UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

ANNY CAROLINE DA SILVA CRUZ

UM AMBIENTE DE COSSIMULAÇÃO COM HLA PARA VIGILÂNCIA DE ROBÔS COM ROS/*STAGE*

QUIXADÁ

2021

ANNY CAROLINE DA SILVA CRUZ

UM AMBIENTE DE COSSIMULAÇÃO COM HLA PARA VIGILÂNCIA DE ROBÔS COM ROS/*STAGE*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Thiago Werlley Bandeira da Silva

QUIXADÁ

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C96a Cruz, Anny Caroline da Silva.
Um ambiente de cossimulação com HLA para vigilância de robôs com ROS/STAGE / Anny Caroline da Silva Cruz. – 2021.
51 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Computação, Quixadá, 2021.
Orientação: Prof. Dr. Thiago Werlley Bandeira da Silva.

1. Robô autônomo. 2. Vigilância eletrônica. 3. Cossimulação. I. Título.

CDD 621.39

ANNY CAROLINE DA SILVA CRUZ

UM AMBIENTE DE COSSIMULAÇÃO COM HLA PARA VIGILÂNCIA DE ROBÔS COM ROS/*STAGE*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: __/__/____

BANCA EXAMINADORA

Prof. Dr. Thiago Werlley Bandeira da Silva (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr.Antônio Joel Ramiro de Castro
Universidade Federal do Ceará (UFC)

Prof. Me. Francisco Helder dos Santos Filho
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Armando Cavalcante Aguilar
Universidade Federal do Ceará (UFC)

RESUMO

Este trabalho propõe-se o desenvolvimento de um sistema de vigilância de um ambiente interno usando robô autônomo. Esse desenvolvimento teve como objetivo principal a elaboração de um sistema de monitoramento interno. O robô terá que seguir uma rota predefinida no ambiente simulado e realizará a comunicação com a central de controle. Essa comunicação será realizada através de cossimulação. O princípio fundamental da cossimulação é prover suporte à execução de diferentes simuladores cooperativamente. Para isso, utiliza o conceito de *Virtual Bus* com base na especificação padrão para realizar a interoperabilidade e o reúso de código, conhecida como *High Level Architecture* (HLA).

Palavras-chave: Robô autônomo. Vigilância eletrônica. Cossimulação

ABSTRACT

This work proposes the development of a surveillance system for an indoor environment using an autonomous robot. This development had as its main objective the elaboration of an internal monitoring system. The robot will have to follow a predefined route in the simulated environment and will communicate with the control center. This communication will be carried out through co-simulation. The fundamental principle of co-simulation is to support the execution of different simulators cooperatively. For this, it uses the concept of Virtual Bus based on the standard specification to perform interoperability and code reuse, known as High Level Architecture (HLA).

Keywords: Autonomous robot. Electronic surveillance. Cosimulation

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

HLA	*High Level Architecture*
IEEE	*Electrical and Electronics Engineers*
ROS	*Robot Operating System*
PLs	Processos Lógicos
TC	Tempo Contínuo
ED	Eventos Discretos
RMI	*Remote Method Invocation*
CORBA	*Common Object Request Broker Architecture*
DMSO	*Defence Modelling and Simulation Office*
DoD	Departamento de Desefa
OMT	*Object Model Template*
FOM	*Federantion Object Model*
RTI	*Runtime Infraestructure*
API	*Application Programming Interface*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de vigilância são usados com intuito de proteger pessoas e ambientes contra possíveis ameaças. Atualmente, esses sistemas são compostos na maioria das vezes por vigias e câmeras. Esses vigias ficam responsáveis pelo monitoramento do ambiente e, são auxiliados por câmeras, responsáveis pela tomada de decisões mediante de alguma possível ameaça.

Com o avanço da tecnologia e da robótica, a utilização de robôs móveis para realizar tarefas específicas se tornou cada vez mais frequente. Com essa evolução foi surgindo novas categorias de sistemas de vigilância, sistemas mais eficientes e seguros. Existem trabalhos que envolvem técnicas com robôs para vigilância em áreas tanto *indoor* como *outdoor*, um exemplo dessa categoria de sistema é o "RobVigil" desenvolvido pela empresa (CRIIS, 2011). Esse robô de vigilância *indoor* conseguia coletar e receber informações em tempo real a partir de sensores e câmera, bem como funcionar de maneira autônoma ou ser controlado remotamente, conforme ilustrado na Figura 1. Além disso, esse robô mantinha o seu funcionamento independentemente das condições de iluminação. Seu principal propósito era vigilância e segurança de ambientes fechados, realizando o monitoramento do ambiente para conseguir identificar intrusos e algum tipo de mudança no estado do local (temperatura, umidade, nível de monóxido de carbono, entre outros).

Figura 1 – RobVigil.

Fonte: CRIIS (2011)

Existem diversos níveis de complexidade no desenvolvimento de um sistema de vigilância utilizando um robô, tais como:

– alto custos dos componentes;

- comunicação e cooperação entre sistemas diferentes;
- seguir uma rota predefinida com desvio de obstáculos;
- alto localização no ambiente.

Um sistema deve ser capaz de se movimentar no ambiente realizando a identificação de possíveis obstáculos e invasores, em que o robô terá que se comunicar com a central de controle para notificar alguma possível ameaça. A central e o robô são dois sistemas diferentes que precisam trocar informações e se comunicar em um tempo hábil. Para que não tenha necessidade de grandes gastos e seja possível prever se o sistema irá se comportar da maneira esperada, deve-se trabalhar o comportamento real do sistema em um ambiente simulado antes que esse seja fisicamente montado. Para ser possível a construção do ambiente de simulação foi utilizado o *Stage*, que é uma biblioteca de *software* C++ que simula robôs móveis, sensores e objetos em ambientes bidimensionais simulados.

Sistemas que combinam componentes de diferentes domínios que precisam cooperar entre si são chamados sistemas heterogêneos (SCHLAGER, 2008). Segundo Nicolescu *et al.* (2007), a cossimulação representa uma técnica para validação de sistemas heterogêneos, tendo como princípio fundamental prover suporte para execução de diferentes simuladores cooperativamente.

Um padrão bastante conhecido para cossimulação é *High Level Architecture* (HLA). O HLA é um padrão da *Electrical and Electronics Engineers* (IEEE) para interoperabilidade entre simuladores heterogêneos, funciona de forma transparente, síncrona e consistente (IEEE1516.2, 2010). Na arquitetura HLA, cada simulador representa um federado, que possui regras e uma interface para com os outros. Essa arquitetura determina como será feita a comunicação entre os simuladores padronizadamente. Assim, qualquer simulador que possua uma interface de comunicação com o HLA poderá se comunicar com os demais simuladores. Desde que, os mesmos utilizem o mesmo modelo de dados a serem compartilhados. Com base no HLA, para realizar a integração é utilizado uma interface de comunicação denominada *Virtual Bus* implementada com base na especificação da HLA e desenvolvido em Silva *et al.* (2018).

Nesse contexto, existe um problema relacionado a segurança. Por exemplo, se o robô por acaso não consiga se comunicar com a central de controle para comunicar sobre um suposto intruso, o mesmo passaria despercebido e isso ocasionaria em uma falha de segurança. O trabalho teve como foco dois dos problemas encontrados nessa categoria de sistema de vigilância. Primeiro, o robô terá que realizar uma rota pré-definida. Segundo, o robô terá que se comunicar

com a central de controle em um tempo hábil. A solução é executada por cossimulação do ambiente de simulação (do inglês *testbench*) com a implementação do robô simulado. O robô realizará uma rota pré-definida, analisada com antecedência pelo *testbench*, verificando se o robô realiza o percurso estabelecido através da troca de mensagens. As mensagens são trocadas durante a cossimulação por meio da interface do *Virtual Bus*. Enquanto a criação do *testbench*, será utilizado o *Stage* e para realizar os comandos do robô será utilizado o *Robot Operating System* (ROS). O ROS é um *framework* de robótica destinado a facilitar o desenvolvimento de *software* em robôs. Tal *framework* fornece bibliotecas e ferramentas que auxiliam na criação de aplicações, abstração de *hardware*, *drives* de dispositivos, entre outros.

1.1 Objetivos

Esse trabalho tem como objetivo principal: desenvolver um ambiente para simulação e testes de vigilância de um robô, utilizando cossimulação entre o robô e a central de controle.

Os objetivos específicos deste trabalho são os seguintes:

1. Desenvolver o ambiente de simulação no Stage;
2. Viabilizar a comunicação entre o ROS/*Stage*;
3. Viabilizar a comunicação com a interface do *Virtual Bus*;
4. Viabilizar a cossimulação entre ROS/*Stage* utilizando *Virtual Bus*;
5. Realizar a validação do sistema desenvolvido.

1.2 Organização do Trabalho

Este trabalho está organizado da seguinte forma:

– No Capítulo 2 aborda-se a fundamentação teórica, com os conceitos fundamentais para a compreensão da proposta descrita.
– No Capítulo 3 tratam-se os trabalhos relacionados, comparando os aspectos comuns ou divergentes entre eles e o trabalho aqui proposto.
– No Capítulo 4 descreve-se o processo para a criação do ambiente criado.
– No Capítulo 5 descrevem-se os experimentos realizados para o desenvolvimento desse trabalho e seus respectivos resultados.
– No Capítulo 6 tratam-se as considerações finais sobre o trabalho proposto, limitações e propostas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEORICA

Neste capítulo são abordados os conceitos fundamentais para a compreensão da proposta descrita.

2.1 Simulação Distribuída

Neste trabalho, a simulação distribuída se fará presente, pois, o ROS/*Stage* e a Central de Controle serão simulados em máquinas diferentes e irão se comunicar entre si através do *Virtual Bus*.

2.1.1 Simulação

Para entender o que é simulação, se faz necessário entender o que é um modelo, pois é um conceito que a compõe. Modelo é uma aproximação, representação ou idealização de aspectos selecionados da estrutura, comportamento, operação, ou características de um processo do mundo real. Então, simulação é o processo de desenvolver um modelo matemático ou lógico de um sistema real e então realizar experimentos de modo a tentar prever o comportamento preciso do sistema Schlager (2008).

Segundo Schlager (2008), as simulações podem ser classificadas de acordo com tipo, distribuição e domínio. Esses tópicos são:

- **Tipo da simulação:** existem duas categorias de simulações, as discretas e contínuas. Nas discretas as variáveis do modelo de simulação são alteradas apenas em certos instantes de tempo. Já nas contínuas as varáveis do modelo tem seus valores alterados continuamente ao longo do tempo de simulação.
- **Distribuição da simulação:** em relação à distribuição, a simulação pode ser centralizada ou distribuída. A centralizada usa apenas um processador, já a distribuída utiliza múltiplos processadores interligados em rede.
- **Domínio da simulação:** os possíveis domínios de uma simulação serão descritos a seguir:
  - Simulação de ambiente: viabiliza a simulação física de um ambiente de sistema de tempo real;
  - Simulação de protocolo: envolve a validação de características utilizadas em um protocolo de comunicação;
  - Simulação de rede: possui propriedades de uma conexão de rede fornecida;

- Simulação de *cluster*: emula o desempenho de um ou mais nós de um sistema distribuído em tempo real.

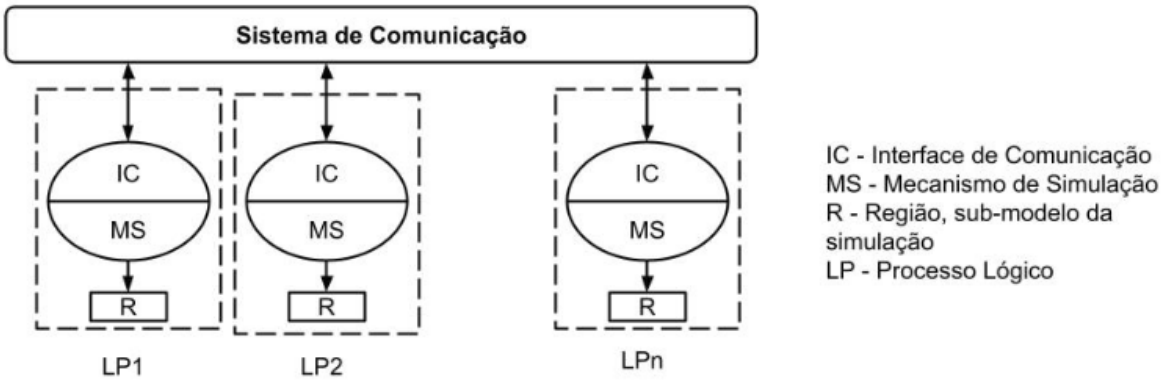
2.1.2 Princípios da Simulação Distribuída

A simulação distribuída é baseada na decomposição do modelo de simulação em Processos Lógicos (PLs), executados em diferentes processadores. Ela é empregada para diminuir o esforço computacional de simulações realistas, fornecendo com isso resultados mais confiáveis em um tempo significativamente mais rápido de processamento (DOTA, 2001).

A simulação distribuída simplifica a interação entre simuladores executados em máquinas de diferentes fabricantes. Ela é usada principalmente para oferecer alguns benefícios como: a redução de tempo de execução; a simulação geograficamente distribuída; possibilidade de integração de diferentes tipos de simuladores (SILVA JUNIOR, 2015). Outro benefício é uma maior tolerância a falhas.

A Figura 2 ilustra uma arquitetura da simulação dos vários PLs de um sistema modelado. Um conjunto de PLs podem executar eventos paralelos de modo assíncrono ou síncrono. A interface de comunicação fornece ao PL mecanismos para a troca de dados com os outros PLs. Cada PL atua em uma região, em uma parte do modelo da simulação, processando os seus eventos locais e gerando eventos remotos (THONDUGULAM *et al.*, 1999).

Figura 2 – Arquitetura de uma simulação de Processos Lógicos



Fonte: Adaptado de Dota (2001)

Para se tornar possível que modelos simulados em diferentes máquinas sejam sincronizados, são utilizados mecanismos de sincronização fundamentais para a simulação distribuída. Como as simulações são compostas por vários PLs que se comunicação através de eventos ou

mensagens, o uso da sincronização garante que eventos realizados por processos lógicos ocorram em seu determinado tempo de forma ordenada (FUJIMOTO, 2001). Ainda segundo Fujimoto (2001), existem dois tipos de algoritmos de sincronização os conservativos e os otimistas.

2.1.2.1 Sincronização Conservativa

Os primeiros algoritmos de sincronização foram baseadas em abordagens conservativas, significando assim que o algoritmo de sincronização tem cuidado para não violar a restrição de causalidade local. Por exemplo, suponha que uma PL está em simulação com tempo = 10, ou seja, rótulo de tempo (*timestamp*) associado igual a 10, e está pronto para processar o próximo evento com *timestamp* = 15 definido. Então, o algoritmo de sincronização deve garantir que nenhum evento com *timestamp* inferior a 15 seja recebido antes que o evento já definido, seja processado (FUJIMOTO, 2001). Os protocolos conservativos evitam que erros de causa e de efeito ocorram, avisando assim quando é seguro processar um evento (DOTA, 2001). Ou seja, um PL não pode processar um evento até que esteja garantido que seja seguro.

2.1.2.2 Sincronização Otimista

Diferente da sincronização conservativa, os protocolos otimistas usam uma estratégia de detecção e recuperação de erros, os erros de causa e de efeito são identificados e um mecanismos de *roolback* é utilizado para recuperação (DOTA, 2001). As abordagens otimistas oferecem duas vantagens em comparação com a conservativa. Primeiro, podem explorar maiores graus de paralelismo, os mecanismos otimistas podem processar eventos simultaneamente. Enquanto, os métodos conservativos devem sequencializar a execução. Em segundo lugar, o mecanismo conservador, geralmente, depende de informações específicas, a fim de determinar quais são os eventos seguros para processar (FUJIMOTO, 2001).

2.2 Cossimulação

2.2.1 Visão Geral

A cossimulação tem como princípio básico o suporte para execução cooperativa de diferentes simuladores, em que cada simulador fica responsável pela execução de uma parte do sistema desenvolvido (OLIVEIRA, 2016). O processo de cossimulação precisa de simuladores

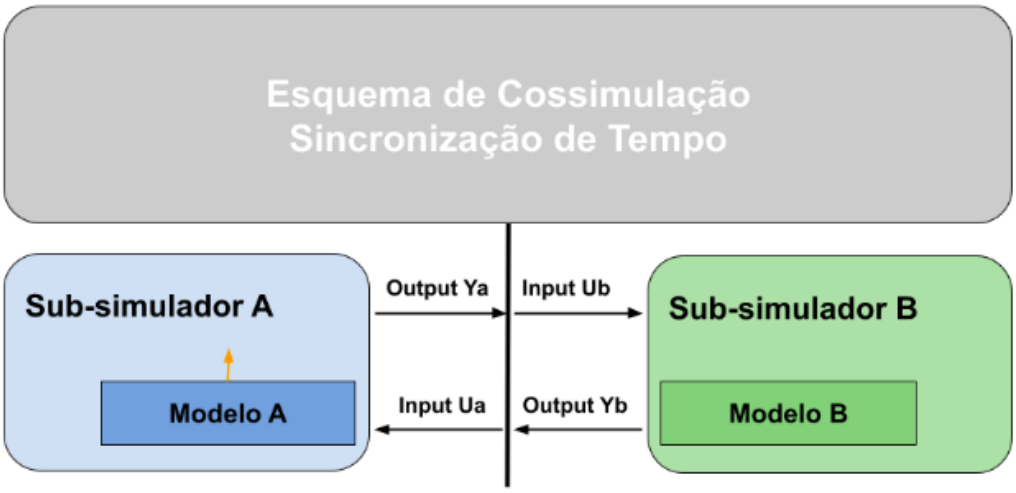
capazes de simular simultaneamente as partes de *software* e *hardware*, garantindo assim a compatibilidade na interação entre as partes.

De acordo com Nicolescu *et al.* (2007), a cossimulação, atualmente, representa uma das técnicas de validação mais populares para sistemas heterogêneos, permitindo que componentes heterogêneos consigam simular de foma conjunta com diferentes tipos de modelos de execução. Cada componente da cossimulação fica responsável por executar de maneira individual uma parte do sistema, que conterá uma linguagem, modelo computacional e nível de abstração específico.

A cossimulação utiliza procedimentos aplicados em simulações distribuídas, já que a cossimulação tem como propósito interligar simuladores de forma que esses consigam trabalhar de forma conjunta (BISHOP; LOUCKS, 1997). Essa interligação pode ocorrer centralizada, ou seja, quando os simuladores estão sendo executados em uma mesma máquina, ou descentralizada, quando os simuladores estão sendo executados em máquinas distintas.

Para Bishop e Loucks (1997), a cossimulação inclui acoplar simulações de Tempo Contínuo (TC) e de Eventos Discretos (ED) rodando em diferentes ferramentas, conforme mostrado na Figura 3, em que a variável de tempo deve ser sincronizada. O modelo de TC é utilizado para mecanismos robóticos dinâmicos e controlados por algoritmos, e o modelo de ED para controle de decisões e lógica implementada no *software* embarcado.

Figura 3 – Arquitetura de Sincronização de Tempo

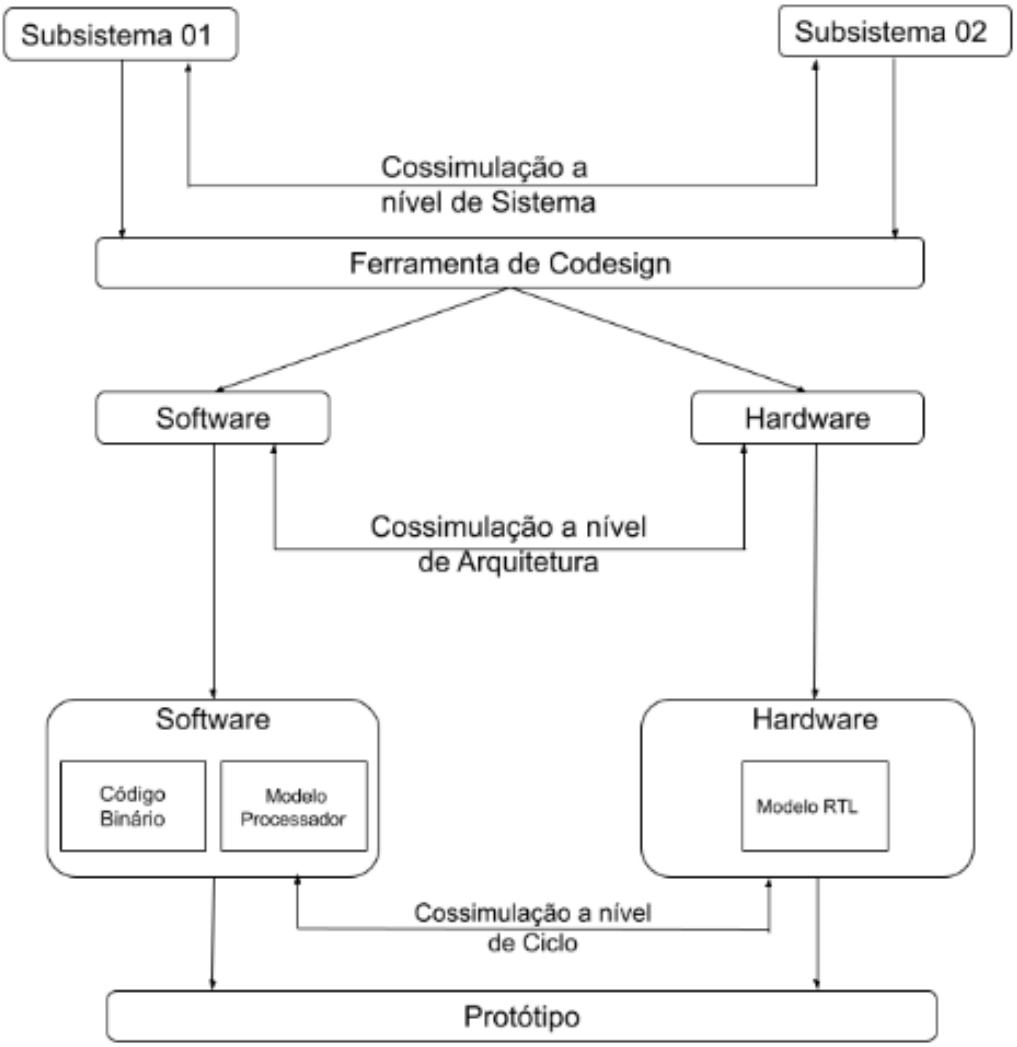


Fonte: Adaptado de Bishop e Loucks (1997)

2.2.2 Níveis de Abstração

Para Amory *et al.* (2002), existem três níveis diferentes de abstração em que a cossimulação pode ser realizada: a nível de sistema, a nível arquitetônico e a nível de ciclo, como ilustrado na Figura 4.

Figura 4 – Cossimulação em diferentes níveis de abstração



Fonte: Adaptado de Amory *et al.* (2002)

Em nível de sistema, se tem como objetivo principal caracterizar a funcionalidade do sistema. Nesse nível, o *software* do sistema é descrita usando uma linguagem de programação como C/C++. A parte do *hardware* é descrita usando VHDL ou Verilog, sendo a linguagem de descrição de *hardware*. O protocolo de comunicação entre o *software* e o *hardware* é abstraída neste nível. Em nível arquitetônico, a comunicação entre *hardware*/*software* é considerada com o dispositivo de destino. Em nível de ciclo, os componentes de *software* são simulados usando

código binário executando em um simulador de nível de ciclo no processador do *host* alvo.

2.2.3 Classificações de Cossimulação

Segundo Amory *et al.* (2002), a cossimulação pode ser classificada de acordo com três critérios: quantidade de simuladores, modelo de tempo de execução e distribuição.

2.2.3.1 Quantidade de Simuladores

Os ambientes de cossimulação tentam preencher o vazio existente nas ferramentas de modelagem e superar a falta de modelos detalhados de computação e rede de comunicação. A principal característica que o ambiente de cossimulação usa de simulações distribuídas é em relação à sincronização, dado que em um único sistema pode ter diferentes tipos de simuladores e cada simulador pode possuir velocidade de execução diferente (SILVA JUNIOR, 2015). Dessa forma, a sincronização garantirá que os eventos disparados pelos simuladores durante a simulação ocorra em seu devido tempo.

Um ambiente de cossimulação pode ser desenvolvido utilizando duas abordagens, simulação homogênea ou heterogênea. A homogênea se dá quando é utilizado somente uma máquina de simulação, e a heterogênea é quando se utiliza duas ou mais máquinas de simulação diferentes. Ambiente de cossimulação homogênea por usar somente uma máquina de simulação não precisa se preocupar com a sincronização, pois todos os dados do sistema e subsistemas encontram-se em uma mesma máquina. Além disso, traduz toda linguagem de descrição dos módulos que compõem o sistema em um único idioma capaz de descrever todo o sistema, denominado como formato intermediário. Nesse tipo de abordagem só é necessário apenas um simulador.

Já no ambiente de cossimulação heterogêneo, é possível utilizar vários mecanismos de simulação, ou seja, podem ser interligados muitos simuladores. Além disso, emprega um simulador dedicado para cada idioma empregado. Diante disso, se faz necessário ter controle sobre a sincronização, comunicação e tipos de mensagens trocadas entre os simuladores e modelos de *hardware*. Para conseguir lidar com esses problemas, se faz necessário que o ambiente de cossimulação possua uma semântica que seja compreendida por todos os envolvidos no sistema Amory *et al.* (2002).

2.2.3.2 Modelo temporal de execução

Neste critério considera o modelo de execução, em que a cossimulação pode ser funcional e temporal. O modelo funcional é utilizado para validar os sistemas em níveis mais altos de abstração, onde detalhes temporais e comunicação são negligenciados. O modelo temporal permite uma validação mais precisa, pois consideram informações de tempo, arquitetura do processador e modelo de barramento quando os simuladores interagem.

2.2.3.3 Distribuição

Segundo Amory *et al.* (2002) a distribuição de uma cossimulação pode ser distribuída ou local. Uma cossimulação distribuída permite execução paralela de simuladores em máquinas geograficamente distribuídas através de redes do tipo *Local Area Network* (LAN) ou *Wide Area Network* (WAN). Os benefícios dessa abordagem são: (i) descentralização do projeto; (ii) projeto e validação de um sistema em desenvolvimento por equipes distribuídas geograficamente; (iii) gerenciamento de propriedade intelectual — simulações podem ser realizadas sem o repasse de códigos-fonte; (iv) simuladores podem ser instalados em algumas máquinas apenas; (v) compartilhamento de recursos. Já em uma cossimulação local, não possui *overhead* de comunicação, dado que não é utilizada uma rede para troca de informações. Contudo, não possui as vantagens presentes nas cossimulações distribuídas.

2.3 *High Level Architecture*

Neste trabalho, a HLA será utilizado para fazer a interligação entre o ROS/*Stage* e a Central de Controle. A HLA fornece uma arquitetura aberta de troca de informação, gerenciamento de tempo e sincronização entre todos os modelos envolvidos. Apesar de ter outras opções de arquiteturas que forneçam as mesmas funcionalidades do HLA, esse foi escolhido por conta de ser uma arquitetura que é específica para simulação distribuída.

Dentro de um ambiente de cossimulação onde tem simuladores se comunicando entre si, existe a necessidade que esse ambiente consiga entender a semântica de todos os modelos envolvidos para que essa comunicação se torne eficaz. Para realizar essa comunicação, se torna fundamental o uso de interfaces adequadas para cada um dos envolvidos.

Para realizar essa comunicação entre componentes em uma simulação distribuída, existem algumas arquiteturas que oferecem recursos necessários para essa comunicação. O

Remote Method Invocation (RMI), o *Common Object Request Broker Architecture* (CORBA) e a HLA, são as mais utilizadas atualmente. Dentre essas três que foram citadas a HLA é a única arquitetura específica para simulação distribuída, o CORBA e o RMI são arquiteturas genéricas para aplicações distribuídas (SILVA JUNIOR, 2015).

A arquitetura HLA foi definida pelo *Defence Modelling and Simulation Office* (DMSO), como uma especificação de uma arquitetura padrão para administrar as informações compartilhadas entre os diferentes simuladores O avanço de tempo de simulação foi desenvolvido pelo Departamento de Desefa (DoD) dos Estados Unidos, com o objetivo de prover a interconexão de diversos simuladores militares. Tem como principais características a reutilização e a interoperabilidade. Essa arquitetura é definida por três padrões da IEEE, o primeiro documento IEEE1516 (2000) especifica o *framework* de forma geral e suas principais regras, o segundo documento IEEE1516.1 (2000) especifica a interface para comunicação entre os simuladores, e o terceiro documento IEEE1516.2 (2010) especifica um padrão de documentação para definição do modelo de dados (*Object Model Template* (OMT)) transferidos entre os simuladores.

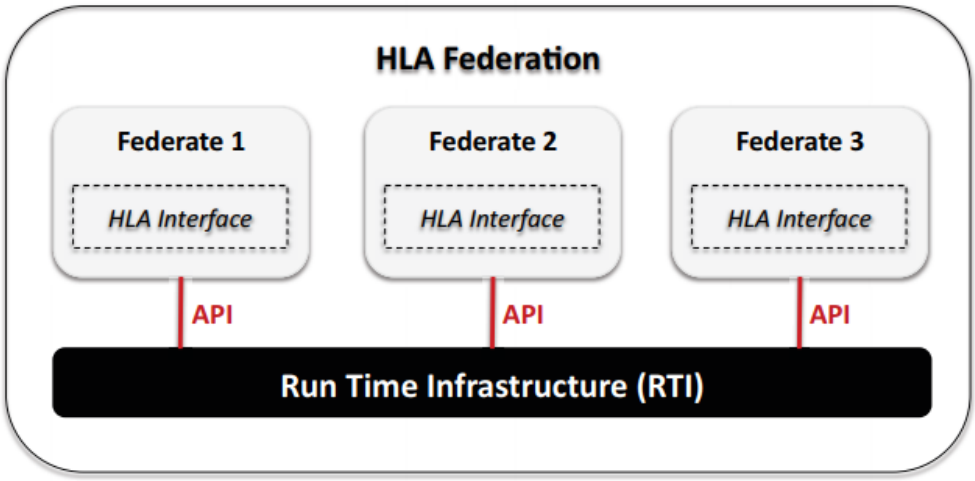
- **Regras do HLA**: especifica como deveram ser realizadas as interações entre os federados em uma federação.
- **Object Model Template**: define a estrutura necessária para cada *Federantion Object Model* (FOM), o FOM cria uma especificação para a troca dos dados utilizando um formato comum e padronizado.
- ***Interface Specification***: especifica a interface entre os federados e o *Runtime Infraestructure* (RTI).

Os simuladores quando compõem uma simulação que utiliza HLA, deve utilizar a Infraestrutura de Tempo de Execução RTI para se tornar possível a comunicação dele com outros simuladores, ele realizar a troca de dados entre si através de um protocolo de rede, denominada TCP/IP. O RTI é um componente fundamental do HLA, ela é a especificação de um *software* que fornece os serviços de interface para sincronização e realiza o controle das trocas de mensagens entre os federados em uma simulação, sendo esta responsável pelo gerenciamento do tempo total de simulação, além de fornecer o protocolo de comunicação, assim todos os simuladores terão acesso à estrutura global do HLA (SILVA JUNIOR, 2015; IEEE1516, 2000).

Para este trabalho foi escolhido utilizar a implementação de código aberto chamada CERTI. O CERTI foi desenvolvido pela ONERA, um centro francês de pequisas aeroespaciais Noulard *et al.* (2009).

Na Figura 5 é mostrado a arquitetura geral de uma federação, em que podem ser visualizados os principais componentes de uma federação que são os federados e a RTI. Pode-se notar que cada simulador é conectado a RTI, os simuladores são chamados federados. Um federado pode ser definido como qualquer *software* que implemente os serviços da interface HLA e seja capaz de se unir a uma federação para enviar e receber informações de outros federados. A federação é o conjunto de todos os federados que atuam juntamente em uma simulação visando alcançar um objetivo em comum (SILVA JUNIOR, 2015; IEEE1516.1, 2000; ABREU, 2019).

Figura 5 – Arquitetura Geral de uma Federação em HLA.



Fonte: Lasnier *et al.* (2013)

A interface HLA disponibiliza serviços que são implementados pela RTI e devem ser chamados pelos federados como, por exemplo, para criar uma nova federação, enviar dados, receber dados, conectar-se a uma federação ativa, etc. Esses federados no que lhe concerne devem implementar os serviços da interface HLA para serem invocados pela RTI através de mensagens de chamadas de retorno (*callbacks*). Eles são chamados no federado em resposta a alguma solicitação realizada pelo próprio federado, ou para recebimento de informações de interesse do federado, ou ainda quando a RTI informa o federado sobre a situação da federação (IEEE1516.1, 2000; ABREU, 2019; BARROS, 2017).

Os serviços invocados pelo federado e implementados na RTI são listados na Tabela 1, já os serviços invocados pela RTI e implementados no federado são listados na Tabela 2. Esses serviços serão melhor entendidos com o exemplo apresentado a seguir.

Segundo Barros (2017), conforme mostrado na Figura 6, é tratado um exemplo de criação de uma federação envolvendo dois federados: Federado A e Federado B. Nesse exemplo,

o autor nos mostra o passo a passo da sequência de chamados para a criação de uma federação envolvendo os Federados A e B, e a RTI.

Tabela 1 – Serviços invocados pelo federado e implementados na RTI.

Serviço	Descrição
synchronizationPointRegistratioSucceeded	Indica que o registro do ponto de sincronização foi registrado com sucesso.
announceSynchronizationPoint	Anuncia aos federados de uma federação qual é o ponto de sincronização.
federationSynchronized	Informa aos federados que a federação está sincronizada porque o ponto de sincronização foi alcançado por todos.
discoverObjectInstance	Informa ao federado associado a existência de uma instância de um objeto
turnUpdatesOnForObjectInstance	Informa ao federado associado que os valores atribuídos do objeto são necessários para a federação.
reflectAttributeValues	Recebe uma atualização com os novos valores dos atributos de interesse do federado.
timeRegulationEnabled	Indica que um pedido prévio para habilitar a regulação do tempo foi aceito.
timeConstrainedEnabled	Indica um pedido prévio para habilitar a restrição de tempo foi aceito.
timeAdvanceGrant	Informa que um pedido prévio para avançar o tempo lógico do federado foi aceito.

Fonte: Barros (2017)

No passo 1, o Federado A cria a federação através da invocação do serviço *createFederationExecution*. Esse procedimento é realizado pelo primeiro federado ao se conectar com a RTI. Uma vez que a federação está criada é possível unir qualquer federado à federação através da chamada do método *JoinFederationExecution*. Nos passos 2 e 3 os dois federados invocam o método *JoinFederationExecution* se unindo assim a federação. Nos passos 4 a 7, os federados descrevem a classe e os atributos que desejam manipular. Isso é feito com a invocação dos métodos *getObjectClassHandle* e *getAttributeHandle* respectivamente.

Nos passos 8 a 19, cada federado publica qual é a classe para qual fornecerá informações e indica qual é a classe que assina para receber informações de outro federado, isso é feito através da invocação dos serviços *publishObjectClass* e *subcribeObjectClassAttributes*. Logo após, no passo 20, o Federado A solicita o registro de um rótulo que será o ponto de sincronização através da invocação do método *registerFederationSynchronizationPoint*. No passo 21 a RTI confirma se o registro do Federado A foi realizado com sucesso , ao invocar o método *synchronizationPointRegistrationSucceeded*.

Tabela 2 – Serviços invocados pela RTI e implementados no federado.

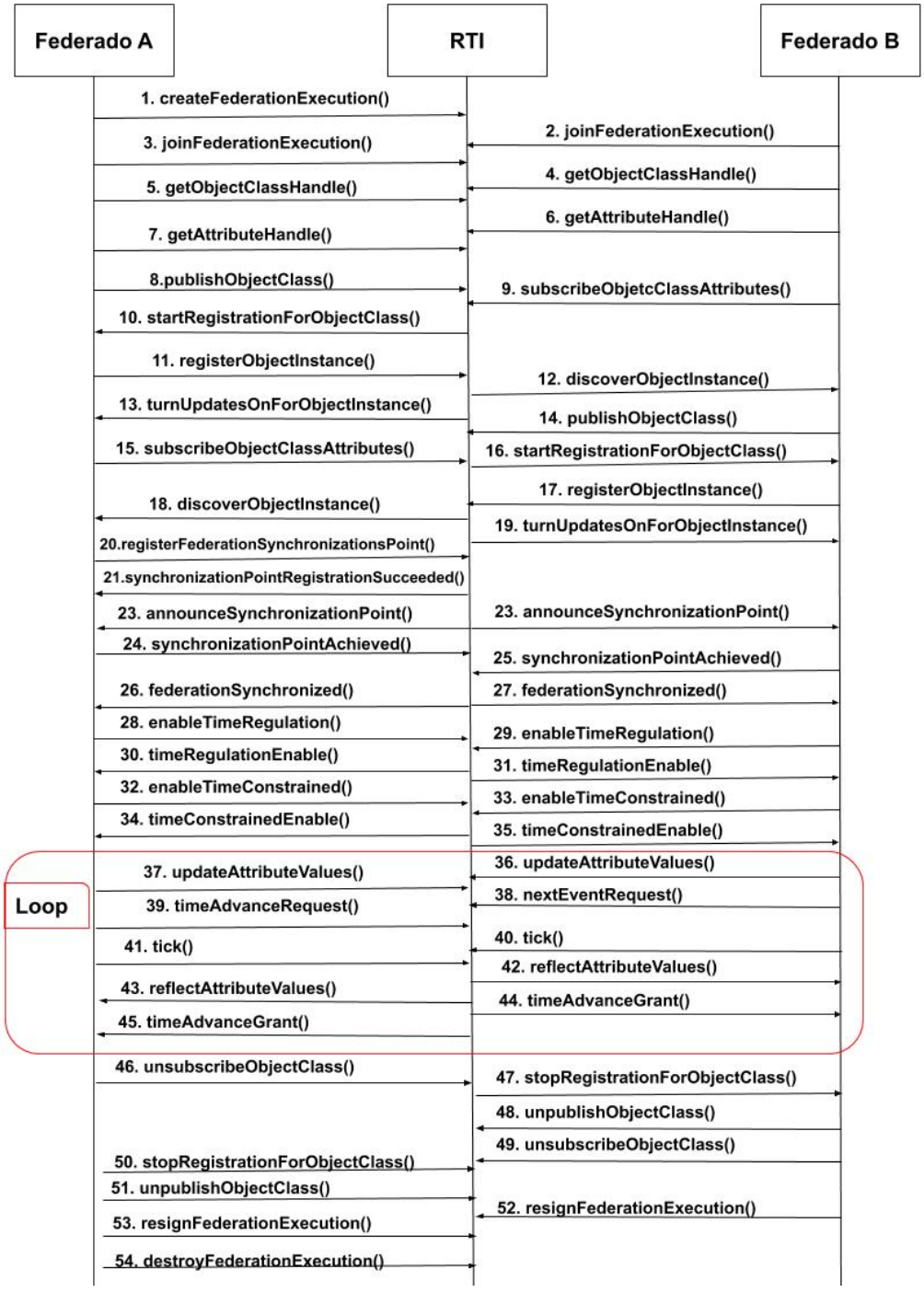
Serviço	Descrição
createFederationExecution	Cria uma nova federação e para isto um FOM deve ser fornecido.
joinFederationExecution	Associa o federado a uma federação.
resignFederationExecution	Desassocia o federado de uma federação.
destroyFederationExecution	Destrói uma federação da RTI.
registerFederationSynchronizationPoint	Solicita o registro de um rótulo que será o pondo de sincronização.
synchronizationPointAchived	Informa a RTI que o federado alcançou ponto de sincronização registrado.
getObjectClassHandle	Retorna a classe do objeto que será manipulado.
getAttributeHandle	Retorna o atributo que será manipulado.
publishObjectClass	Publica a classe que contém as informações que são fornecidas pelo federado.
subscribeObjectClass	Assina o federado como interessado nas instâncias de uma classe.
unsubscribeObjectClass	Cancela a assinatura do federado como interessado nas instâncias de uma classe.
unpublishObjectClass	Cancela a publicação da classe que possui as informações fornecidas pelo federado.
registerObjectInstance	Registra uma instância da classe publicada com as informações do federado.
updateAttributeValues	Atualiza os valores dos atributos da instância registrada para a RTI.
enableTimeRegulation	Habilita o federado a regular o avanço de tempo em outros federados.
enableTimeConstrained	Habilita o federado a se submeter a restrição do tempo.
timeAdvanceRequest	Solicita o avanço de tempo lógico do federado.
nextEventRequest ou nextMessageRequest	Solicita o avanço do tempo lógico do federado para o tempo da próxima mensagem.

Fonte: Barros (2017)

Em seguida nos passos 22 e 23, a RTI invoca o método *announceSynchronizationPoint* em todos os federados, para poder anunciar a todos os federados da federação qual é o ponto de sincronização registrado. Então, nos passos 24 e 25 os federados informam que alcançaram o ponto de sincronização através da chamada do método *synchronizationPointAchieved* da RTI. Já nos passos 26 e 27 a RTI informa que a federação está sincronizada ao invocar o método *federationSynchronized* nos federados. Então, cada federado habilita suas restrições relacionadas ao tempo nos passos 28 a 35.

A troca de informações entre os federados ocorrem nos passos de 36 a 45, exibida no fragmento *loop*. Aqui os Federados A e B publicam novos valores na RTI através do método *updateAttributeValues* e logo em seguida solicitam o avanço do tempo com a chamada do método

Figura 6 – Diagrama de sequência de uma simulação HLA.



Fonte: Barros (2017)

timeAdvanceRequest ou *nextEventRequest*. A RTI invoca o método *reflectAttributeValues* nos federados, para atualizá-los das informações as quais eles têm interesse, e libera o avanço no tempo com a chamada do serviço *timeAdvanceGrant*.

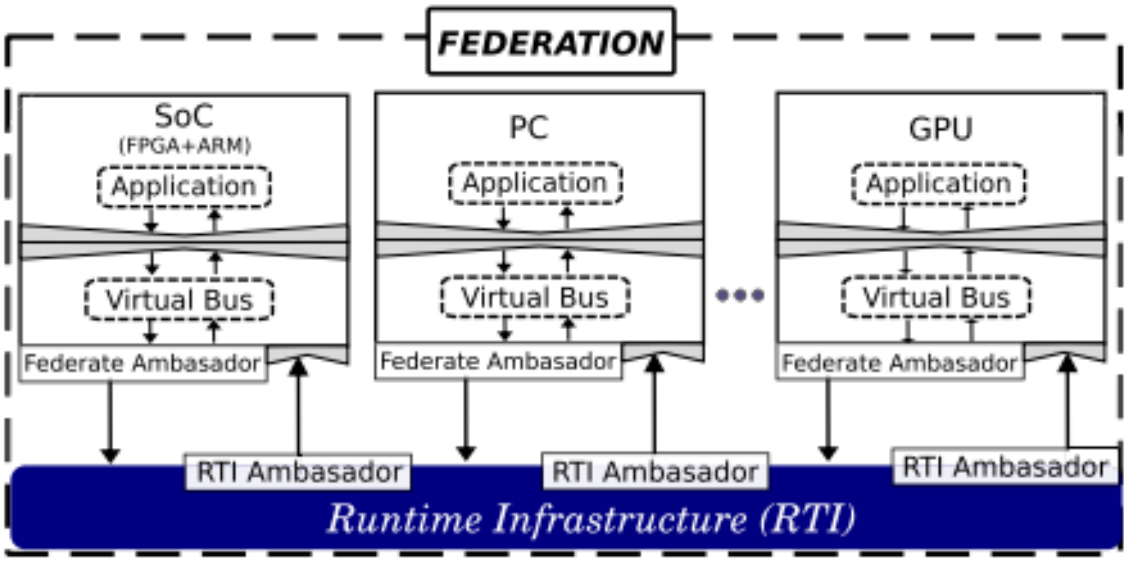
A finalização da federação acontece nos passos 46 ao 54, quando os federados cancelam a assinatura que possuíam para os objetos através da chamada ao método *unsubscribeObjectClass* e, após isso, cancelam a publicação de informações para a classe que forneciam

informações através da invocação do método *unpublishObjectClass*. Em seguida, os federados se desassociam da federação ao invocar o método *resignFederationExecution* e, quando a federação não possui mais nenhum federado associado, então é destruída através da invocação do método *destroyFederationExecution* na RTI.

2.3.1 Virtual Bus

Baseando-se nas especificações HLA mostradas anteriormente, em Silva *et al.* (2018), foi desenvolvido uma interface limpa para intercomunicação entre diferentes simuladores, chamada *Virtual Bus*. A principal tarefa consiste em fornecer uma *Application Programming Interface* (API) sobre a HLA, tornando assim as operações de trocas de dados mais transparentes para o usuário. Ela é responsável por receber e enviar dados na rede, além de garantir a interoperabilidade entre os diferentes simuladores.

Figura 7 – Arquitetura Geral do *Virtual Bus*.



Fonte: Silva *et al.* (2018)

A Figura 7 ilustra um exemplo da extensão do HLA, o *Virtual Bus*, onde mostra diferentes arquiteturas se conectando através do *Virtual Bus* construído sobre o ambiente CERTI/HLA. A API do *Virtual Bus* é composta por três funções essenciais: o *write*, *read* e *advancedTime*. O *write* é a função de escrita, e é responsável pelo envio dos dados. O *read* é a função de leitura, e é responsável por lê os dados recebidos do RTI. O *advancedTime* é a função encarregada por solicitar o avanço de tempo.

No Código-fonte 1 é descrito o trecho referente a função *writeData*, que á a função

de escrita, que cria um objeto para manipular seus atributos, nela também é lido o tempo HLA, realiza uma atualização de seus dados e solicita o avanço de tempo *advanceTime*. No Código-fonte 2 é descrito o trecho referente a função *readData*, que verifica se algum dado foi recebido, caso sim, ele retorna esses dados.

Com isso, para que um federado consiga ingressar em uma federação, terá que chamar a função *runFederate* Código-fonte 3 da API do *Virtual Bus*. Esta função solicita ao *RTIAmbassador* que ele crie a federação, caso ela não exista e crie o *FederateAmbassador* solicitado. Com isso o federado é criado e se torna parte da federação, enviando um aviso ao RTI que está pronto para ser executado, porém, só será executado quando todos os outros federados alcançarem o ponto de sincronização, chamado *READ_TO_RUN*. Quando um federado finalmente chama o método *publishAndSubscribe*, a política de tempo é definida e todos os objetos de que recebe e envia atualizações são registrados Silva *et al.* (2018).

Código-fonte 1 – Trecho do Código da função *writeData*

```
1
2  writeData(id, data){
3      attributes = new RTI::Attribute();
4      attributes->add(id);
5      attributes->add(data[0]);
6      attributes->add(data[1]);
7      attributes->add(data[2]);
8      ...
9      attribute->add(data[N]);
10     time = fedamb->federateTime();
11     rtiamb->updateValues(oHandle, *attributes, time);
12     fedamb->advanceTime();
13 }
```

Código-fonte 2 – Trecho do Código da função *readData*

```
1  Object readData(id, data){
2      if(fedamb->hasRedeiveData(id)){
3          data = fedamb->getReceivedData(id, data);
```

```
4           return data;
5       }
6       else
7           return null;
8   }
```

Código-fonte 3 – Trecho do Código da função *runFederation*

```
1   runFederate(char* federateName){
2       rtiamb = new RTI::RTIambassador();
3       rtiam-> createFederationExecution();
4       fedamb = new FederateAmbassador(federateName);
5       rtiamb->joinFederationExecution(federateName, fedamb);
6       rtiamb->synchronizationPoint(READ_TO_RUN);
7       publishAndSubscribe();
8       oHandle = registerObject();
9   }
```

Essas são as principais funções do *Virtual Bus*, o *Virtual Bus* será adaptado para uso nesse trabalho, onde teremos dois federados, ROS/*Stage* e Central de Controle cossimulando.

2.4 Ferramentas

Neste trabalho, o ROS foi utilizado para implementar os comandos do robô, já o simulador *Stage* para o desenvolvimento do ambiente de simulação. Uma das motivações para utilizar esse simulador é por ser mais leve computacionalmente que os demais simuladores. Também por já ser integrado ao ROS, facilitando ainda mais a comunicação com o ambiente que será desenvolvido.

2.4.1 Robot Operating System (ROS)

O *Robot Operating System* (ROS) é um meta-sistema operacional livre e de código aberto, desenvolvido pela *Open Source Robotics Fundation*. Foi criado para facilitar o desenvolvimento de *softwares* para criação de projetos voltados à robótica, fornecendo para seus

usuários diversos serviços como, abstração de *hardware*, controle de dispositivo de baixo nível, troca de mensagens entre processos, gerenciamento de pacotes, entre outros. Facilitando assim, o desenvolvimento de aplicações para robôs de maneira mais rápida e eficiente Koubâa *et al.* (2017).

ROS abrange diferentes bibliotecas já implementadas como o Python, C++ e Lisp, contudo, existem duas (Java e Lua) que ainda estão em fases experimentais. O ROS concede recursos e diversas bibliotecas para obter, criar e executar códigos em diferentes máquinas. Além disso, o ROS torna possível a reutilização de códigos desenvolvidos e disponibilizados gratuitamente por outros usuários, contendo assim uma grande comunidade de usuários ativos.

2.4.1.1 Arquitetura

O ROS foi desenvolvido para plataformas baseadas em UNIX, porém também pode ser utilizadas em outras plataformas como, Microsoft Windows, Linux, Mac OS X, OpenSUSE e Arch, sendo estes ainda considerados como experimentais, segundo (CARRERA, 2013).

Segundo Araújo (2019) e Level (2013), os principais conceitos existentes na arquitetura do ROS são:

• Master: o master registra nomes, publicações, subscrição, armazena tópicos e informações de registros de serviços para os nós.

• Nós: o nó é um arquivo executável que pode corresponder a um sensor, processamento, funções específicas, etc. Quando um nó é executado o Master é notificado, que por sua vez é um nó e um serviço de registro possibilitando assim que outros nós se comuniquem entre si e troquem dados entre eles através do envio de mensagens.

• Mensagens: uma mensagem é mecanismo de comunicação entre os nós, composta por uma estrutura de dados. Pode ser composta por matrizes e mensagem de outras mensagens.

• Tópicos: o tópico é o nome dado para identificar uma mensagem, é um sistema de transporte de dados baseado em um sistema de publicar/subscrever. O nó envia e recebe mensagens em tópicos, um ou mais nós são aptos a publicar e subscrever dados em um tópico. Então, quando um nó realiza uma publicação, os outros nós que estiverem interessados podem ler os dados fornecidos na mensagem, assim os dados são trocados de forma assíncrona.

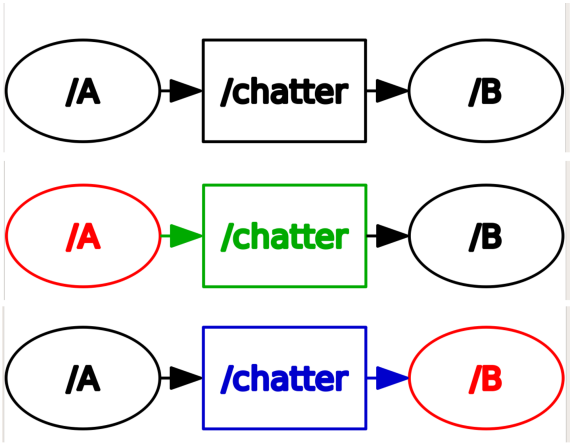
• Serviços: diferente dos tópicos, os serviços são mais restritos e seu transporte não é feito no modelo de muitos-para-muitos. Os serviços funcionam com o modelo de solicitação/ resposta, definidos por um par de estrutura de mensagem: uma para solicitação e outra para a

resposta.

• Bags: as bags são um formato para salvar e reproduzir dados de mensagens e ficheiros reproduzíveis, são importantes para armazenar dados de sensores, por exemplo.

Podemos observar na Figura 8 um exemplo simples de comunicação entre nós. Pode-se notar que temos dois nós, um chamado A e o outro B, e temos um tópico chamado *chatter*. Inicialmente o nó A publica uma mensagem no tópico *chatter* através de uma operação chamada *Publisher* do ROS, usada para publicar mensagens nos tópicos. O *chatter* agora contém essa mensagem que foi publicada pelo nó A. Para que o nó B consiga ter acesso a essa mensagem, ele terá que se inscrever no tópico *chatter*, para isso ser possível ele irá usar a operação chamada de *Subscriber* do ROS, usada para se inscrever nos tópicos e assim ter acesso a mensagens que são enviadas para ele. Como o nó B está inscrito no tópico *chatter*, tudo que o nó A publicar ele terá acesso. Neste exemplo, foi mostrado apenas um nó que publica A e um nó B que se inscreve no mesmo tópico *chatter*, porém podem ter diferentes nós X, Y e Z que publicam e diversos nós que se inscrevem A, B e C, todos em um mesmo tópico *chatter*.

Figura 8 – Conceitos Básicos do ROS



Fonte: Próprio autor

2.4.1.2 Ferramentas utilizadas no ROS

A ferramenta ROS é bastante conhecida e usada por programadores cujo objetivo é criar, simular e executar programas voltados para robôs. Diversas ferramentas e algoritmos podem ser utilizadas juntamente como o ROS para facilitar a vida de um programador, os mais utilizados de acordo com Carrera (2013) são:

• OpenCV: utilizado para o processamento de imagem,

• Pacotes Tf: utilizada para a manipulação de coordenadas,

• Rviz: sistema de visualização 3D,

• PointCloudLibrary: faz a reconstrução de um ambiente 3D utilizando medições a *laser*,

• Stage: simulador 2D, utilizado para testes

• Gazebo: simulador 3D, utilizado para testes.

O ROS vem se tornando cada vez mais importante para o desenvolvimento e evolução para área da robótica, aumentando significativamente seus usuários com o passar dos anos.

2.4.2 Simulador Player/Stage

Para conseguir realizar testes e validação do desenvolvimento da programação de sistemas robóticos, torna-se necessário o uso de simuladores. Os simuladores se tornam fundamentais para que não seja preciso ficar desmontando os robôs e criando diversos ambientes para testes sempre que ocorrer alguma modificação em um código ou quando houver qualquer outro tipo de mudança. Além disso, evita possíveis danos que possam ocorrer no robô. Tendo isso em vista, será apresentado a seguir um simulador bastante conhecido no mundo da robótica, o *Player/Stage* (CORREA, 2013).

Player é um *sotfware* livre para controle de robôs, foi projetado para ser independente de idioma e plataformas, fornece suporte para diversas conexões de clientes, uma interface de rede limpa e simples para os sensores e atuadores do robô. Começou a ser desenvolvido no ano 2000 por pesquisadores de *University of Southern Califórnia — EUA* com objetivo de suprir a demanda de um controlador/simulador de robôs móveis. O *Player* vem sendo desenvolvido junto a dois módulos de simulação: o *Stage* e o *Gazebo*.

O *Player* é baseado no modelo cliente/servidor, onde o servidor é o lado onde fica o programa em contato com o robô e que faz as leituras dos diferentes sensores e envia comandos aos atuadores do robô, o cliente é o outro programa que se conecta e se comunica com o servidor através de um protocolo de comunicação em rede. O cliente *Player* é uma biblioteca de funções que permite a comunicação entre o programa que controla o robô e o servidor. É utilizado para enviar instruções e receber informações do servidor como posição do robô, leitura de sensores, entre outros.

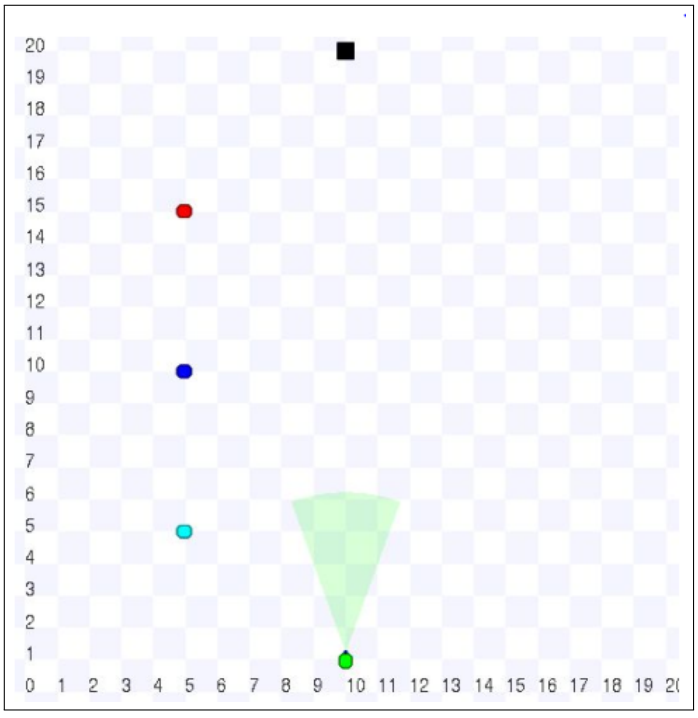
Stage é uma biblioteca de software C++ que simula robôs móveis, sensores e objetos em ambientes bidimensionais simulados, que trabalha em conjunto com o *Player*. Múltiplos

sensores e robôs podem ser simulados simultaneamente, e podem ser controlados por um ou mais clientes. Por ser projetado para ser capaz de simular dezenas de robôs operando simultaneamente em um mesmo computador. Tem como objetivo permitir o rápido desenvolvimento de controladores que eventualmente vão controlar robô reais, possibilitar experimentos robóticos sem acesso ao *hardware* real e sem o ambiente.

O *Stage* foi projetado para sistemas multi-agente, proporciona um ambiente muito simples computacionalmente, e modelos de vários dispositivos. Por ter compatibilidade com o ROS, torna possível que os códigos desenvolvidos no ROS que faz uso de tópicos para se comunicar, também consiga se comunicar com esse simulador através dos tópicos para acessar e enviar dados (COSTA, 2016).

Na Figura 9 mostra uma interface gráfica do simulador *Stage*, nesse ambiente simulado criado tem quatro robôs, onde um deles possui um sensor de distância e também pode ser visto um objeto preto que faz parte do ambiente simulado (COSTA, 2016).

Figura 9 – Ambiente do Stage



Fonte: Costa (2016)

3 TRABALHOS RELACIONADOS

Nesta seção, aborda-se uma revisão bibliográfica com os principais trabalhos relacionados. Em especial, os trabalhos que tratam de Sistemas de Vigilância e Cossimulação.

Em Silva e Almir (2015), os autores propõem um sistema robótico de monitoramento, chamado *Horus Patrol*, usando um robô móvel autônomo com acionamento diferencial para realizar a vigilância de um ambiente virtual. Este sistema realiza uma vigilância interna, patrulhando, para garantir que qualquer possível ameaça seja detectada pela cor de um objeto. A patrulha é feita através de rotas predefinidas, nas quais o robô é responsável por executá-las de maneira autônoma e relata qualquer descoberta anormal a um centro de comando, monitorado por um guarda responsável por filtrar falsos positivos e pela tomada de decisões em caso de perigo. Diferente do trabalho do autor Araújo (2019) que nos mostra o desenvolvimento de um robô vigilante baseado no ROS. Porém, tem como foco as opções de transmissão do vídeo capturado pela câmera em tempo-real como objetivo de serem analisadas por um usuário.

Apesar de os dois trabalhos citados acima utilizar o ROS que é a mesma ferramenta que será utilizado neste trabalho, existe uma diferença deles com o presente trabalho. Pois, nosso foco é voltado a comunicação entre o robô e a central de controle.

Para realizar a comunicação entre o robô e a central de controle, a cossimulação torna-se uma solução para esse problema. Em SILVA JUNIOR (2015), os autores propõem a implementação de um ambiente para verificação de sistemas embarcados heterogêneos através da cossimulação distribuída. A sincronização foi feita de maneira síncrona utilizando a HLA como *middeware*, ou seja, como uma camada oculta de tradução entre o software do sistema e o sistema embarcado, permitindo assim a comunicação e o gerenciamento de dados. Neste trabalho os autores tem como objetivo fornecer suporte para simulações e permitir a integração sincronizada de todos os dispositivos de hardware físico. Utilizaram o *Ptolemy* como ambiente de simulação. Obtiveram o resultado esperado, mostrando a integração bem sucedida entre o HLA e o *Ptolemy* e a verificação de sistemas utilizando *Harware-in-the-loop* e *Robot-in-the-loop*. Será utilizado neste trabalho a HLA que é um padrão de cossimulação, porém a diferença entre o trabalho dos autores SILVA JUNIOR (2015) e este trabalho é que será utilizado o Stage como plataforma de simulação. Além disso, a HLA será integrado com o ROS/Stage e a Central de Controle.

O autor Abreu (2019) propõe um ambiente de teste para analisar drones durante o voo em um ambiente fechado. Para isso o autor utiliza o *framework* Ptolemy II estendido para comunicação com drones reais usando o HLA. Foi realizado dois experimentos de detecção de

falha para testar o ambiente simulado, com um total de 20 voos realizados para cada experimento. Destes 20 voos, 16 foram utilizados para treinar o algoritmo de árvore de decisão, e os 4 restantes foram utilizados para testar o algoritmo em que uma das hélices possuía anomalia. O autor conseguiu uma taxa de acerto de 70 %. No trabalho descrito acima o autor utilizou HLA como padrão de cossimulação, que será o mesmo utilizado neste presente trabalho. Porém, em nosso trabalho será utilizado o Stage como ferramenta de simulação. Além disso, a HLA será integrada com o ROS/Stage e a Central de Controle.

Na Tabela 3 expõem-se os trabalhos relacionados que são comparados a este trabalho.

Tabela 3 – Comparação entre diferentes trabalhos com o trabalho proposto.

Trabalho	Sistemas de Vigilância	Robô autônomo	Cossimulação	Abrangência
Silva; Almir, 2015	X	X	-	Sistema de monitoramento e vigilância em ambientes internos.
Araújo, 2019	X	X	-	Robô Móvel para vigilância usando o ROS.
Júnior, 2015	-	-	X	Ambiente para verificação de sistemas heterogêneos através de cossimulação distribuída.
Abreu, 2019	-	-	X	Ambiente para testes e diagnósticos de drones usando cossimulação.
Trabalho Proposto	X	X	X	Ambiente para simulação e testes de vigilância de um robô através de cossimulação.

Fonte: Próprio autor

4 METODOLOGIA

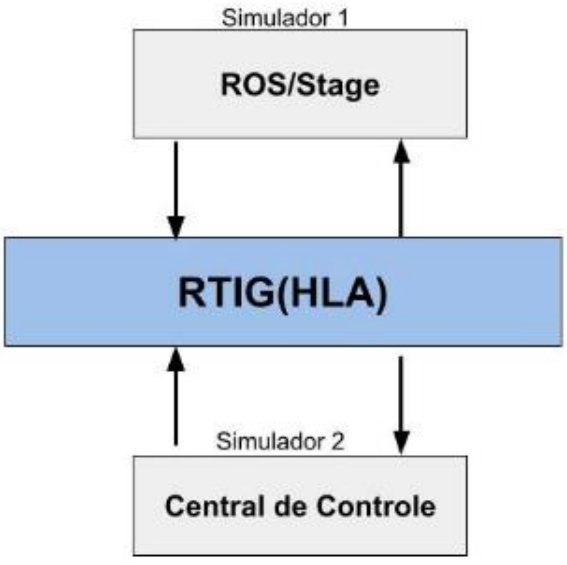
Esse capítulo abrange a descrição de todo ambiente criado para testes, que foi desenvolvido para o estudo de cossimulação com robô.

4.1 Arquitetura do Ambiente de Cossimulação

Para a criação de um ambiente cossimulação existem alguns fatores que devem ser levados em conta como, por exemplo, quais e quantos simuladores serão integrados nesse ambiente, como os simuladores irão se comunicar entre si, ou seja, o que será utilizado para que os diferentes simuladores consigam entender um ao outro e como será integrado um mecanismo de sincronização para que não ocorra perca de mensagens.

A *High Level Architecture* (HLA) foi utilizada para o desenvolvimento do nosso ambiente de cossimulação, pois fornece todos os recursos essenciais para viabilizar a comunicação e sincronização de troca de dados entre os diferentes simuladores. Cada simulador é conhecido como um federado, onde possuem uma interface de comunicação direta com a HLA, podendo assim enviar e receber mensagens de outros federados, o conjunto desses federados é chamada federação.

Figura 10 – Arquitetura Proposta para Cossimulação



Fonte: Próprio autor

O *Virtual Bus* foi utilizado nesse projeto por ser uma interface mais transparente

para a comunicação entre os federados e o RTI. Como foi desenvolvido com intuito de facilitar a comunicação, o *Virtual Bus* utiliza o método simples de leitura e escrita, ocultando assim para o usuário os métodos de publicação, subscrição e o procedimento de sincronização do HLA. Com isso, os elementos independentes podem ser introduzidos na federação e se comunicar de forma síncrona.

O RTI *(RunTime Infrastructure)* é o encarregado pela comunicação entre os federados e o responsável por garantir a entrega de mensagens na ordem certa, utilizando o tempo global do HLA, ele recebe mensagens dos federados e envia aos demais federados, controlando assim o compartilhamento de informações e a sincronização dos federados. Fazendo assim, com que eles estejam interligados e só avancem no tempo de simulação de forma conjunta, evitando que federados fiquem mais avançados ou atrasados em relação aos demais federados. Desse modo, o tempo de simulação é definido pelo federado mais lento, uma vez que os demais federados só podem avançar no tempo quando todos os outros tiverem terminado suas execuções.

O RTI utiliza de um arquivo *.fed* durante o tempo de execução, esse arquivo padroniza os pacotes de dados que serão trocados entre as federações. O formato desses dados é estipulado através da documentação do HLA para definição de um modelo de dados(OMT), e deverá ser comum a todos os federados na federação (IEEE1516.2, 2010).

Para que uma federação possa trocar informações com as outras ela deverá seguir o modelo definido pelo arquivo *.fed*. Um exemplo de arquivo OMT para o *Virtual Bus* pode ser visto no Código-fonte 4. Pode-se ver que esse arquivo *.fed* possui: um atributo para o id que especifica o federado, atributos para as variáveis de posição e orientação, e atributos para dados.

Código-fonte 4 – Trecho do Código de um Arquivo *.fed*

```
1 (FED
2     (Federation Test)( class Virtual
3         (attribute privilege)
4             (class RTIprivate)
5             (class chat
6                 (attribute id)
7               (attribute posicao_x)
8               (attribute posicao_y)
9               (attribute orientacao_w)
```

```
10                  (attribute data0)
11                  (attribute data1)
12                  (attribute data2)
13                  (attribute data3)
14                          . . .
15                  (attribute dataN)
16              )
17    )
18    (interactions)
19 )
```

A quantidade de atributos depende da quantidade de dados que se pretende trocar entre os federados e tem que ser especificado no arquivo *.fed*. Cada federado deve ter uma classe especificada nesse arquivo, para que o RTI consiga distinguir os federados existentes na federação, nesse exemplo temos uma classe chamada *chat* que especifica um federado, caso outro federado seja adicionado a federação tera que ser criado uma nova classe com o nome diferente, porém contendo os mesmos atributos. Com esses conhecimentos, pode-se montar a nossa cossimulação.

Neste trabalho é proposto o desenvolvimento de um ambiente de cossimulação, que comunica um sistema de vigilância robótico de um ambiente interno com a uma central de controle através de mensagens Figura 10 . O ambiente de cossimulação, está dividido entre o ROS/*Stage*, onde o robô será simulado e a Central de Controle, que receberá informações do robô e poderá enviar comandos para o mesmo. Para viabilizar esta cossimulação foram realizados experimentos para testar cada parte separadamente e depois o experimento final para testar o todo.

Os experimentos são detalhados no próximo capítulo estão divididos da seguinte maneira: simulação ROS/*Stage* 5.1, onde está sendo validado a comunicação entre o ROS e o simulador *Stage*, simulação da Central de Controle 5.2, onde está sendo validado a comunicação da Central de Controle e o *Virtual Bus* e por último a validação da cossimulação entre o ROS/*Stage* e a Central de Controle 5.3.

5 EXPERIMENTOS E RESULTADOS

Esse capítulo abrange os experimentos realizados após a criação do ambiente e seus devidos resultados.

5.1 Simulação ROS/*Stage*

O ROS/*Stage* é a parte encarregada por simular o ambiente e o robô, com suas devidas características. O robô é simulado dentro de um ambiente de simulação do *Stage* 2D, pode-se ver um trecho do código de configuração desse ambiente na Figura 11. Nesse arquivo é onde se encontra toda a configuração para a montagem do nosso ambiente de simulação.

Figura 11 – Trecho do Arquivo de Configuração do Ambiente de Simulação do Stage utilizado nos Experimentos

```
window
(
  size [ 635 666 ] # in pixels
  scale 22.971   # pixels per meter
  center [ -20.306  21.679 ]
  rotate [ 0.000  0.000 ]

  show_data 1              # 1=on 0=off
)

# load an environment bitmap
floorplan
(
  name "willow"
  bitmap "willow-full.pgm"
  size [54.000 58.700 0.500]
  pose [ -29.350 27.000 0.000 90.000 ]
)

# throw in a robot
erratic( pose [ -11.277 23.266 0.000 180.000 ] name "era" color "blue")
```

Fonte: Próprio autor

Nele temos o bloco *window*, que contém toda a configuração do ambiente onde o robô será incluindo, dentro desse bloco temos: o *size* que define o tamanho do ambiente, *scale* que é a escala, *center* que é a posição inicial e o *rotate* que é a rotação. Em seguida, temos o bloco *erratic* que adiciona um robô no ambiente, dentro desse bloco temos: *pose* que é a posição (x, y, z) onde o robô será inserido no ambiente, *name* que se refere ao nome dado a esse robô e o *color* que é a cor dada ao mesmo. Com isso, o ambiente e o robô estão configurados e prontos para ser usado Figura 12.

Após a configuração do ambiente e do robô, foi desenvolvido o código para realizar

Figura 12 – Ambiente de Simulação do Stage utilizado nos Experimentos

Fonte: Próprio autor

envio de comandos para o robô simulado, testando assim a comunicação entre o ROS e o *Stage*. Podemos observar na Código-fonte 5, um trecho do código desenvolvido em C++ para realizar a comunicação ROS/*Stage*.

Inicialmente foi incluído as bibliotecas que foram utilizadas do ROS. Na main() temos a função de inicialização ros::init(), função fundamental para que se rode um código que use o ROS. Em seguida temos o ros::NodeHandle [nome], que juntamente com o ros::init() são responsáveis por permitir que uma aplicação em C++ se torne um nó, podendo assim realizar comunicação com o sistema do ROS e conseguir se inscrever e publicar nos tópicos. A seguir temos ros::Publisher [nome], responsável por indicar a um tópico que deseja enviar informação, nesse código irei publicar mensagens do tipo *Twist* que enviaremos informações, no *Twist* conseguimos alterar a velocidade angular e linear do robô. A função ros::Subscriber [nome] é usada quando se deseja receber informações de um determinado tópico, nesse código ele se inscreveu no tópico chamado "/odometria", que retorna a posição atual do robô.

A próxima função ros::Rate, responsável por determinar quantas vezes o *loop* executará por segundo, nesse caso executará a uma taxa de 1Hz. A seguir, uma função while(ros::ok()) será executada enquanto o ROS estiver funcionando, dentro dela o rate.sleep() fará uma pausa

conforme o tempo determinado pela função ros::Rate. Com o código e o ambiente de simulação finalizados, foi realizado experimentos para testar a comunicação entre eles.

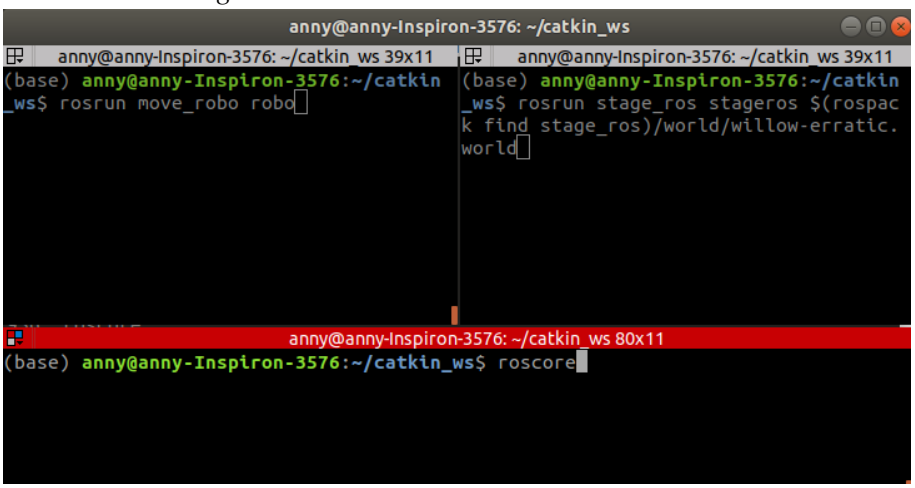
Código-fonte 5 – Trecho do Código para realizar a Comunicação do ROS e o Robô

```
#include <stdlib.h>
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "sensor_msgs/LaserScan.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"

int main(int argc, char** argv){
  ros::init(argc,argv, "Robo");
  ros::NodeHandle node1;
  ros::NodeHandle node2;
  ros::Publisher pub = node1.advertise<geometry_msgs::Twist
     >("/cmd_vel", 1000);
  ros::Subscriber sub = node2.subscribe("/odometria", 1000,
      recebeMsg);
  ros::Rate rate(1);

    while (ros::ok())
    {
          CODE HERE

    rate.sleep();
    }
  return 0;
  }
}
```

5.1.1 *Experimentos de Comunicação entre o ROS e o Stage*

Para realizar essa simulação três terminais foram executados simultaneamente, pode-se ver um exemplo de execução dessa simulação na Figura 13. O primeiro com o comando roscore que é o comando essencial para que os nós do ROS consigam se comunicar entre si, pois é ele que inicia o *Master* e deve ser o primeiro comando a ser executado. No segundo terminal tem que executar o ambiente com o robô criado no *Stage* e no terceiro tem que executar o código criado para mover o robô.

Figura 13 – Exemplo de uma execução de simulação enter ROS e *Stage*



Fonte: Próprio autor

5.1.1.1 *Experimento 1: Envio de mensagem do ROS para o robô (Stage)*

O Experimento 1 consistem em realizar a comunicação ROS e o robô que esta sendo simulado no *Stage*. Para a realização do primeiro experimento, foi implementado um código para mover o robô fazendo com que o mesmo realize uma rota simples. Essa rota consiste no robô se locomover em círculo no sentido anti-horário. O trecho do código responsável pela configuração da rota do robô é mostrado no trecho do Código-fonte 6.

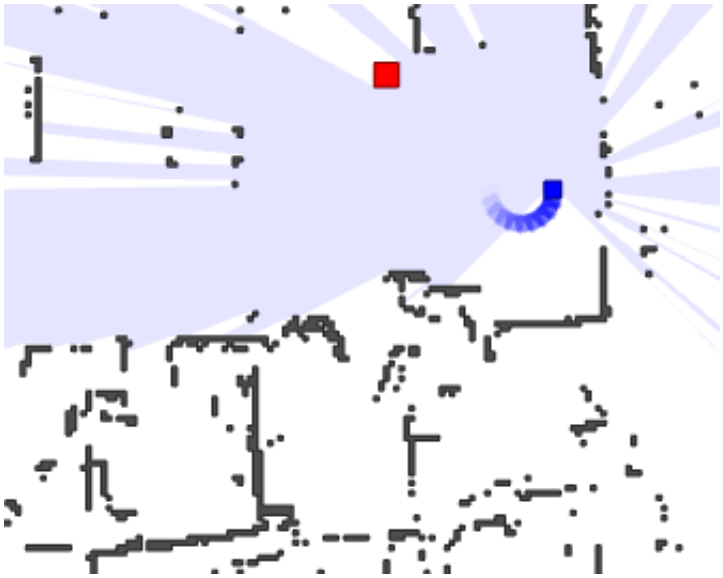
Código-fonte 6 – Trecho do Código da Rota 1

```
1 ros::Publisher pub = node_robo.advertise<geometry_msgs::
    Twist>("/cmd_vel", 1000);
2 geometry_msgs::Twist vel;
3 while(ros::ok()){
```

```
4     vel.linear.x = 2;
5     vel.angular.z = 0;
6     pub.publish(vel);
7     ros::spinOnce();
8     sleep(1);
9     vel.linear.x = 0;
10    vel.angular.z = M_PI/2;
11    pub.publish(vel);
12    ros::spinOnce();
13    sleep(1);
14 }
```

Como explicado anteriormente, o código para rota do robô é implementada dentro do while(ros::ok). Pode-se notar que inicialmente está sendo criado um Publisher responsável por indicar a um tópico que será enviado informação do tipo *Twist*, é através dele que será alterado a velocidade angular e linear do robô. Logo abaixo, temos um construtor do tipo Twist, através dele que será atribuído valores para mudar a velocidade do robô.

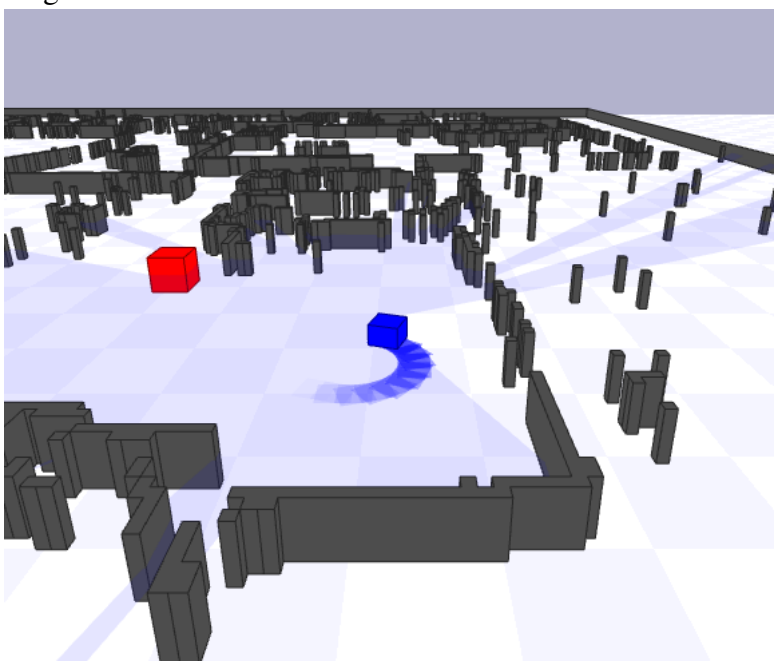
Figura 14 – Robô realizando a Rota 1 - Visão Bidimensional



Fonte: Próprio autor

Dentro do while(ros::ok) está sendo passado a velocidade linear e angular respectivamente, nesse momento está sendo modificado a velocidade linear no eixo x do robô para 2 e a angular no eixo z para 0. Em seguida, esse valor está sendo publicado no robô, após temos um

Figura 15 – Robô realizando a Rota 1 - Visão Tridimensional



Fonte: Próprio autor

ros::spinOnce(), que permiti o processamento de mensagens enviadas, temos uma pausa de 1s, e o processo se repete só modificando os valores da velocidade linear e angular, para que o robô realize a rota desejada. Como resultado, é esperado que o robô realize a rota implementada.

Pode-se constatar o resultado do Experimento 1 nas Figuras 14 e 15. A primeira imagem é a visão unidimensional do ambiente e a segunda é a imagem bidimensional. O experimento 1 obteve o resultado esperado, pois como pode-se ver, o robô realizou a rota com sucesso validando assim a comunicação de envio de instruções entre ROS/*Stage*.

5.1.1.2 Experimento 2: Envio de mensagens do robô(Stage) para o ROS

O Experimento 2 consiste em testar o recebimento de informações dos sensores do robô no ROS. Para realizar o segundo experimento foi implementado um código mais elaborado, onde o robô ficar percorrendo o ambiente evitando se chocar contra a parede ou algum obstáculo e enviará sua posição para o ROS. O robô envia ao ROS os dados do sensor Odometria, que tem sua posição e orientação atual. O trecho do código responsável pelo recebimento desses dados do sensor é mostrado no trecho do Código-fonte 7.

Código-fonte 7 – Trecho do Código para receber informações do robô

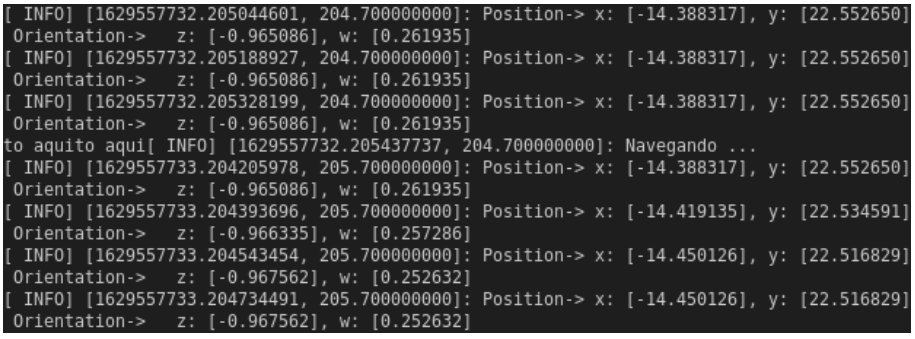
```
1 void recebeOdom(const nav_msgs::Odometry::ConstPtr& msg){
2    ROS_INFO("Position-> x: [%f], y: [%f] \n Orientation->
```

```
        z: [%f], w: [%f]",
3   msg->pose.pose.position.x,msg->pose.pose.position.y,msg
      ->pose.pose.orientation.z, msg->pose.pose.orientation
      .w);
4 }
5 int main(int argc, char** argv){
6       CODE
7    ros::Subscriber sub = node_robo.subscribe("/odom", 100,
       recebeOdom);
8         CODE
9 }
```

Para se tornar possível receber valores de um sensor do robô inicialmente terá que se inscrever em um tópico e criar uma função para receber esse tipo de mensagem. Na main() foi criado um Subcriber para poder receber valores do sensor de odometria, nos parâmetros temos: o nome do tópico, a quantidade de mensagens e a chamada para função para receber esses dados.

Foi criada uma função do tipo void(), pois não precisa retornar nada, e foi especificado o tipo de mensagem que ela irá receber, foi especificado como uma mensagem do tipo Odometria. Para conseguir visualizar esses dados, para verificar se realmente estavam chegando, foi colocado o ROS_INFO que irá imprimir no terminal a posição x, y e a orientação w.

Figura 16 – Dados do sensor de Odometria do robô recebidos pelo ROS



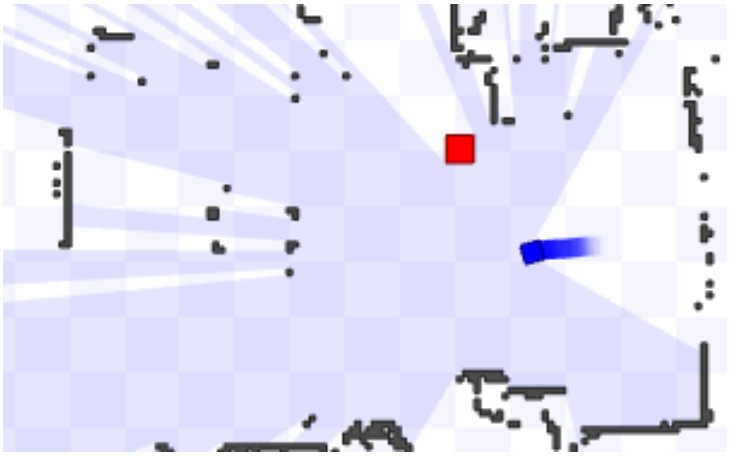
Fonte: Próprio autor

Como explicado anteriormente, para verificar se a função realmente estava recebendo valores do sensor de Odometria foi usado o ROS_INFO para receber e mostrar no terminal o dado recebido do sensor do robô. Pode-se constatar o resultado do Experimento 2 na Figura 16.

Nela podemos visualizar os valores recebidos do sensor, que o robô enviava enquanto estava percorrendo pelo ambiente, enviando os valores de sua posição e orientação. Podemos observar também nas Figuras 17 e 18 temos o robô realizando a rota 2.

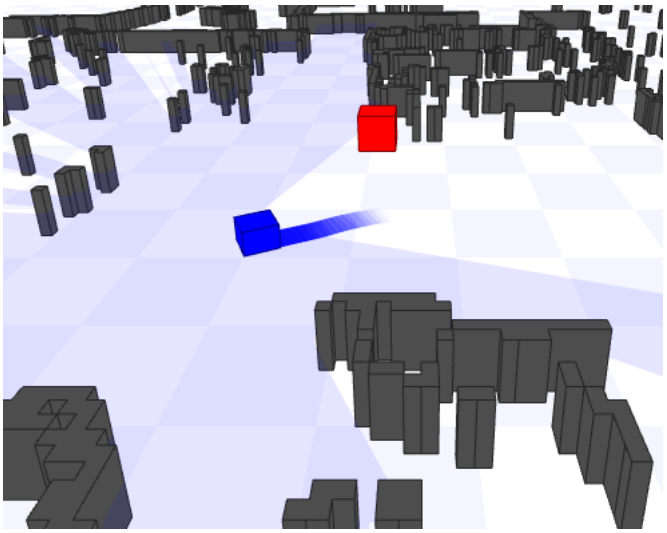
Como resultado, o experimento 2 obteve o resultado esperado, pois como pode-se ver, o robô estava enviando sua localização de acordo com sua trajetória. Validando assim a recepção de informações dos sensores do robô no código desenvolvido com o ROS.

Figura 17 – Robô realizando a Rota 2- Visão Bidimensional



Fonte: Próprio autor

Figura 18 – Robô realizando a Rota 2- Visão Tridimensional



Fonte: Próprio autor

5.2 Simulação Central de Controle

A Central de Controle é a parte responsável por receber informações do robô enquanto ele está trafegando pelo ambiente de simulação. Também responsável por enviar coor-

denadas caso seja necessário que o robô mude de posição no ambiente. Ela receberá do robô variáveis x, y e w. Onde as variáveis x e y são referentes a posição do robô e a variável w é referente a orientação do robô. Nesta seção será realiza o teste de comunicação entre a Central de Controle e a HLA.

5.2.1 Experimento de comunicação entre a Central de Controle e a HLA

Para realizar a verificação se a Central de Controle está conseguindo enviar e receber informações, foi realizado um teste de comunicação com o *Virtual Bus*/HLA. Inicialmente, para que a Central de Controle se torne um federado, terá que implementar o pacote do *Virtual Bus* e ter o modelo de compartilhamento de dados, o arquivo *.fed* Código-fonte 4. Com isso, se torna um federado e assim envia e recebe informações através dos serviços do RTI. Lembrando que, o arquivo *.fed* tem que ser utilizado por todos os federados da federação.

A cossimulação deve ocorrer entre a central e ROS/*Stage*, para isso o RTIG tem que está sendo executado em um terminal, conforme ilustrado na Figura 19. Com o RTIG sendo executado, os federados podem iniciar a comunicação e serem inseridos na federação. Para isso, a federação é criada para a central e o ROS/*Stage*, conforme ilustrado na Figura 20. Por último, são enviadas mensagens de teste, verificando a integridade das mensagens, caso a comunicação fosse realizada com sucesso será mostrado no terminal a palavra PASS, como pode ser verificado na Figura 21 a comunicação entre eles foi realizada com sucesso.

Figura 19 – CERTI RTIG

```
CERTI RTIG 4.0.0 - Copyright 2002-2018  ONERA & ISAE-SUPAERO
This is free software ; see the source for copying conditions. There is NO
warranty ; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

CERTI RTIG up and running ...
```

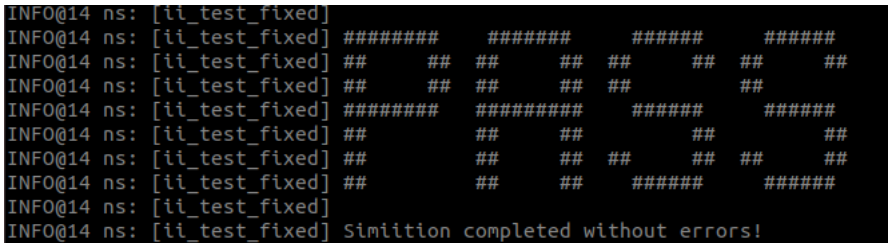
Fonte: Próprio Autor

Figura 20 – Criação de uma Federação

```
Created Federation
Joined Federation as 1
initializeHandles
Synchronization point announced: ReadyToRun
Successfully registered sync point: ReadyToRun
registerFederationSynchronizationPoint
 >>>>>>>>>> Press Enter to Continue <<<<<<<<<<
```

Fonte: Próprio Autor

Figura 21 – Verificação da Comunicação entre o RTI e a Central de Controle

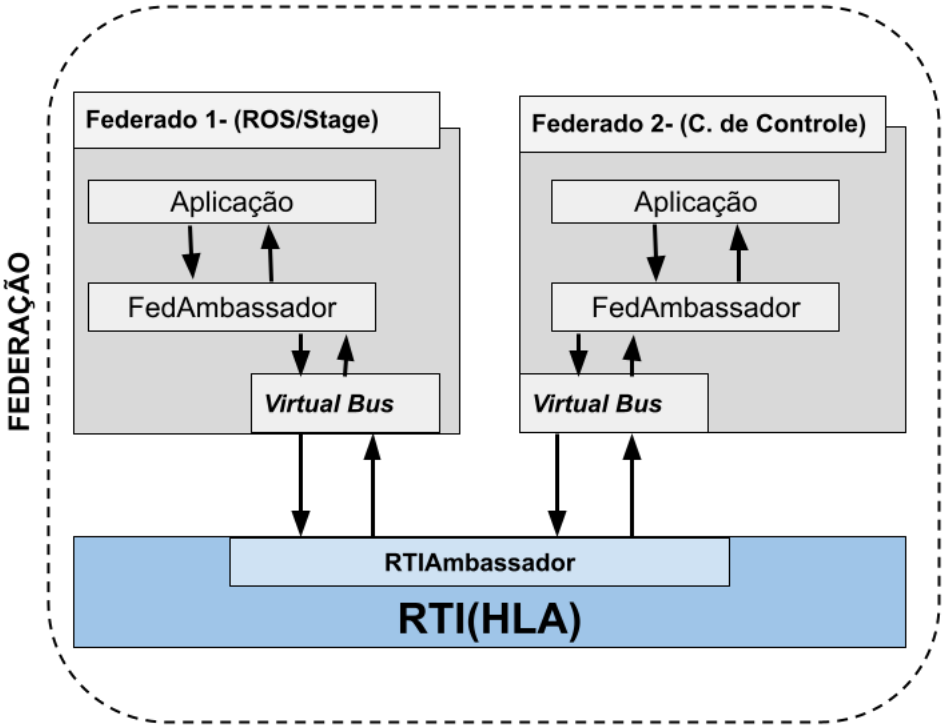


Fonte: Próprio Autor

5.3 Cossimulação entre Central de Controle e ROS/*Stage*

Com o ROS/*Stage* se comunicando entre si e a Central de Controle já sendo um federado, pois se comunica com RTI, será realizado o último experimento, fazer com que eles realizem a cossimulação. Para que isso se torne possível o ROS/*Stage* terá que se tornar um federado.

Figura 22 – Arquitetura Desenvolvida

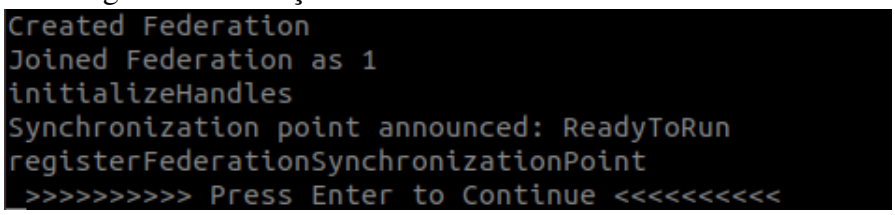


Fonte: Próprio Autor

Para isso ele terá que ter o modelo de compartilhamento de dados, o *.fed* e usará a interface para a comunicação com o RTI, o *Virtual Bus*. Com isso, poderá se tornar um federado e assim conseguirá se comunicar com o RTI enviar os dados do robô e receber os comandos da Central de Controle.

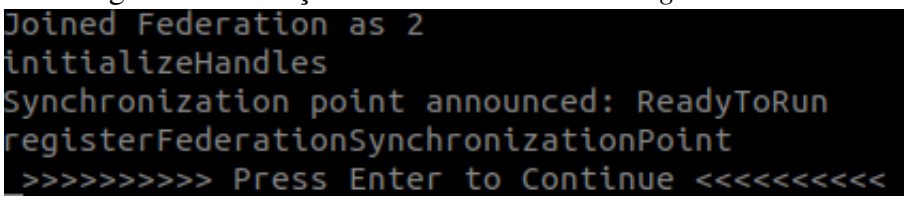
Pode-se observar a arquitetura final desenvolvida neste trabalho na Figura 22. Nela temos uma federação formada por dois federados, federado 1 que é o ROS/*Stage* e o federado 2 que é a Central de Controle. Os dois federados contém tem a API do *Virtual Bus* que instancia o FedAmassador onde ele se comunica com a aplicação. O *Virtual Bus* que se comunica diretamente com RTIAmbassador do HLA, podendo receber e enviar dados para os federados através do RTI.

Figura 23 – Criação da federado 1 - Central de Controle



Fonte: Próprio Autor

Figura 24 – Criação da federado 2 - ROS/*Stage*



Fonte: Próprio Autor

Para testar se os dois federados estão se comunicando entre si, foi realizada a configuração para troca de dados. São enviadas mensagens do tipo *route*, onde se tem *n* mensagens, correspondente ao *id*, para identificar a federação, *msg* para enviar alguma mensagem de erro caso seja necessário, *x* e *y*, referentes a posição do robô, *w* correspondente a orientação do robô no ambiente. As *n* mensagens são enviadas em *n* variáveis, será uma variável em cada mensagem.

Será realizado um experimento onde vão enviar mensagens um para o outro, para esse experimento serão trocadas mensagens com valores 0, só para podermos testar se eles estão conseguindo se comunicar entre si. Como explicado anteriormente, o RTI é o primeiro a ser executado, pois, as federações precisam dele para realizar a troca de mensagem.

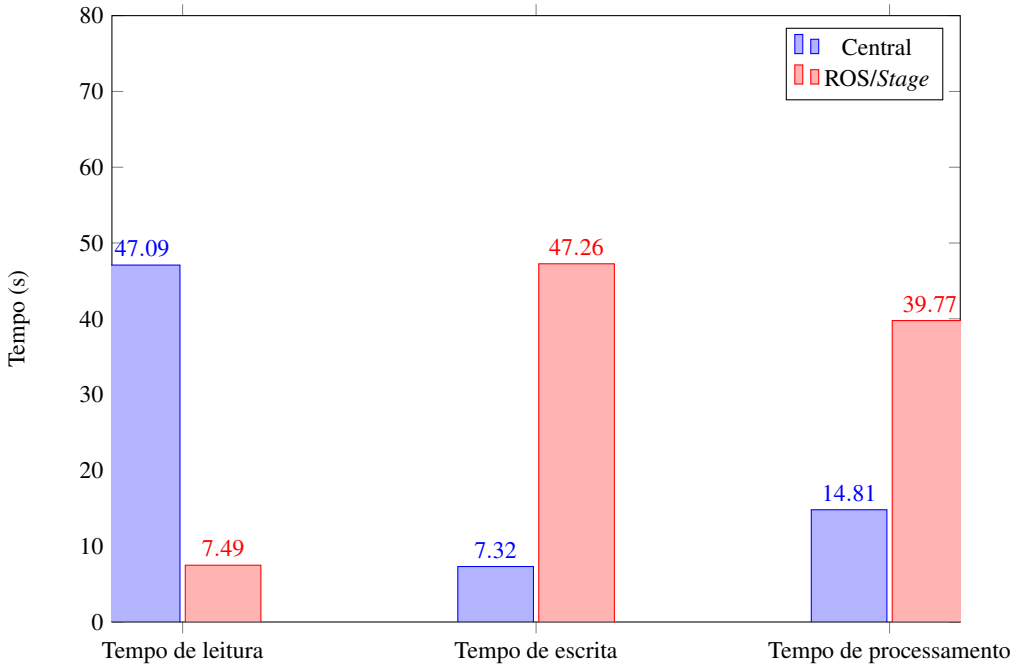
Com o RTI funcionado será executado a Central de Controle e o ROS/*Stage* para se tornarem federado, pode-se observar nas Figuras 23 e 24 os dois federados sendo criados, ambos sendo executados em terminais distintos. Com o RTI executando e as federações criadas foi testado o envio de mensagens entre eles.

5.4 Resultados

Nos experimentos, as mensagens trocadas pelo *Virtual Bus*/HLA são consideradas com dois federados, executando a simulação de 10.000 interações. Em ambas as abordagens é calculado a média do total de interações. Durante a troca de mensagens são considerados 3 valores por atributos (HLA), um atributo para cada coordenada. Os resultados podem ser mostrado na Figura 25, em que foram trocadas mensagens pelo *Virtual Bus*/HLA entre a central e o ROS/*Stage*. Os federados realizam a troca de mensagens e validam o ambiente de cossimulação. Nesse experimento é importante para avaliar o impacto de diferentes abordagens com *Virtual Bus*/HLA mesmo com o aumento do processamento gerado pelo federado ROS/*Stage*.

O tempo de escrita dos dados enviados da central para o ROS/*Stage* ocorre com $7,32$ segundos, enquanto o retorno das mensagens demora o tempo de processamento do ROS/*Stage* de $39,77$ com o tempo de escrita da central, ocorrendo com $47,26$. O tempo de leitura do ROS/*Stage* acontece em $7,49$, enquanto o da central ocorre depois do processamento do ROS/*Stage* em $47,09$, sendo a leitura realizada pela central no final do processamento do robô.

Figura 25 – Análise do tempo de leitura, escrita e processamento



Fonte: Próprio autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho realizou a cossimulação entre o simulador ROS/*Stage* e uma Central de Controle, sendo a central responsável por enviar os dados para o percurso do robô. Para isso ser possível foi usada a especificação HLA, considerando adaptações para uma interface de comunicação comum e transparente para o usuário conhecida como *Virtual Bus*, que possibilitou que a cossimulação ocorra sincronizada e administrada na troca de mensagens.

Para desenvolver o ambiente de simulação foi mostrado no Capítulo 2 toda a fundamentação necessária para implementar o ambiente de simulação usando o simulador *Stage* 2D. Para viabilizar a comunicação entre o ROS/*Stage* foi desenvolvido um código em C++ para que o robô que estava sendo simulado no *Stage* conseguisse receber e enviar informações para o ROS. A comunicação entre a Central de Controle e a interface *Virtual Bus*, mostrando como foi realizada essa comunicação. Por último, foi viabilizado a cossimulação entre ROS/*Stage* e a Central de Controle 5.3, utilizando a *Virtual Bus*, validando assim todos os objetivos traçados.

Como resultados temos que, os experimentos realizados obtiveram ótimos resultados. O experimento 5.1 que era a comunicação entre ROS/*Stage* foi validado com sucesso. O experimento 5.2, que era a simulação da Central de Controle com *Virtual Bus* também foi validado. Por último, o ambiente finalizado 5.3, que realizou a cossimulação entre os dois federados.

Como trabalhos futuros é possível a implementação de mais robôs no ambiente de simulação, também implementar um código de rota que faça com que o robô possa percorrer todo o mapa e adicionar mais sensores no robô para que o mesmo fazer a detecção de objetos no ambiente.

REFERÊNCIAS

ABREU, R. R. d. **Um ambiente para teste e diagnóstico de drones usando cossimulação**. Dissertação (Mestrado) – Universidade Federal da Paraíba, João Pessoa, 2019.

AMORY, A.; MORAES, F.; OLIVEIRA, L.; CALAZANS, N.; HESSEL, F. A heterogeneous and distributed co-simulation environment [hardware/software]. In: IEEE. **Proceedings. 15th Symposium on Integrated Circuits and Systems Design**. [*S. l.*], 2002. p. 115–120.

ARAÚJO, A. M. A. A. de. **Robô Móvel para Vigilância**. Dissertação (Mestrado) – Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 2019.

BARROS, J. S. **Uma ferramenta para avaliar estratégias de voos de VANTs usando cossimulação**. 77 f. Dissertação (Mestrado em Informática) – Universidade Federal da Paraíba, João Pessoa, 2017.

BISHOP, W. D.; LOUCKS, W. M. A heterogeneous environment for hardware/software cosimulation. In: IEEE. **Proceedings of 1997 SCS Simulation Multiconference**. [*S. l.*], 1997. p. 14–22.

CARRERA, T. F. **Movimentos reativos no robô turtlebot utilizando o kinect**. Dissertação (Mestrado em Engenharia Industrial) – Universidade de León, León, Espanha, 2013.

CORREA, D. S. O. **Navegação autônoma de robôs móveis e detecção de intrusos em ambientes internos utilizando sensores 2D e 3D**. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemática e Computação, Universidade de São Paulo, São Carlos, 2013.

COSTA, L. F. S. **Um ambiente de monitoramento para sistemas multi-robôs com cossimulação federada e Hardware-in-the-Loop**. 82 f. Dissertação (Mestrado em Informática) – Universidade Federal da Paraíba, João Pessoa, 2016.

CRIIS. **RobVigil**. 2011. Disponível em: http://criis.inesctec.pt/index.php/criis-projects/robvigil/. Acesso em: 20 ago. 2020.

DOTA, M. A. **Avaliação de desempenho de uma ferramenta para simulação distribuída baseada no protocolo otimista time warp**. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2001.

FUJIMOTO, R. M. Parallel and distributed simulation systems. In: IEEE. **Proceeding of the 2001 Winter Simulation Conference (Cat. No. 01CH37304)**. [*S. l.*], 2001. v. 1, p. 147–157.

IEEE1516. Ieee standard for modeling and simulation (m s) high level architecture (hla) - framework and rules. **IEEE Std 1516-2000**, p. 1–28, 2000.

IEEE1516.1. Ieee standard for modeling and simulation (m s) high level architecture (hla) federate interface specification. **IEEE Std 1516.1-2000**, p. 1–480, 2000.

IEEE1516.2. Ieee standard for modeling and simulation (m amp;s) high level architecture (hla)– object model template (omt) specification. **IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)**, p. 1–110, 2010.

KOUBÂA, A. *et al.* **Robot Operating System (ROS).** [*S. l.*]: Springer, 2017. v. 1.

Lasnier, G.; Cardoso, J.; Siron, P.; Pagetti, C.; Derler, P. Distributed simulation of heterogeneous and real-time systems. In: **2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications**. [*S. l.*: *s. n.*], 2013. p. 55–62.

LEVEL, R. C. G. **ROS Computation Graph Level**. 2013. Disponível em: https://wiki.ros.org/ROS/Concepts. Acesso em: 21 jun. 2014.

NICOLESCU, G.; BOUCHENEB, H.; GHEORGHE, L.; BOUCHHIMA, F. Methodology for efficient design of continuous/discrete-events co-simulation tools. **High Level Simulation Languages and Applications-HLSLA**, SCS, San Diego, CA, p. 172–179, 2007.

NOULARD, E.; ROUSSELOT, J.-Y.; SIRON, P. **CERTI, an Open Source RTI, why and how**. 01 2009.

OLIVEIRA, T. J. S. **Um Ambiente para Simulação e Testes de Comunicação entre Multi-Robôs através de Cossimulação**. Dissertação (Mestrado em Informática) – Universidade Federal da Paraíba, João Pessoa, 2016.

SCHLAGER, M. **Hardware-in-the-Loop Simulation:** a scalable. time-triggered hardware-in-the-loop simulation framework. VDM Verlag, Component-based, 2008.

SILVA JUNIOR, J. C. V. **Verificação de Projetos de Sistemas Embarcados através de Cossimulação Hardware/Software**. 68 f. Dissertação (Mestrado em Informática) – Universidade Federal da Paraíba, João Pessoa, 2015.

Silva, R. G. N.; Almir, K. Indoor surveillance robotic system using an autonomous mobile robot. In: **2015 CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)**. [*S. l.*: *s. n.*], 2015. p. 829–833.

SILVA, T. W.; MORAIS, D. C.; ANDRADE, H. G.; LIMA, A. M.; MELCHER, E. U.; BRITO, A. V. Environment for integration of distributed heterogeneous computing systems. **Journal of Internet Services and Applications**, SpringerOpen, v. 9, n. 1, p. 1–17, 2018.

THONDUGULAM, N. V.; RAO, D. M.; RADHAKRISHNAN, R.; WILSEY, P. A. Relaxing causal constraints in pdes. In: IEEE. **Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999**. [*S. l.*], 1999. p. 696–700.