes nas páginas 14, 18, 19 e 20.

ALTERA-OPENSOURCE. *linux-socfpga*. 2022. Disponível em: <https://github.com/altera-opensource/linux-socfpga>. Citado na página 43.

ARM. *ARM ® Cortex ® -A9 MPCore Technical Reference Manual*. r4p1. [S.l.], 2016. Citado 2 vezes nas páginas 15 e 16.

ARM. *arm Glossary FPGA*. 2022. Disponível em: <https://www.arm.com/glossary/fpga>. Acesso em: 5 março 2022. Citado na página 17.

ARM, D. *Cortex A9*. 2022. Disponível em: <https://developer.arm.com/Processors/Cortex-A9#Technical-Specifications>. Acesso em: 23 junho 2022. Citado na página 15.

FERGUSON, M. *Rosserial*. 2018. Rosserial. Disponível em: <http://wiki.ros.org/rosserial>. Acesso em: 20 julho 2021. Citado na página 40.

FLYNN, M. J.; LUK, W. *Computer System Designs: System-on-Chip*. 1. ed. New Jersey: Wile, 2011. Citado na página 13.

INTEL. *FPGAs Cyclone® V e FPGAs SoC*. 2022. Disponível em: <https://www.intel.com.br/content/www/br/pt/products/details/fpga/cyclone/v.html>. Acesso em: 23 março 2022. Citado na página 14.

JOSEPH, L. *Mastering ROS for Robotics Programming*. 1. ed. Birmingham: Packt Publishing Ltd, 2015. Citado 2 vezes nas páginas 27 e 28.

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 46.

KOLAK, S. et al. It takes a village to build a robot: An empirical study of the ros ecosystem. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, p. 430–440, 2020. Disponível em: <https://ieeexplore.ieee.org/document/9240632/references#references>. Citado na página 9.

MAAN, L.; BAKER, L. B. *Intel to buy Altera for $16.7 billion in its biggest deal ever*. 2015. Disponível em: <https://www.reuters.com/article/us-altera-m-a-intel-idUSKBN0OH2E020150601>. Acesso em: 23 março 2022. Citado na página 14.

MAHTANI, A. et al. *Effective Robotics Programming with ROS*. 3. ed. Birmingham: Packt Publishing Ltd, 2016. Citado 2 vezes nas páginas 22 e 32.

MARTINEZ, A.; FERNáNDEZ, E. *Learning ROS for Robotics Programming*. 1. ed. [S.l.]: Packt Publishing, 2013. Citado na página 29.

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 37.

NETO, N. P. *interface_socketr*. 2021. Disponível em: <https://github.com/NestorDP/interface_socket>. Citado na página 41.

NETO, N. P. *interface_socket_server*. 2021. Disponível em: <https://github.com/NestorDP/interface_socket_server>. Citado na página 45.

PYO, Y. et al. *ROS Robot Programming*. Nova York: ROBOTIS, 2017. Citado 4 vezes nas páginas 9, 22, 24 e 31.

ROBIN, S. *rsyocto*. 2022. Disponível em: <https://github.com/robseb/rsyocto>. Citado 2 vezes nas páginas 43 e 44.

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 42 e 43.

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

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 22 e 23.

ROS. *catkin/package.xml*. 2019. Disponível em: <http://wiki.ros.org/catkin/package.xml>. Acesso em: 21 setembro 2021. Citado na página 39.

ROS. *Pckages*. 2019. Open Source Robotics Fundation. Disponível em: <http://wiki.ros.org/Packages>. Acesso em: 20 julho 2021. Citado na página 26.

TERASIC. *DE10-Nano User Manual*. 2.2. ed. [S.l.], 2020. Rev. B2/C Hardware. Citado na página 21.

XILINX, A. *Field Programmable Gate Array (FPGA)*. 2022. Disponível em: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>. Acesso em: 5 março 2022. Citado na página 16.