UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

UM AMBIENTE DE MONITORAMENTO PARA SISTEMAS MULTI-ROBÔS COM COSSIMULAÇÃO FEDERADA E HARDWARE-IN-THE-LOOP

LUÍS FELIPHE SILVA COSTA

JOÃO PESSOA-PB
Janeiro - 2016

Universidade Federal da Paraíba
Centro de Informática
Programa de Pós-Graduação em Informática

Um ambiente de monitoramento para sistemas multi-robôs com cossimulação federada e Hardware-in-the-Loop

Luís Feliphe Silva Costa

Proposta de Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Sinais, Sistemas Digitais e Gráficos

Alisson Vasconcelos de Brito (Orientador)

João Pessoa, Paraíba, Brasil

C837u Costa, Luís Feliphe Silva.
Um ambiente de monitoramento para sistemas multi-robôs com cossimulação federada e Hardware-in-the-loop / Luís Feliphe Silva Costa.- João Pessoa, 2016.
82f. : il.
Orientador: Alisson Vasconcelos de Brito
Dissertação (Mestrado) - UFPB/CI
1. Informática. 2. Ciência da computação. 3. Sistemas digitais e gráficos. 4. Robot-in-the-loop. 5. Arquitetura de alto nível. 6. Cossimulação.

UFPB/BC CDU: 004(043)

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de **LUIS FELIPHE SILVA COSTA**, candidato ao título de Mestre em Informática na Área de Sistemas de Computação, realizada em 15 de janeiro de 2016.

Ao décimo quinto dia do mês de janeiro do ano de dois mil e desesseis, às dez horas, no Centro de Informática - Universidade Federal da Paraíba (unidade Mangabeira) reuniram-se os membros da Banca Examinadora constituída para examinar o candidato **Sr. Luis Feliphe Silva Costa**, vinculado a esta universidade pela matrícula 2014108734, ao grau de Mestre em Informática, na área de "*Sistemas de Computação*", na linha de pesquisa "*Sinais, Sistemas Digitais e Gráficos*". A comissão examinadora foi composta pelos professores doutores: **Alisson Vasconcelos de Brito (PPGI – UFPB),** Orientador e Presidente da Banca, **Tiago Pereira do Nascimento (PPGI-UFPB)**, Examinador Interno e **Luiz Marcos Garcia Gonçalves (UFRN)**, Examinador Externo à Instituição. Dando início aos trabalhos, o presidente da banca cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a palavra ao candidato para que o mesmo fizesse, oralmente, a exposição do trabalho de dissertação intitulado "*Um ambiente de monitoramento para sistemas multi-robôs com cossimulação federada e hardware-in-the-loop*". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que emitiu o seguinte parecer: "*aprovado*". Assim sendo, eu, Clauirton de Albuquerque Siebra, Coordenador do PPGI – Programa de Pós Graduação em Informática, lavrei a presente ata que vai assinada por mim e pelos membros da Banca Examinadora. João Pessoa, 15 de janeiro de 2016.

Clauirton de Albuquerque Siebra

Profº Alisson Vasconcelos de Brito
Orientador (PPGI-UFPB)

Profº Tiago Pereira do Nascimento
Examinador Interno (PPGI – UFPB)

Profº Luiz Marcos Garcia Gonçalves
Examinador Externo à Instituição (UFCG)

1

325.114.571-20

Resumo

Simulações frequentemente são utilizadas no desenvolvimento de sistemas para prever erros antes de sua implantação no ambiente final. Há uma diversidade entre os tipos e técnicas de simulações, entre elas a de *hardware-in-the-loop*. Nela dispositivos de *hardware* são adicionados a simulação para aumentar a realidade dos resultados. Neste trabalho utilizamos esta técnica em conjunto com a cossimulação dos simuladores Stage e Ptolemy para prover um ambiente de simulação multi-robôs e de sistemas embarcados. Este ambiente pode ajudar na detecção de falhas de *hardware* e de *software*. O ambiente é integrado por meio do padrão IEEE 1516, a Arquitetura de Alto Nível, que gerencia o tempo de simulação e dados compartilhados durante a simulação. No trabalho são realizadas simulações para estudo do ambiente de sincronização, visto que erros neste sentido podem comprometer os resultados das simulações. Há ainda a utilização de robôs reais para validar o ambiente final desenvolvido.

Palavras-chave: Robot-in-the-loop, Arquitetura de Alto Nível, cossimulação.

Abstract

Simulations often are used in development systems do predict how systems will work on final environment. Many kinds of simulations and techniques are available, one of then is Hardware-in-the-Loop Simulation that provides a more realistic environment by using real devices on simulations. In this work this technique is used with co-simulation between Ptolemy and Stage to provide an multi-robot environment and embedded systems design. It can help to find hardware and software errors during development. High Level Architecture, an IEEE pattern to interoperability between simulators is used to manage time and data that is shared on co-simulation. This work also studies synchronization aspects and uses real robots on simulations to validate proposed environment.

Keywords: Hardware-in-the-Loop, High Level Architecture , co-simulation.

Agradecimentos

Agradeço a Deus, por permitir que todo este trabalho pudesse ser realizado. Em seguida ao professor que me orientou, Alisson Brito. Ele apoiou-me no desenvolvimento deste trabalho desde o início das atividades, se fazendo sempre presente e atencioso em tudo. Agradeço ainda ao professor Tiago Nascimento que forneceu os robôs para realização dos experimentos deste trabalho e também acompanhou-me desde o início do mestrado, sendo professor da disciplina de robótica móvel. Ao meu orientador no período de mobilidade na Faculdade de Engenharia da Universidade do Porto, o professor Luís Almeida que viabilizou a experiência do intercâmbio. Pelas contribuições da banca avaliadora, composta por: Luís Marcos, Tiago Nascimento e Alisson Brito. Agradeço ainda a todos professores que contribuíram de forma direta ou indireta com este trabalho. Agradeço a minha família que sempre apoiou meus estudos: Erasmo, Vanderléia e Eduardo. Aos meus primos: Anna, Segundo, Jeremias. Aos meus tios e tias: Rosinha, Terezinha, Rosita, Rosilda, Lenilda, Bismarck, Keyla, Reijane, Vanderli, Magno e Lúcia. Ao meu avô Manuel e avós Rita e Laurita (*in memoriam*). A família do ejc nas pessoas de João e Luíza. Aos meus amigos de Faculdade que acompanharam desde cedo meu trabalho: Lettiery, Amanda Vilar, Eline, Diego, Cláudio, Charles S., Renan Soares, Ihago A., Arquijoaquitônio, Mariana M., Jessyca F., Diorgenes M., Rafael D., Barbara F., Cinthya P. e Aline M. Aos amigos que acompanham-me desde o ensino médio: Philippe Luiz (*in memoriam*), Rodrigo A., Sloany G., Amanda S., Alessandra S., Thuany G., Bruno G., Nalyje L. e Bruno F.. Aos Amigos do período de mobilidade no Porto: Sofia, Pedro Zão, Pedro P., Nicola, Daniel C., Zahid, Luís Oliveira, Ana Rita, Ricardo, Ágata, Pedro L., Cláudia, Daniel M., Bruno, Bia Pereira e Inês.

Conteúdo

Lista de Símbolos

HLA : *Arquitetura de Alto Nível - High-Level Architecture*

HIL : *Hardware-in-the-Loop*

RiL : *Robot-in-the-Loop*

RTI : *Infraestrutura de tempo de Execução - Runtime Infraestructure*

SOM : *Modelo de Objeto Simulado - Simulated Object Model*

FOM : *Modelo de Objeto Federado - Federate Object Model*

MOM : *Modelo de Gestão de Objetos - Model Object Management*

FSM : *Máquina de Estados Finita - Finite State Machine*

MoC : *Modelo de Computação - Model of Computation*

BBB : *Beaglebone Black*

ROS : *Sistema Operacional Robótico - Robotic Operational System*

Lista de Figuras

Capítulo 1

Introdução

1.1 Apresentação

Um dos motivos da utilização de simulações em projetos é reduzir gastos e prever se o sistema vai se comportar como esperado em seu ambiente final. Uma técnica que vem sendo utilizada em sistemas embarcados de tempo real antes da implantação em ambiente final é a de *Hardware-in-the-loop* (HiL) (SCHLAGER, 2008). Neste tipo de simulação, componentes de *hardware* são adicionados a simulação permitindo que seja possível obter resultados mais realistas. Essa técnica também é utilizada quando o componente que se deseja simular possui alta complexidade não sendo de fácil virtualização. A partir da HiL, uma nova técnica que também envolve dispositivos de hardware chamada *Robot-in-the-Loop* (RiL) com foco em robôs é utilizada neste trabalho.

Um exemplo da utilidade das simulações RiL é quando não se tem o ambiente em que se deseja fazer o experimento (HU; ZEIGLER, 2015). Este ambiente pode ser um labirinto ou um ambiente inóspito como a superfície de um planeta ou o fundo do oceano. Dessa forma um robô real seria utilizado na simulação *Robot-in-the-Loop* com o ambiente simulado. Outra utilidade pode ser quando se deseja trabalhar com uma equipe de robôs e não há robôs suficientes para a atividade, por meio de simulação pode-se utilizar os robôs reais e robôs virtuais em um mesmo ambiente de simulação viabilizando o experimento desejado.

Assim, este trabalho propõe a criação e o estudo de aspectos importantes em um ambiente de cossimulação de robôs e utilização de HiL. Entre estes aspectos está o gerenciamento do tempo de simulação e o gerenciamento dos dados a serem compartilhados. Na tentativa de

realizar um gerenciamento adequado, a integração proposta em todo ambiente (simuladores e robôs) é realizada pela Arquitetura de Alto Nível (*High Level Architecture* - HLA). A HLA é um padrão IEEE para interoperabilidade entre simuladores heterogêneos de forma transparente, síncrona e consistente (IEEE. . . , 2010b). Nesta arquitetura cada simulador representa um federado, que possui regras próprias e uma interface para com os outros, tornando desnecessário aos demais o conhecimento do funcionamento interno de cada simulador. Essa arquitetura especifica como deve ocorrer a comunicação entre simuladores de forma padronizada. Dessa forma, qualquer simulador que possua uma interface de comunicação HLA poderá se comunicar um com o outro desde que utilize o mesmo modelo de dados a serem compartilhados.

Nesta dissertação, a proposta é utilizar o simulador Ptolemy, o simulador Stage, dispositivos de hardware e robôs em um mesmo ambiente. O Ptolemy é um simulador com foco sistemas embarcados, foi escolhido por que já possui uma interface de comunicação com a Arquitetura de Alto Nível, desenvolvida por Negreiros e Brito (2012). Já o Stage é um simulador de multi-robôs em tempo real em ambientes 2D. A utilização de simuladores com diferentes focos é interessante porque proporciona visões diferentes da simulação, uma voltada para os softwares embarcados e outra para o comportamento dos robôs.

Com o ambiente descrito será possível inserir robôs de duas formas, uma onde os robôs compartilham informações diretamente com o HLA por meio do desenvolvimento de uma interface de comunicação em Python, outra por meio do compartilhamento das informações no Sistema Operacional de Robôs (ROS).

1.2 Contribuição Principal

Desenvolver um ambiente de monitoramento para sistemas muilti-robos com cossimulação federada e *Hardware-in-the-Loop*.

1.2.1 Especificidades

- Desenvolver modelo de dados (Arquivo FED) que permita compartilhamento de informações de robôs no HLA

- Permitir integração de dispositivos de *hardware* ou robôs com a HLA utilizando modelo de dados proposto
- Viabilizar cossimulação entre os simuladores Ptolemy e Stage
- Realizar experimentos e análise do ambiente

1.3 Metodologia

As tarefas realizadas para alcançar os objetivos deste trabalho se dividem em cinco. A primeira é o desenvolvimento da interface entre Ptolemy e HLA que pode ser utilizada para inserir dispositivos de *hardware* na simulação. A segunda tarefa é o desenvolvimento de uma interface de comunicação entre o ROS e a Arquitetura de Alto Nível, permitindo assim o compartilhamento de dados entre os simuladores bem como a gerência de tempo neste ambiente. O estudo do ambiente é a terceira tarefa e consiste no estudo do ambiente que foi desenvolvido por meio de experimentos e coleta de informações como tempo de resposta. A quarta tarefa consiste em experimentos com o robô espelho, mostrando que é possível utilizar um avatar do robô real no ambiente simulado para detecção de falhas de *hardware*. Já a quinta tarefa são os experimentos com turtlebot mostrando a integração de robôs reais no ambiente proposto.

As tarefas se dividem em atividades, a seguir é possível verificar as atividades da tarefa T1.

A1.1 Desenvolver novo modelo de compartilhamento de dados (Arquivo FED).

A1.2 Desenvolver aplicação em python capaz de entrar em uma federação.

A1.3 Modificar atores existentes do Ptolemy para que possam utilizar novo arquivo FED.

A1.4 Realizar simulação HiL utilizando simulador Ptolemy e HLA para gerenciamento da simulação.

Ao final deste conjunto de atividades, com a realização do experimento A1.4, teremos um ambiente capaz de administrar simulações entre Ptolemy e diferentes dispositivos de *hardware* que suportem a linguagem Python.

As atividades da segunda tarefa (T2) proporcionam a interface entre o ROS e HLA , estão listadas a seguir:

A2.1 Modificar a aplicação para que se torne um federado (Tornando a aplicação o componente Ponte.

A2.2 Realizar experimento para verificar o envio e recebimento dos dados por parte do Ptolemy e ROS.

A aplicação desenvolvida na atividade A2.1 será considerada a ponte de comunicação entre os dois ambientes, o ROS e HLA. O ponto de partida dessa aplicação é basicamente a aplicação desenvolvida na atividade A1.2 da tarefa I, visto que ela já possuí a interface de comunicação com a Arquitetura de Alto Nível e já utiliza o modelo de dados proposto em A1. A realização do experimento da atividade A2.2 vai permitir constatar se o ambiente criado viabilizou a cossimulação federada entre os dois simuladores: Ptolemy e Stage.

Visto que o ambiente proposto já é funcional, decide-se realizar experimentos para verificar o seu funcionamento. A as atividades que compõem a tarefa de estudo do ambiente (T3) são mostradas a seguir:

A3.1 Criar mapa no stage para realizar experimentos.

A3.2 Desenvolver algoritmo de Controle de trajetória a ser utilizado nos experimentos.

A3.3 Criar Script em python para armazenar as informações de trajetórias dos robôs.

A3.4 Realizar Experimentos (utilizar máquinas diferentes, realizar simulações distribuídas e centralizadas, realizar simulações com sobrecarga).

As atividades A3.1, A3.2 e A3.3 da tarefa T3 viabilizam a realização dos experimentos, enquanto que a atividade A3.4 já propõe os experimentos que vão permitir maior conhecimento do ambiente em estudo.

A tarefa T4 possui as atividades referentes aos experimentos com robô espelho, servindo para comparação com o robô virtual e poderá facilitar a busca por falhas de *hardware*.

A4.1 Medição de tempo no HLA (variar frequências e comparar).

A4.2 Detecção de falhas de *hardware*.

Por último temos a tarefa T5 com utilização do robô real Turtlebot. Com a criação de todo este ambiente, a utilização de um robô real servirá para verificar o funcionamento da simulação HiL propriamente. O experimento A4.2 consiste no mesmo experimento A4.1, entretanto uma falha de *hardware* é simulada por meio da modificação de uma roda do robô real. Ao perceber que a trajetória foi diferente é proposta uma modificação no algoritmo de controle para mitigar esta falha. Este grupo de experimentos poderia ocorrer em alguma fase de desenvolvimento do sistema de controle de algum robô, e não seria possível no ambiente virtual. A utilização do robô real permitiu neste caso, encontrar falhas de *hardware* e tentar mitigar por meio de *software* antes mesmo da implantação no ambiente final.

A5.1 Verificar ambiente para robôs.

A5.2 Modificar roda do robô real para simular falha de *hardware*.

A5.3 Mitigar falha por modificação no algoritmo de controle.

1.4 Trabalhos Relacionados

O trabalho de Martin e Emami (2006) demonstra a criação de uma plataforma que permite robôs móveis reais e modelos de computadores interagirem em um ambiente virtual, o presente trabalho também tem como objetivo a criação de um ambiente virtual que possibilite esse tipo de interação. O artigo apresenta ainda uma arquitetura de simulação de tempo real de manipuladores robóticos com a capacidade de ter tanto o módulo controlador como o módulo de juntas (do manipulador robótico) no *loop* da simulação. O trabalho tem como proposta o projeto e teste de módulos conjuntos e do sistema de controle de manipuladores robóticos que pode ser aplicado em qualquer sistema de controle de manipuladores robóticos. Já o presente trabalho permite a simulação HiL entretanto com a ajuda do Simulador Robótico será possível inserir diversos tipos de robôs no ambiente de simulação.

Uma continuação do trabalho (MARTIN; EMAMI, 2006) é proposta por Martin, Scott e Emami (2006) e tem como objetivo simular uma carga dinâmica para movimentação de manipuladores robóticos. O trabalho permitiu a análise e síntese do manipulador robótico,

a capacidade de calcular o torque dinâmico eficientemente e aplicar ao atuador utilizando Simulações HiL. Entretanto, o trabalho tem foco em manipuladores robóticos enquanto que pretendemos algo mais genérico.

Alguns conceitos para estabelecer uma interface entre o sistema em teste (SUT, do inglês *System Under Testing*) e um simulador com HiL são apresentados no trabalho de Schlager, Elmenreich e Wenzel (2006). Estes trabalhos que utilizaram HiL permitem constatar que esta técnica vêm sendo utilizada em diversos trabalhos no projeto de sistemas embarcados, trazendo maior realidade aos dados simulados entre outros benefícios que são apresentados também por Schlager, Elmenreich e Wenzel (2006).

Já no que se refere a trabalhos de cossimulação e simulação de sistemas embarcados, o trabalho de Brito et al. (2013) apresenta benefícios de simulações distribuídas de modelos do Ptolemy. Este simulador é o mesmo software utilizado neste trabalho, com a cossimulação administrada pela HLA.

O trabalho feito por Lane et al. (2001) relata a experiência obtida ao integrar sistema simulado e real utilizando o HLA, com a intenção de tentar facilitar a inserção de novos atuadores ou sensores. Para isso foi criado um *software* chamado Coresim que era formado por seis federações: interface piloto, estação de fundo , propulsão, sensores, dinâmica do veiculo e administração da simulação. Conclui relatando a dificuldade em integrar sistemas reais aos virtuais e evidenciando a ajuda que o Coresim permitiu. Apesar da utilização do HLA, o presente trabalho pretende ser mais generalizável em relação a diversos tipos de robôs e também em um nível mais alto em relação aos tipos de dados que são compartilhados visando ainda a integração de mais de um simulador ao ambiente de simulação.

O trabalho de Schaumont e Verbauwhede (2004) realiza a cossimulação entre o *Instruction Set Simulator* (ISS) e o Gezel. A interface de cossimulação consistiu em dois elementos: a interface de sincronização, que faz o Gezel ser executado em sincronia com o ISS e a interface de troca de dados que era responsável pelo compartilhamento de informações. O objetivo do trabalho foi otimizar o código utilizando uma técnica chamada partial evaluation. O algoritmo utilizado no trabalho foi o Padrão de Criptografia Avançada (*Advanced Encriptation Standard* - AES). Em semelhança a este trabalho há o gerenciamento da sincronização e a troca de dados entre os simuladores, entretanto utilizaremos a Arquitetura de Alto Nível que é um padrão de interoperabilidade de simuladores para gerenciar estes dois

elementos.

Uma arquitetura para simulações *Robot-in-the-loop* é apresentada por Hu (2005). Esta propõe a separação entre os sensores/atuadores e o modelo de tomada de decisão do robô. Assim seria utilizada uma interface que possibilita a utilização de diferentes tipos de sensores para os modelos de decisão. Também cita três passos principais para o desenvolvimento de simulações: o primeiro passo é o desenvolvimento de uma simulação convencional em que todos os robôs são simulados; o segundo envolve a utilização de robôs simulados (com atuadores/sensores virtuais) e robôs reais (variando entre atuadores/reais e virtuais); enquanto que o terceiro passo é um experimento de sistema real onde todos robôs reais executam em um ambiente físico real e possuem sensores e atuadores reais. O presente trabalho busca um ambiente semelhante, em que seja possível unir sensores virtuais e reais em uma mesma simulação, entretanto, ainda com a cossimulação de sistemas embarcados.

Uma continuação do trabalho (HU, 2005) é proposta por Hu e Zeigler (2015). Neste trabalho são apresentados três exemplos de como RiL pode ser utilizada em diferentes situações. O trabalho utiliza um modelo de gerenciamento que é particionado em dois: o gerenciador de tempo e o gerenciador de espaço. Cada robô envolvido na simulação possui um respectivo para si para gerenciar tempo e espaço especificamente de um robô. No primeiro exemplo, um único robô era utilizado para desviar de obstáculos. O segundo envolvia uma formação de robôs, cada robô realizar uma busca para encontrar o outro robô. Neste experimento os robôs não possuem nenhuma conexão direta entre si, embora estejam conectados a um *software* chamado Manager em um Laptop por uma rede sem fio. Assim que os robôs se encontram, o Manager estabelece uma conexão sem fio direta entre os robôs e envia um comando para que se organizem em um time. Este time segue uma formação em que um robô segue os passos do outro robô, um é o líder e outro seguidor. No experimento apresentado o robô I era o robô real enquanto que o robô II era simulado. Já o terceiro experimento, formava uma patrulha robótica a partir de um número indefinidos de robôs (N>1). Esses robôs ficavam em linha, cada um tem com dois vizinhos: um a frente e atrás. Esta simulação mostrou a possibilidade de utilizar mais de um robô real em uma simulação RiL. O sistema do experimento não possuía comunicação ou coordenação global. Em semelhança com este trabalho, temos a seperação lógica dos sensores e atuadores por meio do arquivo FED que especifíca os dados a serem compartilhados no ambiente de simulação com HLA. Em se-

melhança há ainda a utilização de sistemas multi-robôs, que no nosso trablaho é realizada principalmente pelo ROS, permitindo integração com robôs que já utilizam este sistema.

1.5 Estrutura da dissertação

Este trabalho se organiza da seguinte forma: o capítulo 2 explica simulações e termos utilizados neste trabalho referentes; capítulo 3, é uma visão geral sobre robótica e sistemas embarcados; capítulo 4, explica as diversas abordagens de cossimulação que são utilizadas neste trabalho; o capítulo 5 mostra os experimentos realizados no ambiente virtual; capítulo 6 apresenta os experimentos realizados com o turtlebot; e o capítulo 7 apresenta a conclusão do trabalho.

Capítulo 2

Simulação Distribuída

Este capítulo apresenta os conceitos básicos sobre simulações necessários para compreensão do trabalho desenvolvido. Em particular abordamos os aspectos das simulações, o padrão referente a Arquitetura de Alto Nível, a forma como foi realizada sua comunicação com os simuladores, metodologias de projetos de sistemas embarcados, as ferramentas que foram utilizadas, e os trabalhos relacionados.

2.1 Simulações

Antes de definir o que é simulação é necessário entender o que é um modelo, pois é um conceito que compõe a simulação. Um modelo é uma aproximação, representação ou idealização de aspectos selecionados da estrutura, comportamento, operação, ou outras características de um processo do mundo real (IEEE. . . , 1989).

Já a simulação é um modelo que se comporta ou opera como um sistema quando provido de um conjunto de entradas controladas (IEEE. . . , 1989). Ou seja, a simulação envolve o uso de um modelo para prever como algum sistema reage em um dado ambiente.

As simulações envolvem diversos conceitos de tempo (SCHLAGER, 2008), entre eles o conceito de tempo físico, tempo de simulação e *wall time clock*. O tempo físico é consumido com o ambiente físico real, um exemplo é o momento que um determinado evento em observação acontece. O Tempo de simulação representa o tempo físico com a simulação, uma das formas de representar o tempo físico na simulação é utilizar uma variável inteira e incrementar este valor a medida que o tempo avança. Já o *wall clock time*, é dado pelo

intervalo de tempo para realizar determinada atividade.

2.1.1 Classificação das simulações

De acordo com Schlager (2008) as simulações podem ser classificadas quanto ao tipo, distribuição e domínio. Abaixo estes tópicos são explicados.

Tipo de simulação: Os tipos de simulação podem ser distinguidos entre dois, as simulações discretas e simulações contínuas. Nas simulações discretas as variáveis do modelo de simulação são modificadas apenas em instantes discretos. Já a simulação contínua tem seus estados modificados continuamente de acordo com o tempo. As simulações discretas podem ser classificadas simulações de tempo discreto e eventos discretos (GALLA, 2015). Um simulador baseado em simulação de tempo discreto incrementa um relógio passo a passo e também a ativação de eventos quando o seu tempo de ativação é igual ao tempo interno do simulador. Em contraste na simulação de eventos discretos o tempo de simulação avança progressivamente de evento em evento.

Distribuição da simulação: A classificação quanto a distribuição distingue simulação centralizada ou distribuída. A simulação centralizada utiliza um único processador, enquanto que a simulação distribuída envolve a utilização de múltiplos processadores interligados em rede. A principal vantagem de uma simulação distribuída está no ganho de desempenho. Entretanto geralmente a complexidade de desenvolver uma simulação distribuída é maior do que uma simulação centralizada. Simulações complexas frequentemente envolvem diferentes simulações individuais, que são validadas em aspectos diferentes por um ambiente global que precisa ser simulado (BRITO et al., 2013).

Domínio da simulação: Os domínios possíveis em uma simulação são apresentados a seguir:

- Simulação de rede: possui propriedades de uma dada conexão de rede.
- Simulação de protocolo: os objetivos desse tipo de simulação envolvem a validação de características utilizadas em um protocolo de comunicação.
- Simulação de ambiente: promove a simulação física de um ambiente de sistema de tempo real.

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

2.2 Padrão IEEE 1516, a Arquitetura de Alto Nível

Diante das dificuldades enfrentadas no gerenciamento de co-simulações, um padrão para administrar as informações compartilhadas entre os simuladores e o avanço de tempo de simulação foi desenvolvido pelo departamento de defesa (*Department of Defence* - DoD) dos Estados Unidos com o intuito de integrar diversos simuladores militares que eram utilizados em ambiente de conflito militar. Este padrão veio a se chamar Arquitetura de Alto Nível (*High Level Architecture* - HLA).

Existem diversos padrões que especificam a HLA, os que são utilizados neste trabalho são especificações que são atualizações de padrões publicados no ano 2000. São eles: IEEE 1516-2010, IEEE 1516.1-2010 e o IEEE 1516.2-2010. Antes de explicar essas especificações, faz-se necessário esclarecer algumas terminologias desta arquitetura que são apresentadas em no padrão IEEE... (2010b). A Infraestrutura de Tempo de Execução (*Runtime Infrastructure* - RTI) é um *software* que provê um conjunto de serviços definidos pela especificação de interface dos federados, é utilizado pelos federados para coordenar operações e mudança de dados durante o tempo de execução da federação. Uma aplicação federada (ou um federado) é uma aplicação que suporta a interface HLA para o RTI e que é capaz de entrar em uma federação. A federação é o nome de um conjunto de aplicações federadas que possuem um modelo de objeto comum. A figura 2.1 ilustra uma federação formada por dois federados.

O padrão IEEE 1516-2010 descreve as regras de Framework da Arquitetura de Alto Nível e define componentes e responsabilidades dos federados e federações para implementação consistente desta arquitetura. Existem dois objetivos principais na HLA: promover interoperabilidade entre simulações e ajudar no reuso de modelos em contextos diferentes (IEEE..., 2010b).

Os principais componentes que formam a Arquitetura de Alto Nível são:

- Framework HLA, especificação das regras de interação dos federados em uma federação e definir responsabilidades dos federados e federações.

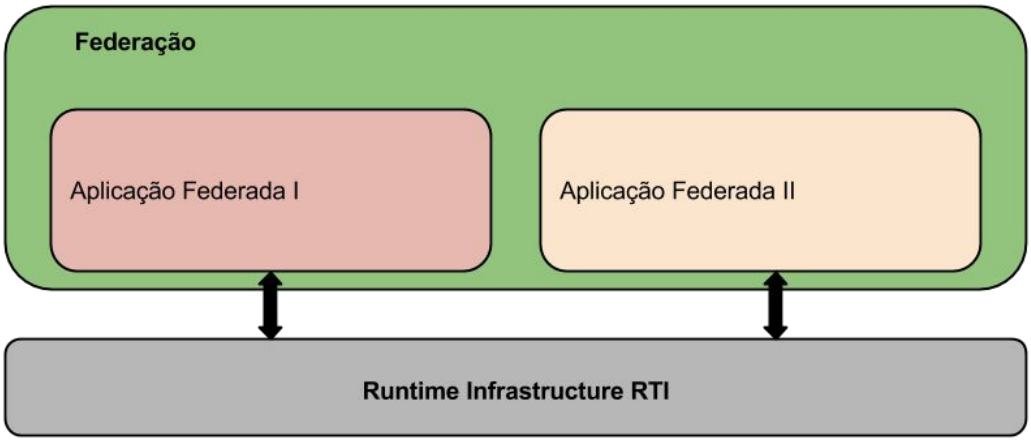


Figura 2.1: Ilustração de uma federação com dois federados

- O *Object Model Template* (OMT) que constitui a base necessária para reuso e documentação padrão descrevendo os dados utilizados por um modelo em particular.
- A especificação da interface dos federados.

2.2.1 Modelo de Objetos

A especificação (IEEE..., 2010b) mostra os três principais modelos de objeto que fazem parte da HLA. O Modelo de Objeto Simulado (*Simulated Object Model* - SOM) é o primeiro, ele trata dos tipos de informação de aplicação federada individual pode oferecer ou receber das federações HLA. O segundo é o Modelo de Objeto da Federação (*Federation Object Model* - FOM) que especifica a informação compartilhada em tempo de execução. Essa informação inclui classes, atributos de objetos de classe, conteúdo do MOM, entre outros. O terceiro modelo é o Modelo de Gestão de Objetos (*Management Object Model* - MOM) que se compõe de um grupo de construtores predefinidos que proporcionam suporte para monitorar e controlar a execução da federação.

2.2.2 Grupos de serviços da Arquitetura de Alto Nível

A especificação IEEE 1516-2010 (IEEE..., 2010b) descreve os grupos de serviços providos pelo RTI. São eles:

Gestão da Federação: são serviços que permitem a coordenação das atividades da fe-

deração ao longo de sua execução. Entre estes serviços estão: criação de federação; entrada na federação; destruição da federação; entre outros.

Gestão da Execução: estes serviços permitem especificar objetos de dados que serão enviados e recebidos. Utilizam o serviço de publicação e subscrição.

Gestão de objetos: dá suporte a atividades de objetos e interações utilizadas por federados. São exemplos o registro, descoberta de instâncias de objetos, entre outros.

Gestão de Posses: Estabelece privilégios a um federado específico para prover valores para um atributo de instância de objeto, bem como facilitar a transferência desse privilégio para outros federados.

Gestão de Tempo: permite aos federados operar com o conceito de tempo lógico para manter um relogio global virtual.

Gestão de Distribuição de Dados: Permite especificar as condições de distribuição para dados específicos. Dessa forma o RTI utiliza essas informaçoes para encaminhar dados dos produtores para receptores de forma mais personalizada

Serviços de Suporte: Serviços diversos que são utilizados na federação. Um exemplo é a inicialização do RTI.

O padrão IEEE 1516.1 (IEEE..., 2010a) proporciona a especificação para interfaces funcionais entre os federados e a RTI, basicamente são um conjunto de funções que servem para que os federados se comuniquem com o RTI. Já o IEEE 1516.2 (IEEE..., 2010c) documenta os modelos de objetos, que já foram brevemente apresentados. A HLA requer que federados e a federação sejam descritos por estes modelos que identificam as trocas de dados durante a execução da federação.

2.3 Comunicação com HLA

Vista a necessidade de um RTI para que seja possível executar simulações com a Arquitetura de Alto Nível, foi necessário escolher uma entre as disponíveis comerciais e não comerciais. Pela experiência adquirida em trabalhos anteriores como (NEGREIROS; BRITO, 2012) (NEGREIROS; BRITO, 2013) (BRITO et al., 2013), optou-se por utilizar a implementação de código aberto chamada CERTI. Esta versão é desenvolvida pela ONERA, um centro francês de pesquisas aeroespaciais. O CERTI é um RTI que está diretamente ligado a admi-

nistração dos dados compartilhados entre as simulações, suporta a especificação HLA 1.3 e parcialmente a versão IEEE 1516-v2010 (C++) (SAVANNAH, 2014).

Escolhida a infraestrutura de tempo de execução, foi necessário escolher também as bibliotecas que deveriam permitir a comunicação de aplicações com o RTI. Os trabalhos (NEGREIROS; BRITO, 2012) (NEGREIROS; BRITO, 2013) (BRITO et al., 2013) utilizaram a JCERTI nos componentes do Ptolemy, o que permitiu ao RTI comunicação com os atores do Ptolemy.

Já existindo a possibilidade de utilizar a linguagem Java para se comunicar com o HLA por meio da JCERTI, foi pensada a possibilidade de utilizar uma nova interface para comunicação, permitindo que aplicações desenvolvidas em novas linguagens se tornassem federados. As linguagens candidatas para comunicação eram C++ e Python, pois estas estão entre as que são utilizadas no Sistema Operacional de Robôs (ROS) que também será utilizado no ambiente proposto por este trabalho. A linguagem Python foi escolhida pelas facilidades de desenvolvimento que acabam por agilizar este processo. A biblioteca escolhida para permitir a comunicação entre HLA e aplicações desenvolvidas em Python foi uma biblioteca de código aberto chamada PyHLA (NONGNU, 2014).

Com a utilização desses três *softwares*, o CERTI, JCERTI e PyHLA será possível que aplicações em Python e Java se tornem federados e se inscrevam em uma federação administrada pelo CERTI RTI.

Capítulo 3

Sistemas Embarcados e Robótica

3.1 Projetos de Sistemas Embarcados

A complexidade dos projetos de sistemas embarcados tem aumentado em taxa exponencial devido a demandas do mercado por novas aplicações e avanços tecnológicos que proporcionaram diversos processadores em um único chip (GAJSKY et al., 2013). Dessa forma os métodos tradicionais em que sistemas são projetados diretamente em baixo nível de *hardware* ou de *software* estão rapidamente se tornando impraticáveis.

Algumas soluções podem ser elevar o nível de abstração utilizado em projetos ou tentar automatizar os processos de projeto de sistemas sempre que possível (GAJSKY et al., 2013). Existe ainda a necessidade de aplicar técnicas de automação de projeto para modelagem, simulação, síntese e verificação para o processo de projeto de sistemas. Por outro lado a automação não é fácil se o nível de abstração do sistema não é bem conhecido. As próximas subseções vão explicar alguns períodos em que aconteceram mudanças consideráveis no fluxo de projeto de sistemas de acordo com (GAJSKY et al., 2013).

3.1.1 Período de captura e simulação (1960-1980)

Nesse período as metodologias de projeto de *hardware* e *software* eram separadas criando um espaço entre produtividade e complexidade de projetos (*design gap*). Os projetistas de *software* testavam algoritmos e ocasionalmente escreviam uma especificação inicial que era enviada aos projetistas de *hardware* que iniciavam o projeto do sistema com a utilização de

um diagrama de blocos. Eles não sabiam se o projeto iria satisfazer a especificação até que o projeto em nível de portas fosse produzido. Quando a *netlist*, um termo utilizado para descrição de um projeto eletrônico por meio de componentes conectados, era capturada e simulada, era possível determinar se o sistema funcionava como esperado. Normalmente não era o caso, tipicamente a especificação era modificada para acomodar capacidades de implementação criando o mito de que a especificação nunca estava completa.

3.1.2 Período de descrever e sintetizar (1980 - 1990)

Os anos 80 trouxeram ferramentas para a síntese lógica que modificou significativamente o fluxo de projeto de sistemas. Projetistas especificavam primeiro o que eles queriam com equações booleanas ou Máquinas de Estado Finitas (*Finite State Machines* - FSM), então as ferramentas geravam a implementação lógica em termos de níveis lógicos da *netlist*. Nessa metodologia o comportamento ou função vinha primeiro e a estrutura e implementações depois. Além disso as descrições eram simuladas, o que era uma melhoria acentuada em relação as metodologias de captura e simulação, por permitir uma verificação mais eficiente.

3.1.3 Metodologias de especificar, explorar e refinar (1990 até os dias atuais)

Esse período adota metodologias que consistem em uma sequência de modelos em que cada modelo é um refinamento do anterior. Esta metodologia, segue um processo natural de projeto onde projetistas especificam a intenção primeiro, então exploram as possibilidades e finalmente refinam o modelo de acordo com as decisões. Este fluxo pode ser visualizado como várias iterações das metodologias que utilizavam o método descrever e sintetizar.

3.2 Ferramentas

Algumas ferramentas que foram utilizadas neste trabalho sendo elas componentes do ambiente de cossimulação proposto são descritas a seguir, nomeadamente o simulador Ptolemy, o Sistema Operacional de Robôs e o simulador Stage.

3.2.1 Ptolemy II

O projeto Ptolemy é formado por um grupo de pesquisadores que se dedica ao estudo de modelagem heterogênea, simulação e projeto de sistemas concorrentes (PTOLEMY, 2014). Eles foram os desenvolvedores da ferramenta de mesmo nome que é utilizada neste trabalho para executar as simulações de sistemas embarcados.

O Ptolemy II é uma ferramenta de código aberto, que funciona como um ambiente para experimentos com simulações heterogêneas. O que caracteriza simulações heterogêneas é a diversidade de Modelos de Computação (*Models of Computation* - MoC's). Um dos motivos para que a ferramenta seja de código aberto é encorajar pesquisadores a desenvolver seus próprios métodos e expandir a infraestrutura fornecida pelo *software*.

Alguns componentes do Ptolemy permitem o uso de hierarquia, isto é, um componente contendo um submodelo. Estes componentes são chamados componentes compostos e são eles quem dão a capacidade para o Ptolemy utilizar heterogeneidade. Sendo assim, o modelo pode utilizar um MoC enquanto o submodelo utiliza um outro MoC.

A Figura 3.1 apresentada no trabalho (BROOKS, 2014) mostra um componente composto do Ptolemy, que possui um submodelo. É possível ver alguns componentes (ou atores) que vão ajudar a compreender melhor este ambiente de simulação. O diretor (*director*) especifica qual modelo de computação é utilizado. As portas (*port*) permitem a comunicação entre atores através de relacionamentos para identificar qual o destino/origem dos dados. A abstração hierárquica é o encapsulamento de um modelo dentro de um único ator, simplificando o modelo e o reuso de componentes.

Além da possibilidade de criar atores utilizando modelos, no ambiente de desenvolvimento do Ptolemy, também é possível criar atores a partir de código na linguagem Java. Pelo projeto ser código aberto, pode-se então criar o código de um ator nessa linguagem e adiciona-lo ao Ptolemy.

3.2.2 Sistema Operacional Robótico - ROS

O Sistema Operacional Robótico (*Robotics Operational System* - ROS) é um sistema operacional licenciado sob uma fonte aberta que é específico para robôs (ROS, 2014). Ele foi utilizado neste trabalho para permitir a partilha informações entre um ambiente de simula-

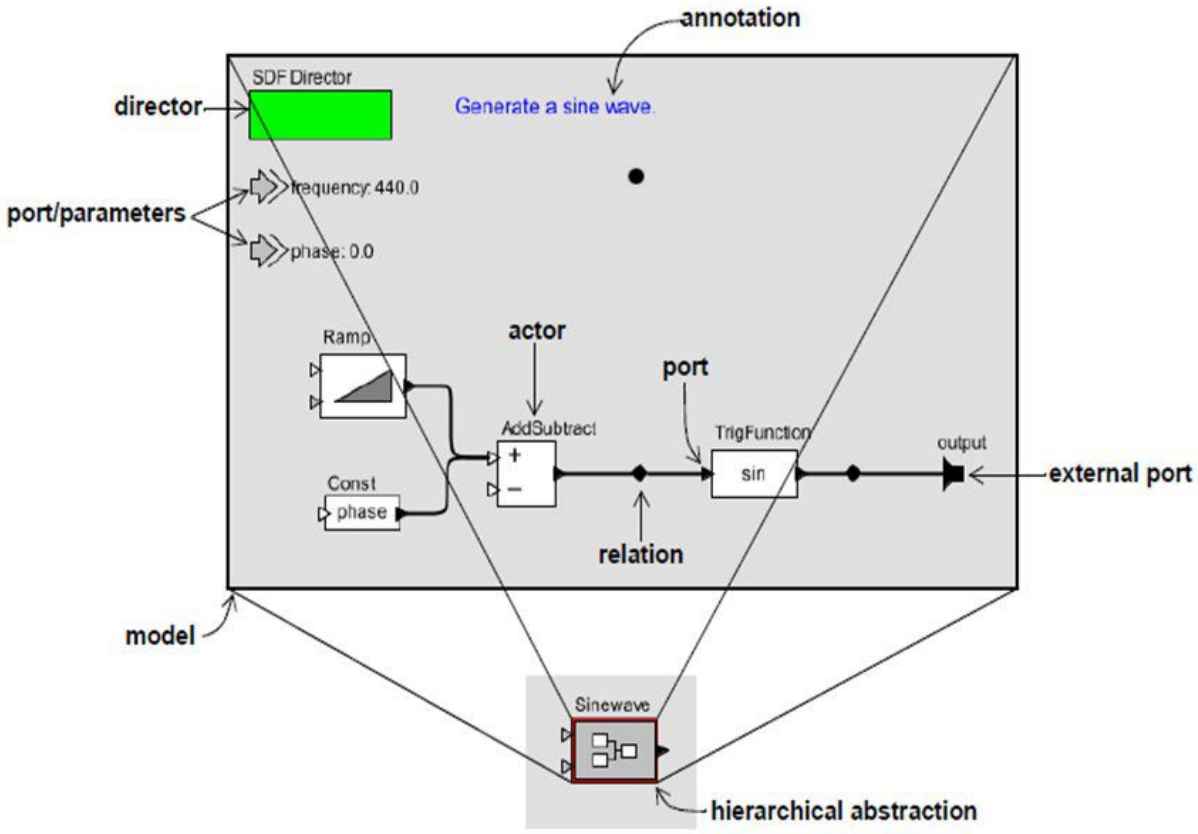


Figura 3.1: Ator Composto e seus componentes

ção robótico onde robôs simulados, robôs reais e simuladores trocam informações entre si. O ROS fornece bibliotecas e ferramentas para ajudar os desenvolvedores de software criar aplicações para robôs.

Entre os objetivos do ROS estão: comunicação *peer-to-peer*, ser baseado em ferramentas, multilinguagens e ser "magro"(QUIGLEY et al., 2009). Para ser um ambiente *peer-to-peer* ele necessita de um mecanismo que auxilie os processos a encontrarem um ao outro. No ROS, este mecanismo é chamado de master ou serviço de nome. No ROS diversas ferramentas são utilizadas para construir e executar vários componentes. Essas ferramentas realizam diversas tarefas como buscar e configurar parametros, visualizar a topologia *peer-to-peer* entre outras atividades.

O ROS busca permitir compatibilidade com diversas linguagens tentando ser neutro em relação a este assunto. Foi projetado para suportar quatro tipos diferentes de linguagens de programação nomeadamente: C++, Python, Octave e LISP. O ROS também encoraja os desenvolvedores de algoritmos e *drivers* a criarem biblitecas sem dependências ao ROS e

o uso de código aberto por facilitar o *debug* especialmente quando o *hardware* e diversos niveis de *software* estão sendo projetados ou testados paralelamente.

Alguns termos da nomenclatura do ROS apresentados em (QUIGLEY et al., 2009) são os nós do ROS, as mensagens e os tópicos. Os nós são processos que executam computação. Estes se comunicam entre si pelo envio de mensagens, estruturas de dados tipadas. As mensagens também podem ser formadas por outras mensagens ou ainda por *arrays* de mensagens. Uma mensagem é publicada em um dado tópico, que é uma string como "odometria"ou "mapa". Um nó que tem interesse em determinado tipo de dado vai se inscrever no tópico apropriado para isso. Concorrentemente podem existir diversos publicadores e inscritores em um mesmo tópico, enquanto que um único nó também pode publicar e se inscrever em diversos tópicos.

3.2.3 Simulador Robótico

Existem diversos simuladores robóticos, neste trabalho três simuladores foram cogitados para serem utilizados: O Morse, o Gazebo e o Stage ROS. Gazebo (GAZEBO, 2014) é um simulador multi-robôs para ambientes externos, capaz de simular vários robôs, sensores e objetos em um mundo tridimensional. O Gazebo gera tanto *feedbacks* realísticos aos sensores como interação física plausível entre objetos.

O Morse (MORSE, 2014) é um simulador genérico para robótica acadêmica. Seu foco é em simulação 3D realística de ambientes pequenos ou grandes, internos ou externos, com um ou dezenas de robôs autônomos. Ele pode ser diretamente controlado da linha de comando. Cenas da simulação são geradas de simples scripts em Python. O Morse vem com um conjunto de padrões de sensores, atuadores, controladores de velocidade, entre outros. Novos componentes também podem ser facilmente adicionados. A renderização deste simulador é baseada na *Blender Game Engine*.

O Stage é um simulador multi-robôs que possibilita o controle dos robôs por sockets, permitindo que diversas linguagens programação possam utilizar esse simulador. Ele simula uma população de robôs, sensores e objetos de ambiente e tem como objetivos: permitir o rápido desenvolvimento de controladores que eventualmente vão controlar robôs reais; possibilitar experimentos robóticos sem acesso ao *hardware* real e sem o ambiente; e o desenvolvimento com sensores que ainda não existem, determinando o benefício de desenvolver

determinado tipo de sensor (GERKEY; VAUGHAN; HOWARD, 2003).

O Stage foi projetado para sistemas multi-agente, proporciona um ambiente muito simples computacionalmente, e modelos de vários dispositivos. Também tem a intenção de ser realista o suficiente para permitir aos usuários mover controladores entre robôs do Stage e robôs reais, enquanto permanece rápido o suficiente para simular grandes números de robôs.

O simulador que pareceu mais adequado para este trabalho foi o Stage. Uma das motivações para utilizar este simulador é a simplicidade e 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 se pretende desenvolver.

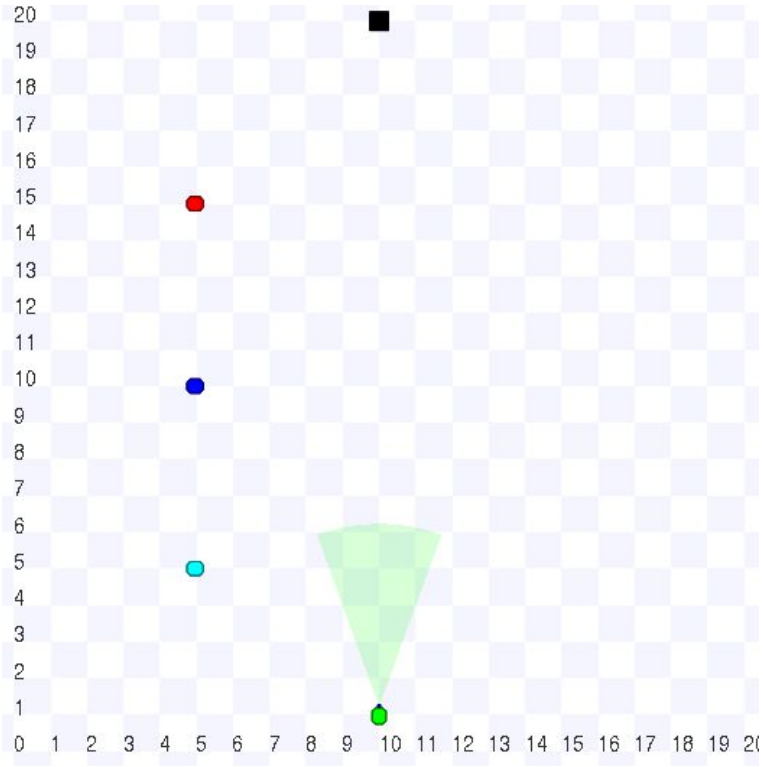


Figura 3.2: Ambiente do Stage

A figura 3.2 mostra a interface gráfica do simulador Stage, nela percebe-se diversos robôs sendo um com sensor de distância e um objeto preto que faz parte do ambiente simulado. Entre alguns aspectos que fazem o Stage adequado para sistemas multi-robôs estão: boa fidelidade; população de robôs escalável, pois os algoritmos utilizados são independentes do tamanho da população de robôs; modelos e dispositivos combináveis; e a interface Player (GERKEY; VAUGHAN; HOWARD, 2003).

O Stage tem compatibilidade com o Sistema Operacional Robótico. Isto torna possível

que algoritmos desenvolvidos no ROS que utilizam tópicos para se comunicar também se comuniquem com este simulador por tópicos para acessar ou enviar dados.

3.3 Turtlebot

O turtlebot (ROS, 2015), apresentado na figura 3.3, é uma plataforma de *hardware* de código aberto. Por meio do ROS é possível que ele lide com visão, localização, comunicação e mobilidade. Ele é capaz de mover objetos que estão em sua base para algum local alvo desviando de obstáculos no caminho. Os principais componentes de *hardware* são: a base Kobuki, um netbook compatível com o ROS, um kinect, a estrutura do robô entre outros componentes.

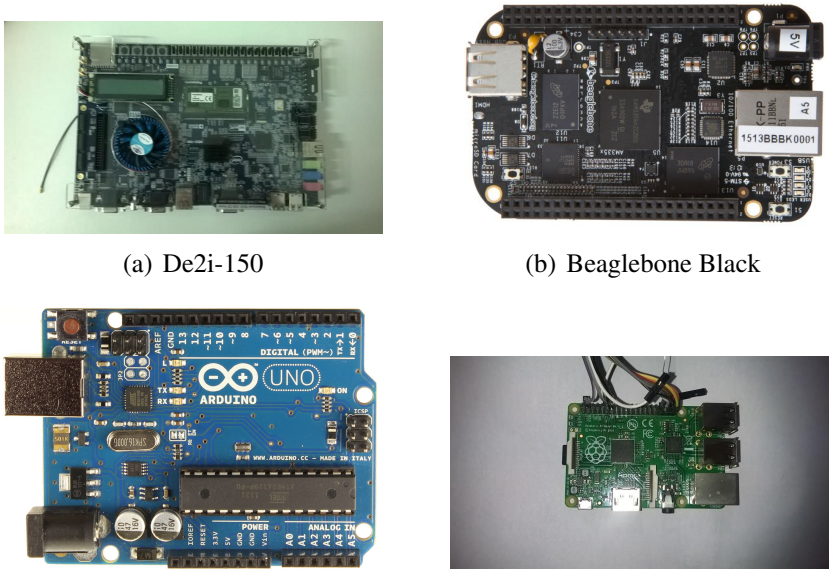
Figura 3.3: Turtlebot

3.4 Plataformas de desenvolvimento

No presente trabalho algumas placas microcontroladoras foram utilizadas e testadas no ambiente de simulação. Entre elas a Beaglebone Black (BBB) que faz parte da família de placas da BeagleBoard. Ela tem baixo custo, utiliza um processador Cortex A8 ARM da *Texas Instruments*. É similar a Beaglebone, mas com algumas características diferentes (BOARD, 2014). Têm capacidade de executar um sistema Linux, e neste trabalho foi utilizada a distribuição Debian específica para esta placa.

A DE2i-150 é uma plataforma de sistemas embarcados que combina um processador embarcado Intel N2600 com a flexibilidade de uma FPGA Altera Cyclone IV GX. É um sistema computacional que funde a capacidade de processamento de alta performance e grande configurabilidade. O Processador Intel Atom e o dispositivo FPGA são ligados por meio de duas PCI que garantem uma comunicação entre eles de alta velocidade (TERASIC, 2014). O sistema operacional utilizado nesta placa foi o Xubuntu, uma versão do Linux Ubuntu que utiliza uma interface gráfica mais simples, permitindo melhor desempenho nas atividades realizadas neste trabalho.

Arduíno é uma plataforma de computação física de código aberto baseada em uma simples placa microcontroladora. Ele pode ser utilizado para desenvolver objetos interativos, pegar dados de uma variedade de sensores, e controlar várias luzes, motores e outros dispositivos físicos (ARDUINO, 2014). Também podem se comunicar com *softwares* executando em um computador . Este dispositivo foi utilizado para auxiliar no controle de sensores e atuadores de um robô simples em parceria com alguma outra controladora com Linux. Dessa forma ela foi programada para se comunicar com as demais placas via USB.



(a) De2i-150 (b) Beaglebone Black

(c) Arduíno (d) Raspberry Pi

Figura 3.4: Plataformas de desenvolvimento utilizadas nas simulações

O Raspberry Pi (FOUNDATION, 2015) é um computador de baixo custo que pode ser plugado em um monitor de computador ou em uma televisão e utiliza teclado e mouse padrão. Tem capacidade de realizar algumas atividades de computadores desktop como navegar

na internet visualizar vídeos em alta definição, jogar, entre outras atividades. Neste trabalho foi utilizada para controlar um Arduíno, utilizando o Raspbian que é uma versão do Debian Linux otimizada para a Raspberry Pi. A figura 3.4 ilustra as plataformas utilizadas.

Capítulo 4

Abordagem de cossimulação

Este capítulo descreve de forma incremental, todo ambiente que foi desenvolvido para o estudo de cossimulação de equipes de robôs.

4.1 Abordagem de software

A abordagem utilizada para integrar os simuladores Stage e Ptolemy, bem como dispostivos de *hardware*, foi baseada na Arquitetura de Alto Nível. A Figura 4.1 demonstra como se dá a integração de dois simuladores utilizando esse padrão de interoperabilidade. Cada simulador possui uma interface de comunicação com a HLA e desta forma podem compartilhar dados entre si.

A HLA recebe informações dos simuladores e repassa aos demais federados gerindo o compartilhamento de informações entre todos os federados, assim como a passagem de tempo, ou seja, a sincronização dos federados. Isso faz com que os simuladores que estejam conectados avancem os passos da simulação de forma conjunta, impossibilitando que federados fiquem mais avançados ou atrasados em realação aos demais. Dessa forma, o tempo de simulação se dá pelo federado mais lento, visto que os outros esperam que este termine sua atividade para poderem avançar no tempo todos juntos.

A Figura 4.2 apresenta a Arquitetura de Alto Nível sendo utilizada para inserir um robô real na simulação. Isso é possível a partir do desenvolvimento de uma aplicação em Python que é utilizada pelo robô para se comunicar com a HLA. Por incluir no *loop* da simulação um robô temos uma simulação *Robot-in-the-loop*. O termo *RunTime Infrastructure Gateway*

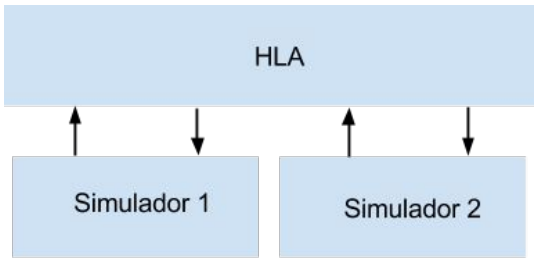


Figura 4.1: Integração de simuladores com a HLA

(RTIG) apresentado na figura é o processo que coordena todos os federados envolvidos na cossimulação.

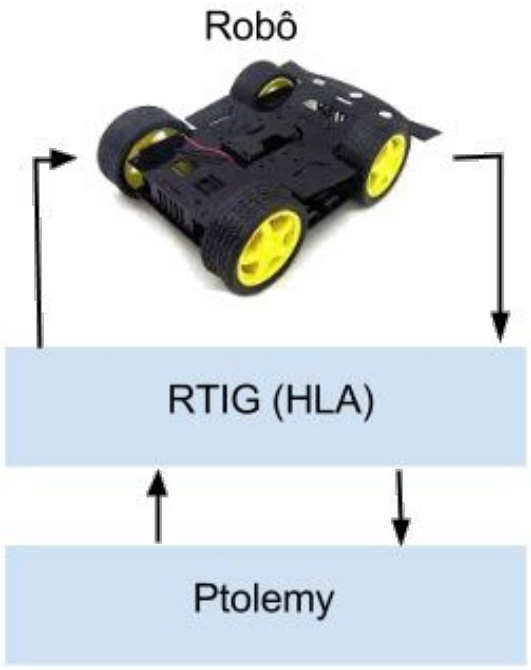


Figura 4.2: Abordagem de simulação com robô real

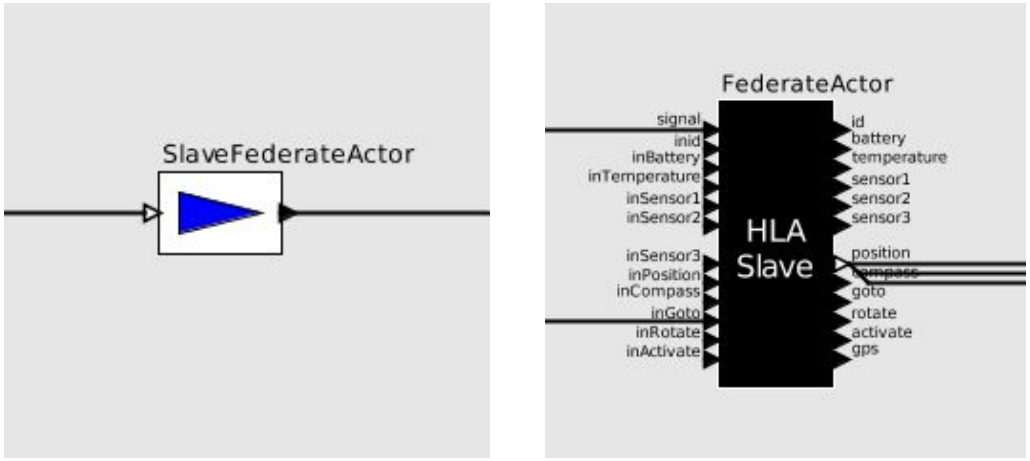
Para possibilitar este tipo de cossimulação, foi desenvolvido um federado na linguagem Python que pudesse ser utilizado em dispositivos de *hardware* para permitir compartilhamento de informações do *hardware* e gerenciamento de tempo. A aplicação utiliza a biblioteca PyHLA que provê funções que permitem comunicação com a HLA. Também foi necessário desenvolver um arquivo FED (código completo disponível no apêndice A) que especificou um tipo de objeto que descrevia os dados que seriam compartilhados entre os

simuladores. O código 4.1 mostra a classe robot do arquivo FED contendo os principais atributos utilizados.

Código Fonte 4.1: Classe Robot no arquivo FED

```
1  (class robot
2          (attribute id reliable timestamp)
3          (attribute battery reliable timestamp)
4          (attribute temperature reliable timestamp)
5          (attribute sensor1 reliable timestamp)
6          (attribute sensor2 reliable timestamp)
7          (attribute sensor3 reliable timestamp)
8          (attribute gps reliable timestamp)
9          (attribute compass reliable timestamp)
10         (attribute goto reliable timestamp)
11         (attribute rotate reliable timestamp)
12         (attribute activate reliable timestamp)
13 )
```

As variáveis escolhidas para compor esta classe foram: o id, responsável por identificar os dispositivos no ambiente; a variável bateria, que compartilha o estado de carregamento da bateria; os sensores 1, 2 e 3 que são utilizados com propósito genérico; a variável temperatura informa a temperatura do dispositivo; posição, é responsável por informar a posição do robô de acordo com seu sistema de localização; a bússola compartilha a direção do dispositivo; a variável rotate pode ser utilizada para rotacionar o robô; activate para ativar ou desativar sensores ou atuadores do robô; e goto é utilizado para enviar comandos para mudar a posição do robô.



(a) Ator antigo (b) Novo ator

Figura 4.3: Atores que permitem comunicação do Ptolemy com HLA

Com a especificação do arquivo FED concluída, foi necessário desenvolver atores do Ptolemy que utilizariam esta especificação para se tornar parte da federação. Trabalhos anteriores (BRITO et al., 2013) (NEGREIROS; BRITO, 2013) já demonstram o desenvolvimento de atores que permitem a comunicação do simulador Ptolemy com a Arquitetura de Alto Nível, entretanto foram necessárias modificações nestes atores para permitir a utilização do novo arquivo FED. Estas modificações foram realizadas na linguagem Java (linguagem em que o simulador Ptolemy é desenvolvido) e consistiram em modificar as portas de entrada e saída bem como o gerenciamento da informação recebida e enviada pela HLA. A figura 4.3(a) mostra o ator antes das modificações, enquanto que a figura 4.3(b) mostra o ator após as modificações para o presente trabalho, já apresentando as portas de entrada e de saída referentes ao novo arquivo FED.

4.1.1 Experimento de comunicação entre Robô e Ptolemy

Um experimento foi realizado para verificar o funcionamento da comunicação entre o robô e o simulador Ptolemy. O robô utilizado é apresentado na Figura 4.4, ele é formado por um eixo com quatro rodas, uma Raspberry e um Arduino, sendo a aplicação em Python com interface com o HLA executada na Raspberry. Para guiar o gerenciamento de tempo da aplicação federada, foi necessário verificar como ele foi realizado no componente do Ptolemy *SlaveFederateActor*. Dessa forma as funções e estrutura utilizadas para comunicação foram semelhantes, apesar de serem desenvolvidas em linguagens diferentes. Assim, o federado realiza um loop de execução e final envia uma mensagem ao RTI requisitando o avanço de tempo, enquanto não obtiver resposta fica em espera. Por outro lado o RTI só envia a mensagem permitindo o avanço de tempo da simulação quando todos federados tiverem requerido o avanço de tempo, permitindo o avanço de tempo igualmente.

O trecho de código 4.2 é responsável por configurar a federação para comunicação com o simulador Ptolemy durante a simulação. Nele percebemos a criação da variável rtia por meio da bilioteca PyHLA, dessa forma será possível utilizar funções da biblioteca para comunicação com o RTI. Em seguida é criada a variável mya do tipo MyAmbassador, esta classe foi desenvolvida para atender as variáveis do novo arquivo FED (o código completo se encontra no apêndice C).

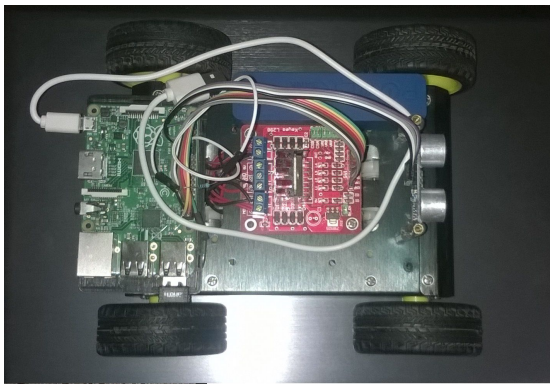


Figura 4.4: Imagem do Robô

Código Fonte 4.2: Configuração da federação

```
1  ############################
2  ### Federation Setup   ###
3  ############################
4
5  print("Create ambassador")
6  rtia = hla.rti.RTIAmbassador()
7  mya = MyAmbassador()
8
9  #Create a federation
10 try:
11     rtia.createFederationExecution("ExampleFederation", "PyhlaToPtolemy.fed")
12 except hla.rti.FederationExecutionAlreadyExists:
13     print ("Federation already exists.\n")
14
15 #join in a federation
16 mya = MyAmbassador()
17 rtia.joinFederationExecution("uav-recv", "ExampleFederation", mya)
18
19
20 mya.initialize()
21 x = input ("Waiting for other federates.\n")
22
23 #Archieve Synchronized Point
24 while (mya.isAnnounced == False):
25         rtia.tick()
26
27 rtia.synchronizationPointAchieved("ReadyToRun")
28 while (mya.isReady == False):
29         rtia.tick()
30
31 # Enable Time Policy
32 currentTime =rtia.queryFederateTime()
33 lookAhead =1
```

```
34
35 rtia.enableTimeRegulation(currentTime, lookAhead)
36 while (mya.isRegulating == False):
37         rtia.tick()
38
39 rtia.enableTimeConstrained()
40 while (mya.isConstrained == False):
41         rtia.tick()
```

Na função de criação da federação é necessário indicar o nome da federação que desejamos criar e o nome do arquivo FED, no caso da federação já existir uma mensagem é enviada ao usuário. Após a criação da federação é necessário ingressar na mesma, para isso a função *joinFederationExecution* permite que sejam indicados os nome do federado e da federação que desejamos entrar respectivamente. A função *initialize* da classe *MyAmbassador* é invocada, neste momento o federado indica ao RTI quais variáveis do arquivo FED deseja enviar e receber informações.

Em seguida, a aplicação espera que o usuário digite alguma entrada qualquer para dar continuidade ao processo. Este tempo de espera é necessário para que os outros federados sejam iniciados.

Continuando a leitura do código, encontramos um *while* que bloqueia a continuidade da aplicação até que a variável *isAnnounced* seja verdadeira. Na classe *MyAmbassador* existe um método chamado *announceSynchronizationPoint*, este método é uma função de *callback* que é ativada pelo RTI. Quando o programa der continuidade a execução teremos então a invocação da função *synchronizationPointAchieved*, que fará com que o RTI ative a *callback* *federationSynchronized* e esta modificará o valor da variável *isReady* dando continuidade ao programa.

Em seguida são ativadas as funções *enableTimeRegulation* e *enableTimeConstrained*, configurando variáveis do RTI para utilizarem a marca temporal (*timestamp*) no envio e recebimento das mensagens. As variáveis *isRegulating* e *isConstrained* também são modificadas por *callbacks* implementadas na classe *MyAmbassador* e ativadas pelo RTI.

O código 4.3 mostra como foi realizado o *loop* principal do robô. No primeiro momento temos o recebimento de informações do Ptolemy, ao receber novas informações a variável *hasData* é modificada para *True*. A variável goto recebe o valor do AttMap que contém todas os valores reccebidos pelo RTI. A variável goto é então comparada com os valores 0, 1, 2

e 3 sendo que cada número equivale a um comando. A classe robot permite mover o robô para frente, para trás e parar o robô. Ao utilizar o valor recebido pelo RTI a variável attMap é reiniciada e *hasData* é modificado para *False*.

Código Fonte 4.3: Loop principal de controle do robô

```
1  ################
2  ## Main loop  ##
3  ################
4  try:
5          while True:
6                  cont+= 1
7                  mya.id = 1
8                  #receive data from HLA
9                  if mya.hasData==True:
10
11                         _goto = mya.attMap["goto"]
12
13                         if (_goto.count("0") == 1):
14                                 robot.para()
15
16                         if (_goto.count("1") == 1):
17                                 robot.frente()
18
19                         if (_goto.count("2") == 1):
20                                 robot.para()
21
22                         if (_goto.count("3") == 1):
23                                 robot.tras()
24
25                         mya.hasData = False
26                         mya.attMap = {}
27
28                 #Send data to HLA
29                 sensor2 = int(robot.readSensor())
30
31                 rtia.updateAttributeValues(mya.myObject,
32                         {mya.idHandle:"1 ",
33                         mya.batteryHandle:"Bateria ",
34                         mya.temperatureHandle: "temperatura ",
35                         mya.sensor1Handle:"sensor1 ",
36                         mya.sensor2Handle:str(sensor2),
37                         mya.sensor3Handle:"sensor3 ",
38                         mya.gpsHandle: "<0;0> ",
39                         mya.compassHandle:"compass ",
40                         mya.gotoHandle:"goto ",
41                         mya.rotateHandle:"rotate " ,
```

```
42                      mya.activateHandle:"activate "},"update")
43
44
45              ######  Time Management  ########
46              timeHLA = rtia.queryFederateTime() + 1
47              rtia.timeAdvanceRequest(timeHLA)
48              while (mya.advanceTime == False):
49                      rtia.tick()
50              mya.advanceTime = False
51              ################################
```

Segue então a parte em que o valor do sensor de distância do robô é atribuído a variável *sensor2* e então enviado ao Ptolemy por meio da função *updateAttributeValues*. Em seguida o federado incrementa o tempo atual e requisita o avanço de tempo ao RTI por meio da função *timeAdvanceRequest*. O federado permanecerá no *loop* enquanto o RTI não utilizar a *callback timeAdvanceGrant* (implementada na classe *MyAmbassador*) indicando que os federados podem avançar para a próxima iteração da simulação.

A Figura 4.5 apresenta o experimento proposto, o Ptolemy possui um ator chamado Clock que é responsável por gerar eventos para os demais componentes, ao receber estes eventos o ator Gerador gera um número que equivale a um comando para o robô. Fisicamente, o robô está posicionado em frente a uma parede e possui um sensor de distância na frente do robô. Isso permite obter a distância aproximada do robô até a parede.

Os comandos gerados pelo Ptolemy são então enviados pelo componente HLA ao RTI. Dessa forma o RTI envia esta informação ao federado que controla o robô que interpreta os dados, ativa os atuadores e sensores do robô e envia um log dos sensores de volta ao simulador Ptolemy. Estes dados são então utilizados para gerar um gráfico que apresenta a distância do robô à parede no decorrer do tempo, o gráfico é apresentado na Figura 4.7. Todos estes passos, são executados de forma que o robô fica em sincronizado com o simulador Ptolemy, visto que ambos têm o avanço de tempo gerenciado pela HLA.

O robô utilizado no experimento é apresentado na figura 4.4. O modelo utilizado na simulação se encontra na figura Figura 4.6, nela é possível ver o componente gerador, o clock, o plotxy que é responsável por gerar o log.

Os comandos eram executados em uma sequência que fazia o robô se afastar da parede e em seguida se aproximar. O sensor de distância detectou a distância do robô em relação a parede e estes dados serviram para criar um log. A partir dos dados temos a Figura 4.7. Nela

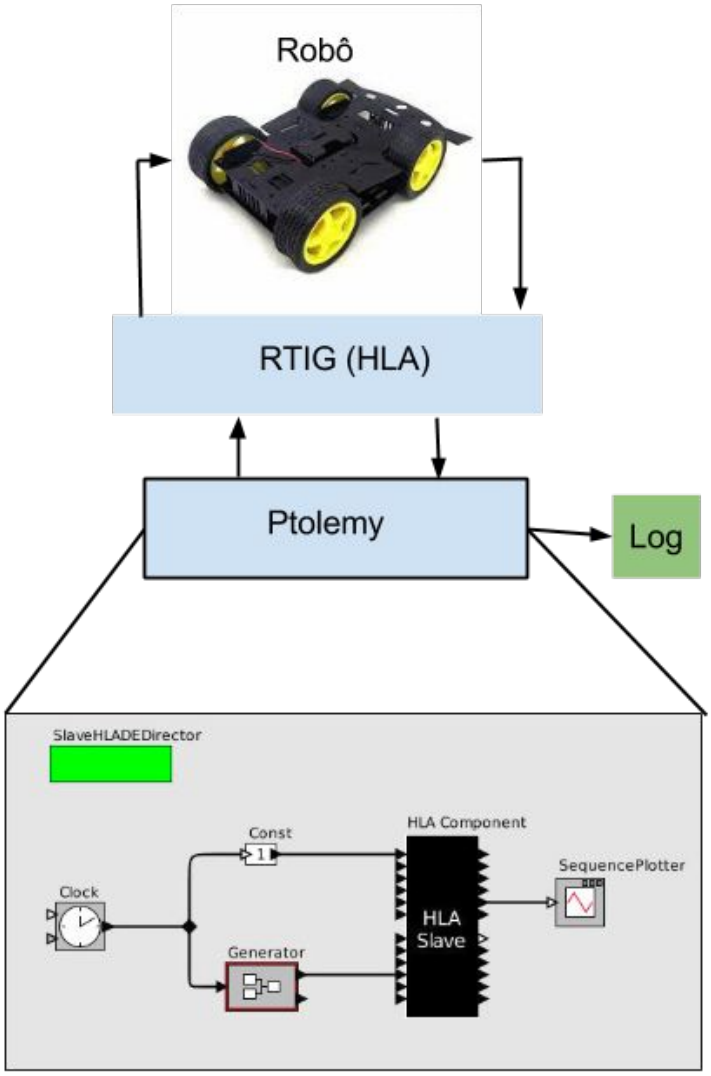


Figura 4.5: Experimento com HiL

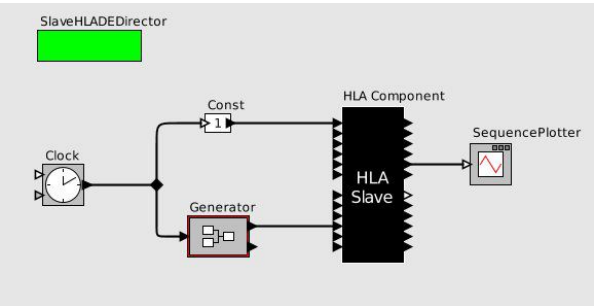


Figura 4.6: Modelo utilizado no Ptolemy

é possível verificar que a distância do robô diminuia e aumentava com o passar do tempo de simulação.

Um aspecto positivo de inserir componentes de *hardware* no ambiente de simulação da forma que é apresentada é que não apenas robôs simples podem ser inseridos no ambiente, mas também dispositivos de *hardware* que executem um sistema operacional que permita a execução de aplicações Python.

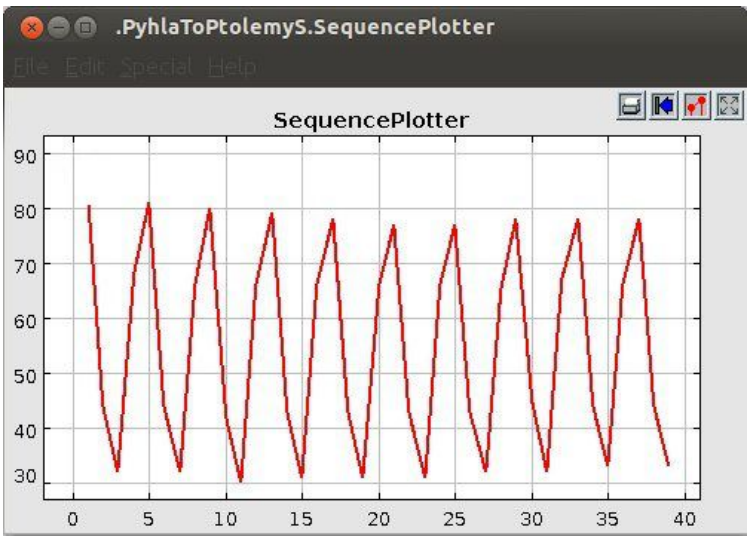


Figura 4.7: Gráfico da distância do robô

A partir da organização anterior, uma nova abordagem foi utilizada para gerar um sistema robótico completamente simulado que utiliza um motor de simulação comum, o simulador Stage. A Figura 4.8 ilustra como é organizado este sistema. Nela é possível perceber a adição do simulador Stage e de um componente chamado ponte. O simulador Stage tem como função simular a física dos robôs, e neste caso simula um robô virtual. Enquanto que o Ptolemy possui o algoritmo de controlo utilizado pelos robôs para se locomoverem. A ponte é o componente responsável por interligar o ambiente ROS com o ambiente HLA.

Para o desenvolvimento da ponte, foi necessário utilizar a aplicação federada em Python que já havia sido desenvolvida nos passos anteriores para simulações HiL com HLA. Ela foi modificada de forma que além de permitir comunicação com a Arquitetura de Alto Nível, também pudesse permitir a comunicação com o ROS e assim compartilhar informações com o simulador Stage. Para isso, a biblioteca rospy foi utilizada para permitir que a ponte se inscrevesse ou publicasse nos tópicos de interesse para compartilhar informações dos robôs. A ponte é um componente da HLA, portanto também tem seu avanço de tempo sincronizado com os demais federados.

Já o algoritmo de controle (disponível no apêndice B) utilizado neste trabalho pelo simu-

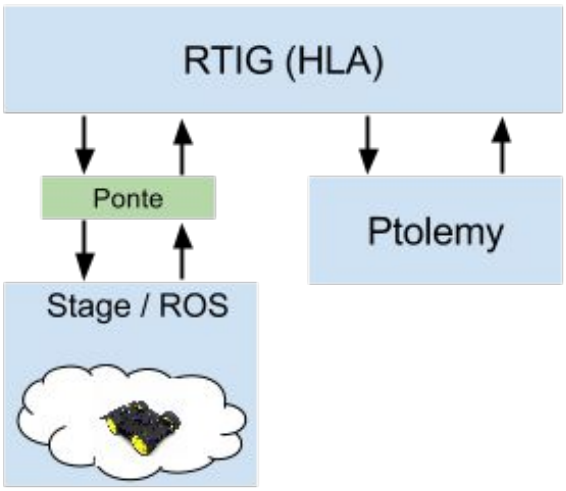


Figura 4.8: Abordagem com simulador Stage

lador Ptolemy, também foi desenvolvido utilizando a linguagem Python. Isso foi possível por meio da utlização de um ator chamado Python Actor, que permite a inclusão de scripts em Python no simulador Ptolemy. O algoritmo consiste em que, a partir da informação de posição do robô que controla e da posição do robô líder ele tenha como saída a velocidade angular e linear que deve ser publicada no ambiente ROS para mover o robô para entrar em formação com o robô lider. No caso do robô utilizado ser o robô líder, ele vai se dirigir a um ponto específico do mapa que será o *target*. Quando o número de robôs utilizado na simulação for de apenas um robô, este robô será o líder.

A partir da abordagem anterior, pode-se utilizar uma abordagem multi-robô totalmente simulada que é apresentada na Figura 4.9. Neste caso, o ptolemy teria mais de uma instância, cada uma responsável por controlar um robô do ambiente ROS. Em relação as pontes, seria necessário adicionar uma ponte para cada instância do Ptolemy, sendo assim cada uma responsável por um robô. Já o stage permaneceria apenas com uma instância, visto que é um ambiente multi-robôs, podendo então simular vários robôs ao mesmo tempo. Nesta abordagem, cada robô está sincronizado pelo simulador Ptolemy respectivo.

4.1.2 Experimento de comunicação entre ROS e HLA

Dada esta abordagem um experimento foi realizado para verificar a comunicação entre o Simulador Ptolemy e Simulador Stage. O componente Ponte (código disponível no Apêndice

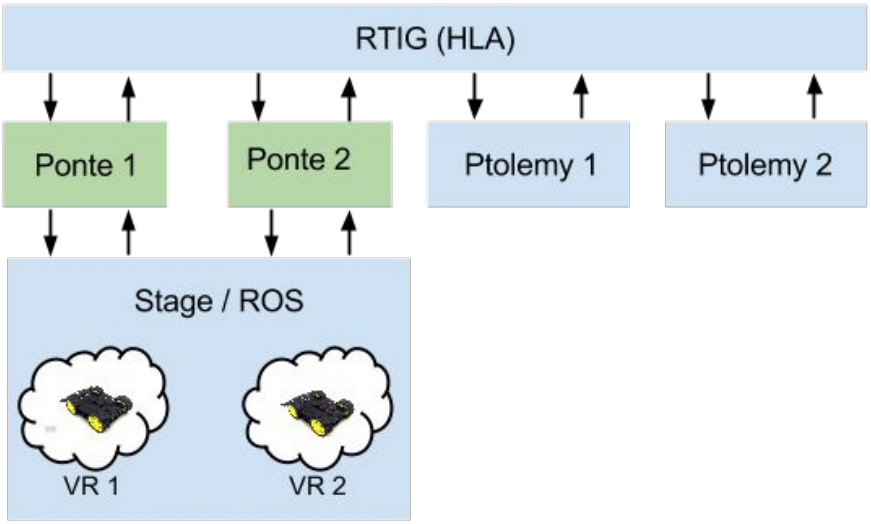


Figura 4.9: Abordagem multi-robôs toltamente simulada

C) foi desenvolvido utilizando a bilioteca rospy para comunicação com o ROS e PyHla para a comunicação com o RTI. No experimento, três robôs foram posicionados em um mapa do simulador Stage enquanto que seus respectivos algoritmos de controle estavam no ambiente Ptolemy. Cada robô era controlado de forma distinta, o primeiro deveria se locomover em circulos no sentido horário, o segundo no sentido anti-horário e o terceiro andaria com base em valores obtidos pela função *random* da linguagem Python.

O trecho de código 4.4 é um exemplo simplificado da aplicação ponte que se comunica tanto com o ambiente ROS como com o HLA.

A princípio temos duas chamadas de retorno getPos0 e getPos1 que armazenam os dados enviados como argumentos na variável odom em uma variável *positions* do tipo dicionário. A primeira função serve para armazenar as variáveis x, y e z do robô líder (id = 0) enquanto que a segunda armazena informações de um robô de id = 1. A terceira função serve para verificar se há informações para serem enviadas ao HLA, isso é feito a partir dos indices da variável *positions*.

A função getDataFromROS() retorna os valores x e y do robô líder e x, y e z do robô 1, apagando logo em seguida os valores da variável *positions*. Já função *sendData* serve para enviar informações para o RTI por meio da função *updateAttributeValues*.

Código Fonte 4.4: Exemplo de utilização do ROS para comunicação

```
1 def getPos0(odom):
```

```
2           global positions
3           positions["leader"] = [round (odom.pose.pose.position.x), round (odom.pose.pose.
                position.y), round (odom.pose.pose.orientation.w)]
4   def getPos1(odom):
5           global positions
6           positions["my"] = [round (odom.pose.pose.position.x), round (odom.pose.pose.
                position.y), round (getDegreesFromOdom(odom))]
7
8   def hasDataToHLA():
9           global positions
10          return positions.has_key("my") and positions.has_key("leader")
11
12  def getDataFromRos():
13          global positions
14          x, y, z = positions["leader"]
15          mx, my, mz = positions["my"]
16          positions = {}
17          return x, y , mx, my, mz
18
19
20  def sendData(idrobot ,battery="", temperature="", sensor1="", sensor2="", sensor3="", gps="
        <0;0>",compass="", goto="", rotate="", activate=""):
21          global mya
22          rtia.updateAttributeValues(mya.myObject,
23          {mya.idHandle:str(idrobot)+" ",
24          mya.batteryHandle:str(battery)+" ",
25          mya.temperatureHandle: str(temperature)+" ",
26          mya.sensor1Handle:str(sensor1)+" ",
27          mya.sensor2Handle:str(sensor2)+" ",
28          mya.sensor3Handle:str(sensor3)+" ",
29          mya.gpsHandle: str (gps)+" ",
30          mya.compassHandle:str(compass)+" ",
31          mya.gotoHandle:str(goto)+" ",
32          mya.rotateHandle:str(rotate)+" " ,
33          mya.activateHandle:str(activate)+" "},
34          "update")
35
36  ######################
37  ##ROS Configuration###
38  ######################
39
40  rospy.init_node('rosNode')
41  #subscribers
42  rospy.Subscriber("/robot_0/base_pose_ground_truth",  Odometry, getPos0)
43  rospy.Subscriber("/robot_1/base_pose_ground_truth",  Odometry, getPos1)
44
45
```

```
46 global p
47 p = rospy.Publisher("robot_1/cmd_vel", Twist)
48
49 r = rospy.Rate(1)# hz
50
51
52
53 ######################
54 #### Main Loop ######
55 ######################
56
57 try:
58         while not rospy.is_shutdown():
59                 #receive data
60                 if mya.hasData():
61                         evento = mya.getData()
62                         _goto = evento["goto"]
63                         _rid = evento["id"]
64                         if (_rid.count(mId) ) >0):
65                                 if (_goto.count("none")<1):
66                                         lin, ang = _goto.split(";")
67                                         twist = Twist()
68                                         twist.linear.x = round (float (lin), 2)
69                                         twist.angular.z = round (float (ang), 2)
70                                 p.publish (twist)
71                 #send data
72                 if hasDataToHLA():
73                         x, y, mx, my, mz = getDataFromRos()
74                         sendData(mId, "", "", x, y, "0", "<" + str(mx)+   ";"  + str(my) +
                                ";" + str(mz)+ ">", "", "", "", "")
75                 #######  Time Management  ########
76                 timeHLA = rtia.queryFederateTime() + 1
77                 rtia.timeAdvanceRequest(timeHLA)
78                 while (mya.advanceTime == False):
79                         rtia.tick()
80                 mya.advanceTime = False
81                 ################################
82                 r.sleep()
```

Entrando na configuração do ROS propriamente dita, percebemos a função init_node(n) que permite uma aplicação python se tornar um nó do ROS com o nome n. A seguir temos os *subscribers*, responsáveis por cadastrar a aplicação nos tópicos de seu interesse. O primeiro argumento da função é o tópico, o segundo é o tipo de dado do tópico enquanto que o terceiro é a função escolhida como função de retorno que sempre será invocada quando

houver modificações no tópico.

O *publisher* é responsável por indicar os tópicos que a aplicação deseja enviar informações, o *Twist* é o tipo de dado do tópico. A função *Rate* é reponsável por indicar quantas vezes o *loop* principal será ececutado por segundo.

No *loop* principal incialmente é verifcado se alguma informação foi recebida do RTI utilizando a função *hasData()*, havendo informações se o identificador (mID) for igual ao do federado é porque a mensagem é para ele. Verifica então se a variável *goto* não está em branco (com valor *None*). Em seguida a função *split* separa os valores da variável *goto* em velocidade angular e linear que estavam no formato "velocidadeAngular;velocidadeLinear". Estes valores são utilizados na variável *Twist*, o tipo utilizado pelo tópico que controla a movimentação do robô e então publicados no ROS fazendo o robô se mover.

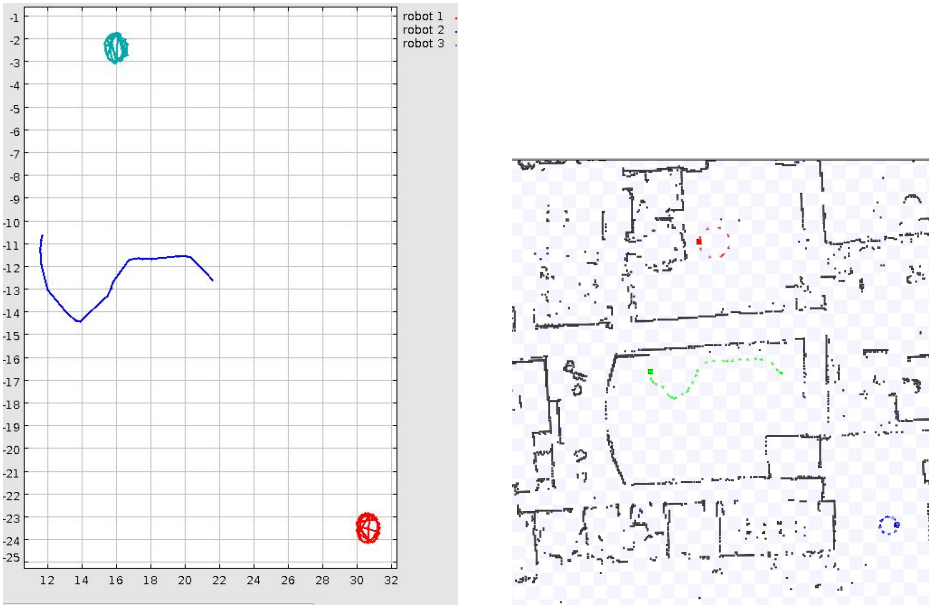
Para enviar dados ao ROS, é verificado se há informações a serem enviadas por meio da função *hasDataToHLA()*. Se houver valores, são utilizados pela função sendData para enviar os dados ao RTI.

Em seguida temos a gerência do tempo, que já foi explicada em códigos anteriores e a função *sleep* que fará com que a aplicação entre em modo de espera de acordo com o valor estipulado na função *Rate*.

No experimento, a ponte além de enviar os dados de controle ao ROS, recebia os valores de posição dos robôs e enviava-os ao Ptolemy para que fossem plotados. Isso permitiu a geração do gráfico apresentado na figura 4.10(a).

A figura 4.10(b) mostra o ambiente Stage durante o experimento. A partir da comparação das figuras, é possível verificar que a informação de posicionamento dos robôs foi compartilhada entre os dois ambientes.

Pode-se ainda utilizar uma técnica de *Robot-in-the-loop* em um ambiente multi-robôs, como é apresentada na figura 4.11. Para que isso seja possível, é necessário adicionar o federado em Python que foi desenvolvido anteriormente, ele seria executado no robô R1. Para seu controlo, também seria necessário a utilização de uma instância do Ptolemy específica chamada Ptolemy R1. Neste caso, o avanço de tempo entre os robôs ocorre da mesma forma pelo gerenciamento com a HLA, entretanto o ambiente de operação do robô real é distinto dos demais. Isso acontece porque ele não tem uma ponte e portanto não tem comunicação com o ambiente ROS.



(a) Ambiente HLA (b) Ambiente ROS

Figura 4.10: Teste de envio de dados do ROS ao HLA

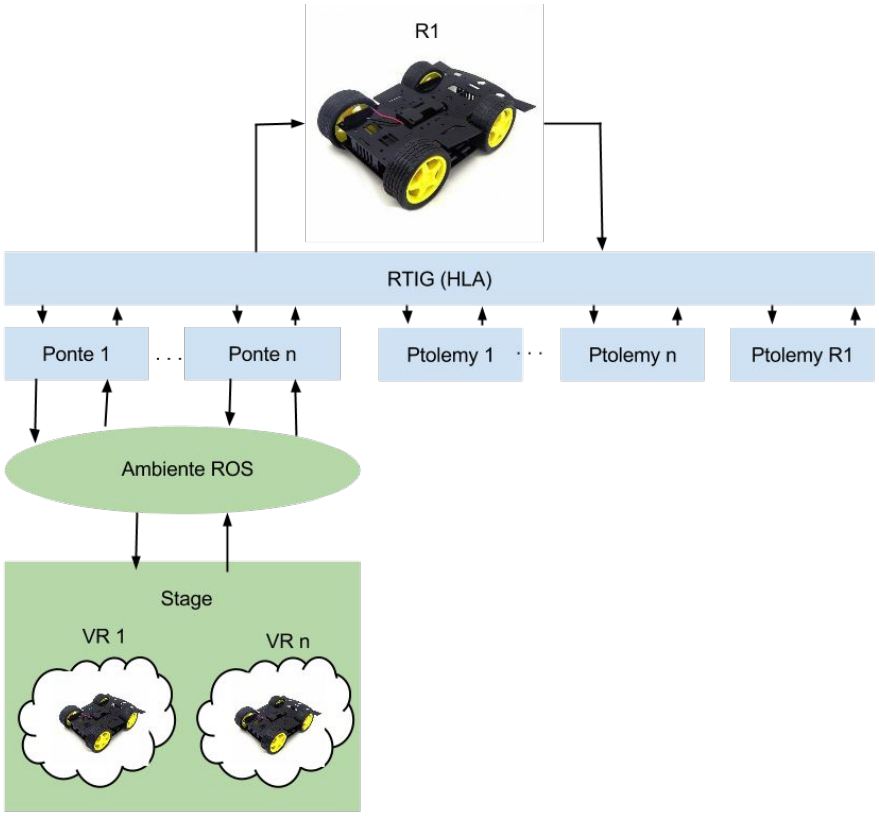


Figura 4.11: Abordagem RiL com multi-robôs simulados

É possível ainda, utilizar diferentes tipos de controlos para os robôs por meio do ROS.

Isso acontece quando um nó do ROS é utilizado para controlar um robô. Dessa forma podemos ter diversos robôs no simulador Stage, sendo parte deles controlados pelo simulador Ptolemy (VRx) e outros controlados pelos algorimos de controlo do ambiente do ROS (VOx), que também podem ser utilizados para robôs reais compatíveis com o Sistema Operacional Robótico (figura 4.12). Quando isso acontece, temos uma cossimulação sincronizada em relação às instâncias do Ptolemy que utilizam o HLA e os robôs que são controlados diretamente pelo ROS que não têm controlo no seu avanço de tempo por parte da HLA.

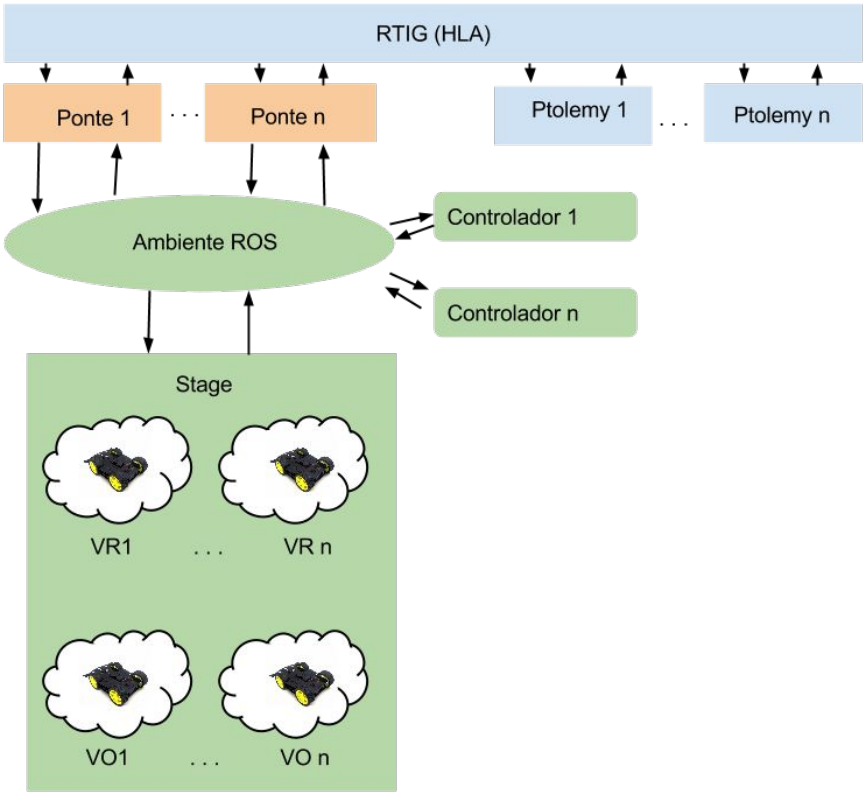


Figura 4.12: Abordagem multi-robôs simulados com controlo no ROS e no Ptolemy

Para integrar Robôs ao *loop* de simulação no ambiente dos robôs com HLA será utilizado um robô espelho, isto é, um robô virtual que vai espelhar o comportamento de um robô real no ambiente de operação simulado e vice-versa. Isso permite que o ambiente de simulação possa ser comparado com o ambiente real. Nesta abordagem os controles conectados ao ROS possuem uma dinâmica própria, sendo independentes entre si. Já os robôs virtuais controlados pelo Ptolemy continuam sincronizados, avançando igualmente no tempo de simulação por meio da administração de tempo da HLA.

4.1.3 Controle virtual

Nos experimentos realizados neste trabalho o termo controle virtual é utilizado no sentido de evidenciar que o robô tem seu controle simulado. Ou seja, a parte física do robô é realizada no simulador Stage enquanto que o controle do robô é feita no simulador Ptolemy. A figura 4.13 ilustra como o robô simulado tem seu controle no ambiente do Ptolemy.

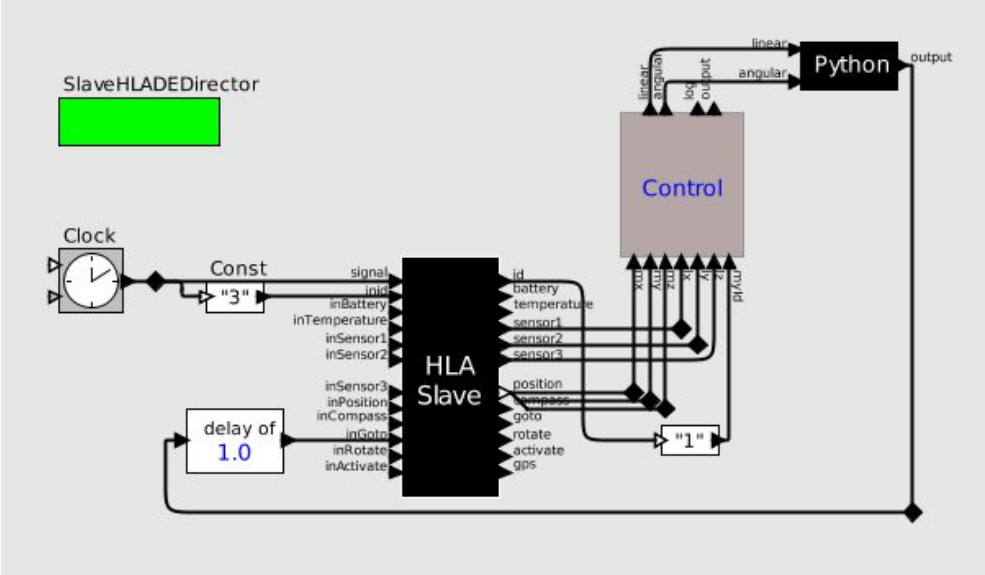


Figura 4.13: Modelo que controla o robô simulado por meio da HLA

Como foi visto, o componente HLA é responsável por receber as informações da federação. Na imagem os dados recebidos são os valores de odometria do robô lider e do robô simulado. Ao passar pelo algorimo de controle os dados de saída são as velocidades angular e linear que devem ser enviadas à ponte e em seguida publicadas em um tópico ¨cmd_vel¨ no ambiente ROS. Isso faz com que o robô se mova no simulador ou no caso de um robô real, o robô se moveria.

O componenente controlo foi criado a partir de um ator do Ptolemy chamado *Python Actor*. Esse ator permite que programas em Python sejam executados no ambiente do Ptolemy, de forma que a cada iteração do simulador uma função do *Python Actor* chamada *fire* é invocada para tratar os dados de entrada do ator e dessa forma enviar à porta de saída a informação processada.

Como este robô tem a implementação do algoritmo de controlo inserida no Ptolemy e consequentemente no ambiente da Arquitetura de Alto Nível, têm sua frequência sincroni-

zada com os demais federados. Ou seja, se um federado demora mais tempo a executar o algoritmo de controlo, os demais deverão esperar que este termine para em seguida avançar igualmente no tempo de simulação.

4.1.4 Controle real

O controle real apresentado neste trabalho, é um robô controlado por uma aplicação Python executando em ambiente ROS. O algoritmo utilizado neste robô é o mesmo algoritmo utilizado no Ptolemy entretanto com implementações diferentes visto que aqui a aplicação pode ser desenvolvida com mais liberdade que em atores do Ptolemy, que necessitam ter uma estrutura minima especificada para funcionar no ambiente.

O robô real, não tem sincronia com o ambiente do HLA. Dessa forma, a dinâmica deste robô não recebe interferência dos demais robôs que são simulados por meio do componente Ponte.

4.1.5 Utilização de um robô espelho

O robô espelho está incluído na federação, portanto tem sua dinâmica dependente dos demais federados. Entretanto este robô, ao invês de receber os valores de seus sensores, recebe os valores do robô real[1]. Assim, todos os ciclos de controle ambos os robôs, o real/simulado e o espelho, partem dos mesmos valores sensoriais e calculam as saídas correspondentes. Desta forma, ambos deverão ter movimentos semelhantes. Todavia, por diversos motivos como o atrito do robô com o solo, a modelagem do robô e fatores de calibragem, esta movimentação pode não ser tão semelhante. Por isso seria interessante que ao invés de aplicar a mesma saída do robô real a este robô espelho, a criação de um robô que recebesse a posição do robô real e automaticamente se movesse para a mesma no ambiente Stage. Mudar a posição do robô no Stage em tempo de execução ainda não é um serviço provido pelo simulador, a não ser para tornar os robôs a posição inicial da simulação. Desta forma seria necessário modificar o código do simulador criando um novo serviço que viabilizasse este requisito mas não foi possível devido ao tempo de desenvolvimento do presente trabalho. Esta forma

[1] Contudo, é possível usar uma fusão dos sensores locais com os sensores do robô real/simulado e projetar essa fusão no ambiente de operação real. Este é o caso quando se pretende refletir no ambiente real interações que ocorrem no ambiente virtual, como é o caso de obstáculos virtuais.

de operação de um robô espelho pode ser muito valiosa, por exemplo, para detectar falhas abruptas no robô real. Quando tal acontece, os movimentos dos dois robôs não vão coincidir, o que pode ser detectado e utilizado para sinalizar a falha.

Com o ambiente em funcionamento, torna-se possível realizar experimentos em que equipes de robôs podem ser simuladas de forma que robôs reais sejam inseridos no ambiente de simulação. Isso permite inserir robôs reais em ambientes virtuais ou ainda realizar testes de algoritmos de equipes de robôs em que parte dos robôs utilizados na simulação são robôs reais e outra parte são robôs simulados. No fundo, cada robô real deverá ter um robô espelho virtual no ambiente de operação simulado que lhe permita interagir com outras entidades simuladas presentes nesse ambiente. Usando terminologia dos ambientes virtuais cada robô espelho é um avatar.

Capítulo 5

Experimentos de sincronização

Este capítulo apresenta os experimentos realizados para verificar aspectos de sincronização no ambiente proposto, envolvendo equipes de robôs onde são utilizados um robô real e vários robôs virtuais, um dos quais é um espelho do robô real e outro é designado líder porque determina a formação (os outros posicionam-se relativamente a este). Foram realizados dois conjuntos de experimentos. O primeiro conjunto está relacionado a HLA enquanto que o segundo conjunto se refere a aspetos de detecção de falhas de *hardware* usando a técnica de robô espelho virtual.

5.1 Experimentos

Os experimentos realizados têm a finalidade de estudar aspectos de sincronização dos robôs em diferentes cenários e assim verificar o comportamento obtido. Um dos aspectos que mudam de cenário em cenário nas simulações é distribuição. Vamos considerar simulações distribuídas e centralizadas. As simulações centralizadas são realizadas de forma que todos os processos que envolvem a simulação são executados na mesma máquina, enquanto a simulação distribuída utiliza duas máquinas.

Os processos utilizados pelo ambiente proposto são descritos a seguir:

- RTIG - *Runtime Infraestructure Gateway*: Processo responsável por prover os serviços da Arquitetura de Alto Nível aos federados. Este processo foi utilizado em todos os experimentos que utilizavam a HLA.

- ROS core - Núcleo do ROS, responsável por configurar a comunicação entre os nós do ROS.
- Instâncias do Ptolemy - Instâncias do simulador Ptolemy com a implementação do algoritmo de controle em Python. Estas instâncias utilizam a ponte para se comunicar com o ROS.
- Simulador Stage - Simula os diversos robôs das simulações.
- Pontes - Responsáveis por realizar a comunicação entre o ambiente ROS e Ptolemy.
- Monitor - Esta aplicação foi desenvolvida para monitorar as informações compartilhadas no ROS. Após receber os dados dos robôs, cria um log que é salvo em arquivo.

As duas máquinas que foram utilizadas nas simulações em que o ambiente era distribuído eram a Máquina A (Processador Intel Core i5, 4 GigaBytes de memória RAM, linux Ubuntu 14.04) e a Máquina B (Processador Intel Core 2 Quad 2.66 GHz, 4 GigaBytes de memória RAM, linux Ubuntu 14.04). Os processos executados na máquina A eram: o RTIG, o ROS core e Stage. Já a máquina B : o monitor e as pontes.

Um algoritmo de controle foi desenvolvido na linguagem Python para controlar os robôs que estavam no ambiente ROS, neste caso o robô real. Entretanto, após algumas modificações e integração com atores do Ptolemy, pôde também ser utilizado por este simulador para o controle dos robôs virtuais. Desta forma, garantimos que o controlador é o mesmo no robô real e nos virtuais. O algoritmo recebe como entrada os valores de odometria dos sensores dos robôs e tem como saída a variação linear e angular que deve ser aplicada aos robôs para chegarem ao destino. No caso do robô líder, o destino é se dirigir a uma posição X,Y específica, enquanto que os demais robôs vão entrar em formação com o robô líder.

A disposição dos robôs no ambiente Stage se deu da mesma forma em todas as simulações e cenários. Assim, a posição incial é mostrada pela figura 5.1 . São utilizados quatro robôs, todos podem ser controlados tanto pelo controlador funcionando no ROS como no Ptolemy, a disposição dos robôs no mapa se dá com as seguintes coordenadas X, Y: Robô cinza (10 , -15); robô Vermelho (-10, 10); robô verde (10, -10); e robô negro(-10 , -10).

Apresentadas as configurações das simulações, seguem agora as descrições mais detalhadas dos experimentos e os resultados obtidos.

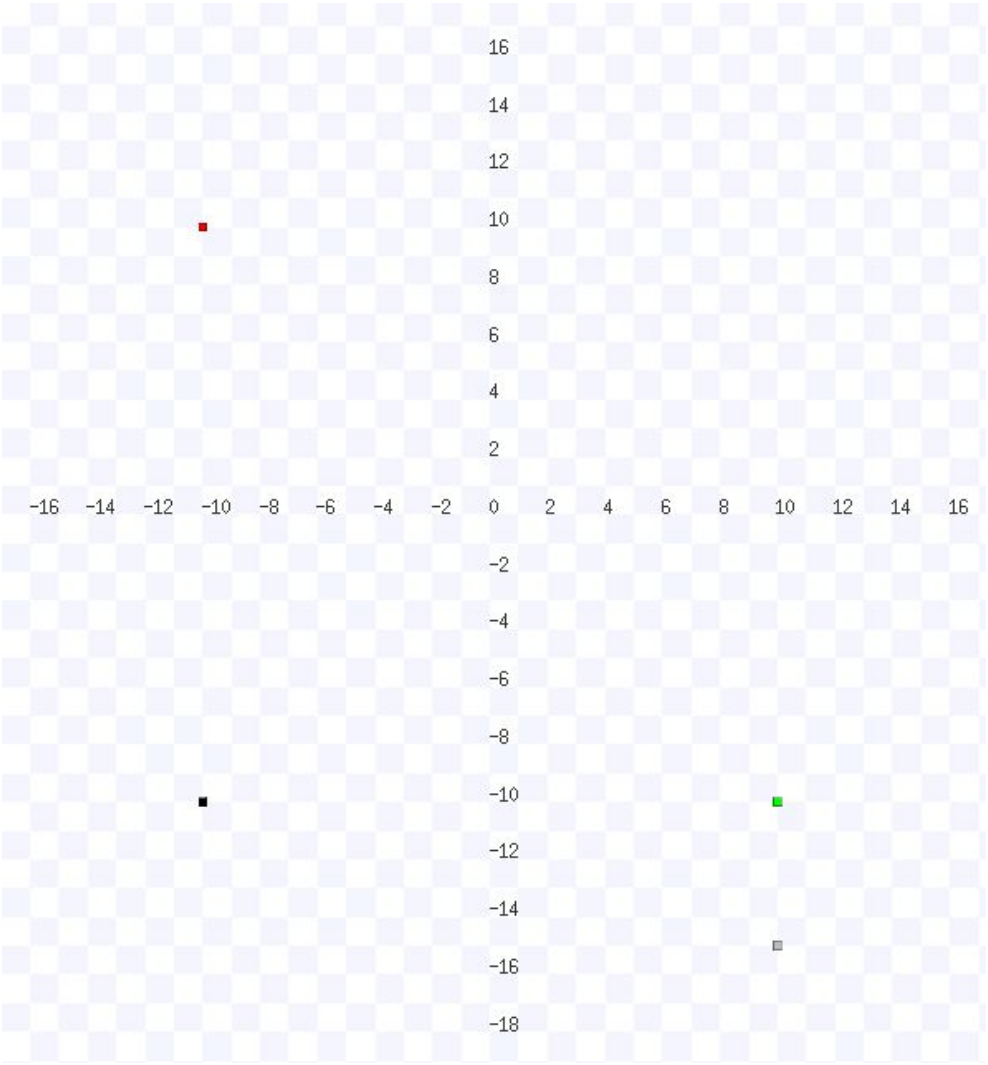


Figura 5.1: Disposição dos robôs no simulador Stage

5.2 Experimentos utilizando o HLA

Este experimento consistiu na realização de simulações em que robôs no simulador Stage eram controlados por algoritmos implementados no simulador Ptolemy, ou seja, eram todos robôs virtuais sincronizados pelo HLA. A comunicação entre os ambientes era realizada utilizando o componente ponte.

A primeira simulação foi realizada na máquina B, diversos robôs saem de pontos distintos do mapa para entrar em formação seguindo o robô líder. A trajetória dos robôs pode ser vista na figura 5.2.

Além dos dados para gerar o gráfico da trajetória, ao fim da simulação o componente ponte apresentava a frequência que tinha operado. Neste primeiro experimento, todos os nós

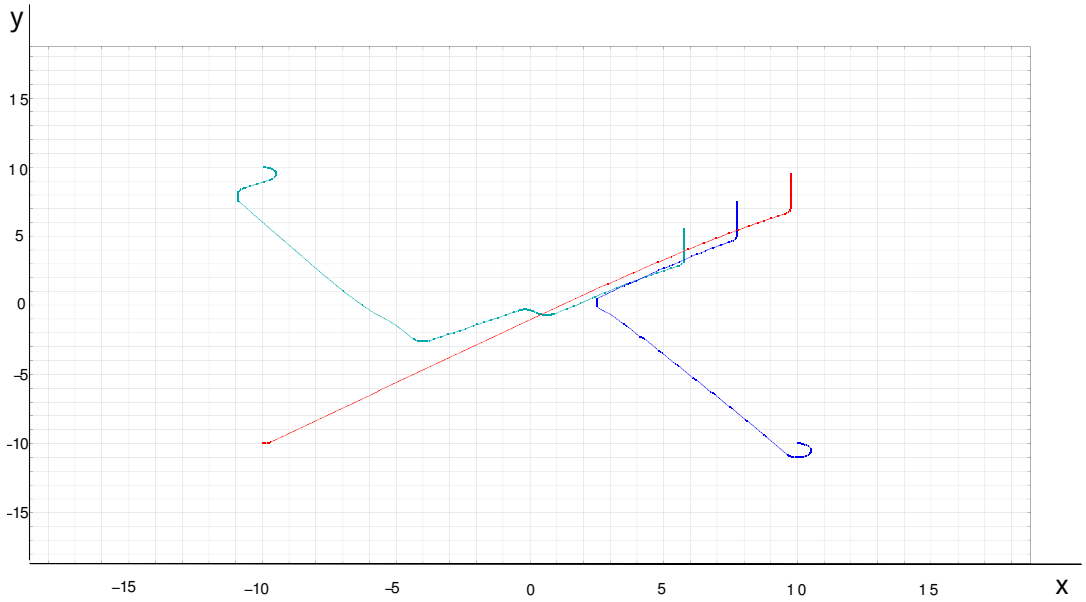


Figura 5.2: Experimento utilizando HLA na máquina B

ponte executaram a uma frequência de 143 Hz.

A mesma simulação realizada na máquina B foi realizada na máquina A, isso vai permitir constatar se a utilização de máquinas heterogêneas pode afetar o comportamento dos robôs. A trajetória resultante desse segundo cenário é apresentada na figura 5.3.

Apesar de ter algumas pequenas diferenças entre as trajetórias apresentadas, não é perceptivel nenhuma variação mais abrupta entre as trajetórias feitas pelos robôs do primeiro cenário e do segundo cenário. Em relação a frequência, as pontes executaram a uma frequência de 236Hz. Estes 2 experimentos mostram que o HLA mantém os simuladores sincronizados apesar das diferentes velocidades de execução, razão pela qual as trajetórias são semelhantes. É possível notar que pequenas diferenças são introduzidas pelo Stage como erros de odometria que poderão variar entre cenários.

Um outro cenário foi utilizado para verificar o comportamento dos robôs quando outras aplicações são executadas enquanto a simulação está ocorrendo. Para isso, foi utilizado um navegador com diversas guias abertas, cada uma executando um vídeo a partir da Internet. Os resultados dessa simulação são mostrados na figura 5.4.

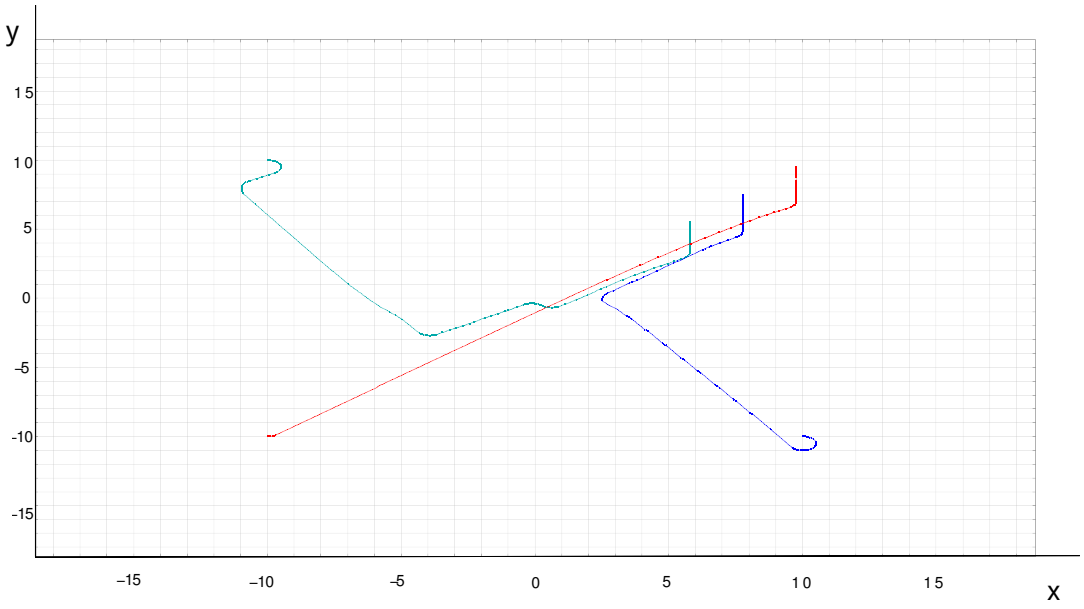


Figura 5.3: Experimento utilizando HLA na máquina A

Neste experimento já verificam-se algumas modificações na trajetória dos robôs, mas são pequenas e a frequência que a ponte operava caiu para 43 Hz quando comparada aos 143 Hz de quando não haviam outras aplicações sendo executadas durante a simulação. Estas diferenças que não ocorriam antes poderão ser devidas a interferência das outras aplicações com os componentes da cossimulação, o que poderá gerar sobrecargas e perda de informação (Ex.: perda de ciclos de controle).

O quarto cenário de simulação que é apresentado utilizou duas máquinas para efetuar a simulação. A trajetória é apresentada na Figura 5.5.

Esta simulação permite verificar que o uso da rede também não influenciou a trajetória dos robôs de forma que modificasse as trajetórias abruptamente. Entretanto os pontes obtiveram frequências de 672 Hz, ou seja, a uma velocidade mais rápida. Isto mostra que o HLA foi capaz de manter os simuladores sincronizados mesmo operando em duas máquinas.

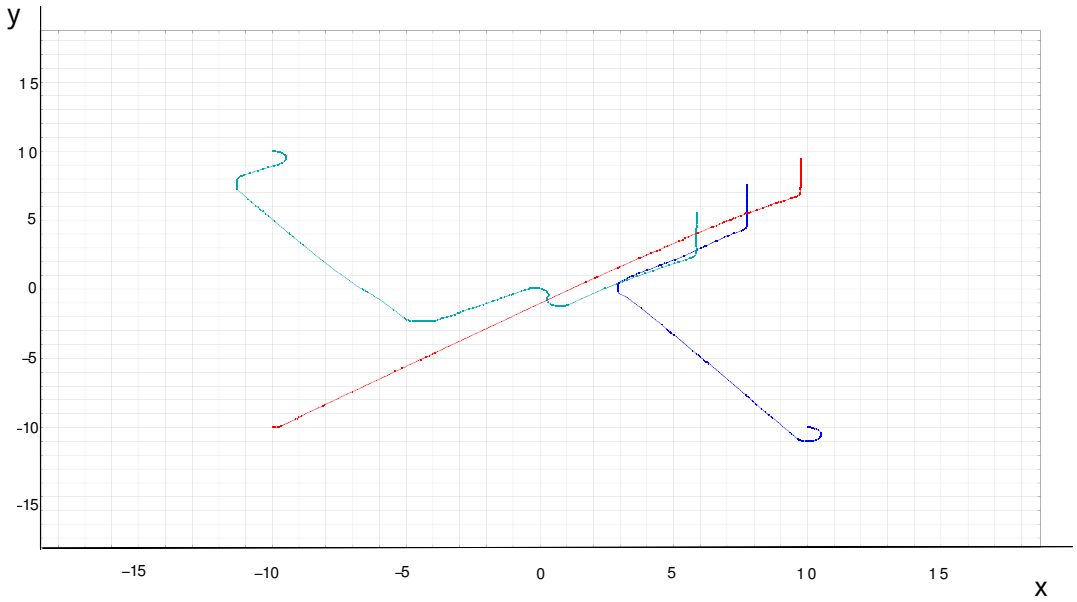


Figura 5.4: Experimento utilizando outras aplicações durante a simulação

5.3 Experimentos com robô Espelho

A utilização do robô espelhado pode ser interessante para a deteção de falhas. Isso se torna possível a partir da análise do caminho percorrido pelo robô real e robô virtual espelho, respectivo. Em um momento que ocorra uma diferença muito abrupta entre a trajetória desses ambientes, isso pode indicar falhas de *hardware*, como uma roda quebrada por exemplo.

Os experimentos realizados demonstram como o ambiente proposto neste trabalho pode ajudar a detectar falhas de *hardware*. Nos experimentos será simulada uma roda quebrada no robô real, isso fará com que o robô começe a andar em circulos seguindo assim uma trajetória diferente do robô virtual e será então detectada uma falha.

Dois conjuntos de experimentos foram realizados neste etapa do trabalho. O primeiro para realizar medições de tempo em relação ao uso da HLA e outro com foco na detecção de falhas de hardware.

Os experimentos desenvolvidos com o robô espelho foram divididos em duas partes, a primeira envolve os experimentos realizados para verificar o tempo que uma mensagem durava para sair do ambiente ROS ir ao HLA e voltar ao Ptolemy.

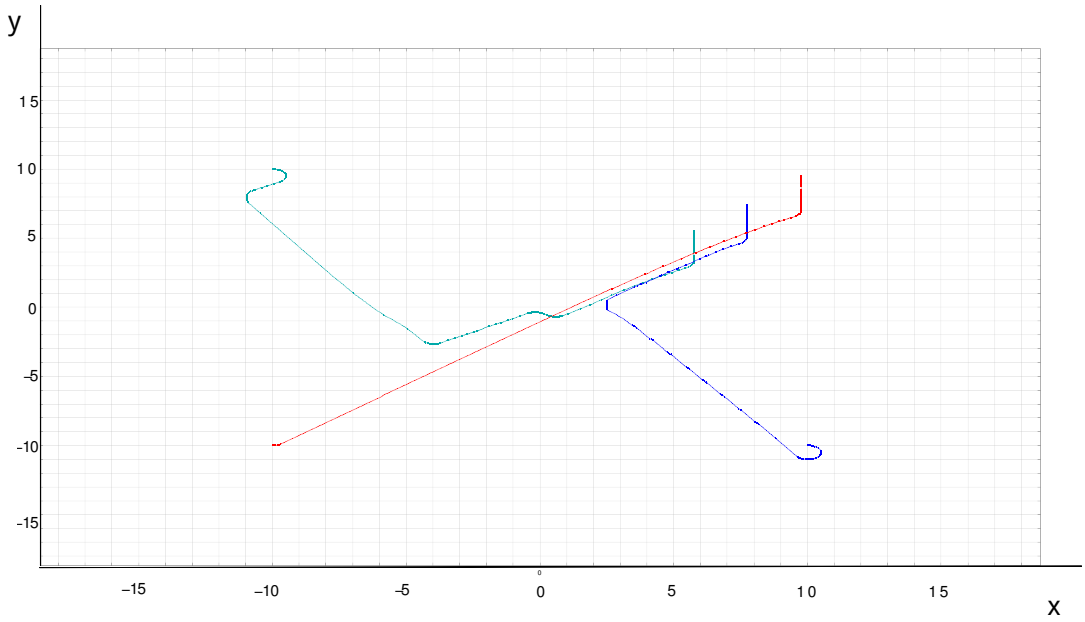


Figura 5.5: Experimento utilizando duas máquinas

5.3.1 Medidas do tempo de execução

Este primeiro conjunto de simulações busca realizar uma análise de tempo e verificar o tempo de resposta do HLA em relação as mensagens enviadas ao algoritmo de controle contido no simulador Ptolemy. Isso permite estipular a frequência que o ROS pode enviar dados ao robô sem prejudicar o funcionamento da HLA. No fundo, estamos agora obrigando que o tempo de execução da simulação (e por conseguinte o tempo de simulação) seja menor do que o tempo de execução do controle do robô real.

O primeiro experimento foi realizado utilizando um controle real com uma frequência de 1 Hz. A figura 5.6 mostra o tempo gasto para uma mensagem enviada pela ponte chegar ao Ptolemy, ser processada e voltar a ponte como resposta, ou seja, o tempo de execução do ciclo de controle dos robôs virtuais.

O diagrama de frequência apresentado na figura 5.7 foi formado com os dados da simulação de 1Hz. A média e o desvio padrão foram 8.093ms e 3.827ms respectivamente, enquanto que a moda e mediana foram 5ms e 7ms respectivamente. A curva apresentada no gráfico apresenta o comportamento de uma distribuição assimétrica positiva, entretanto este

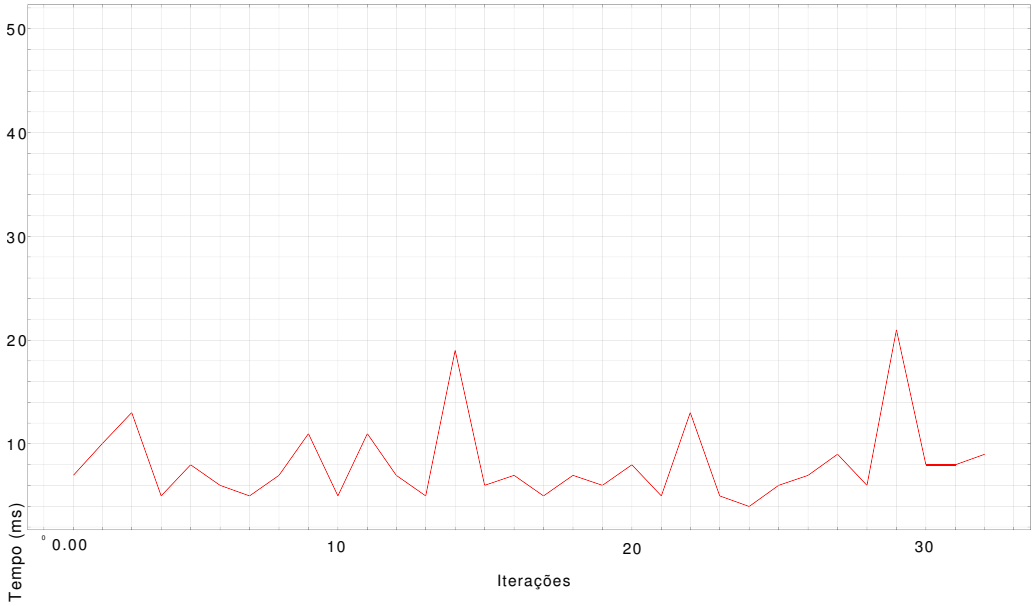


Figura 5.6: Tempo de resposta durante simulação com frequência 1 Hz

experimento não tem grande representatividade porque a quantidade de amostras é pequena.

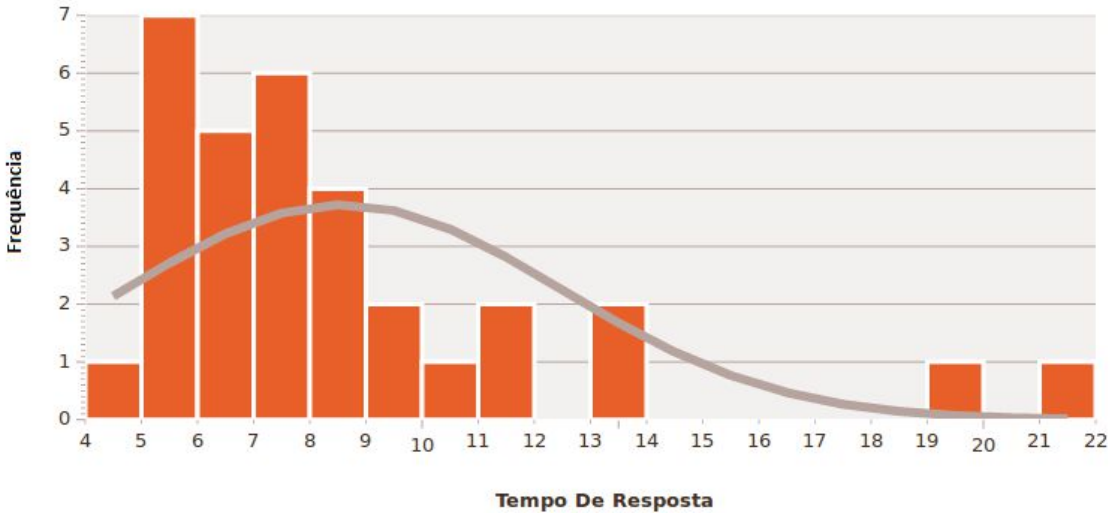


Figura 5.7: Histograma do experimento com 1Hz

O segundo experimento utilizou a frequência de 30 Hz na simulação, o gráfico gerado com os dados da simulação pode ser visualizado na figura 5.8.

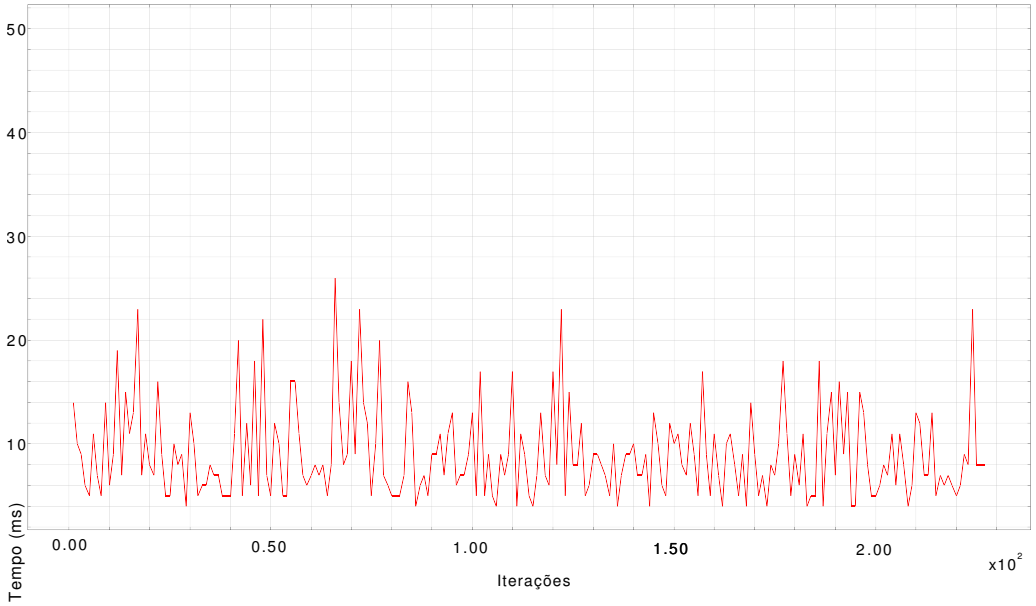


Figura 5.8: Tempo de resposta durante simulação com frequência 30 Hz

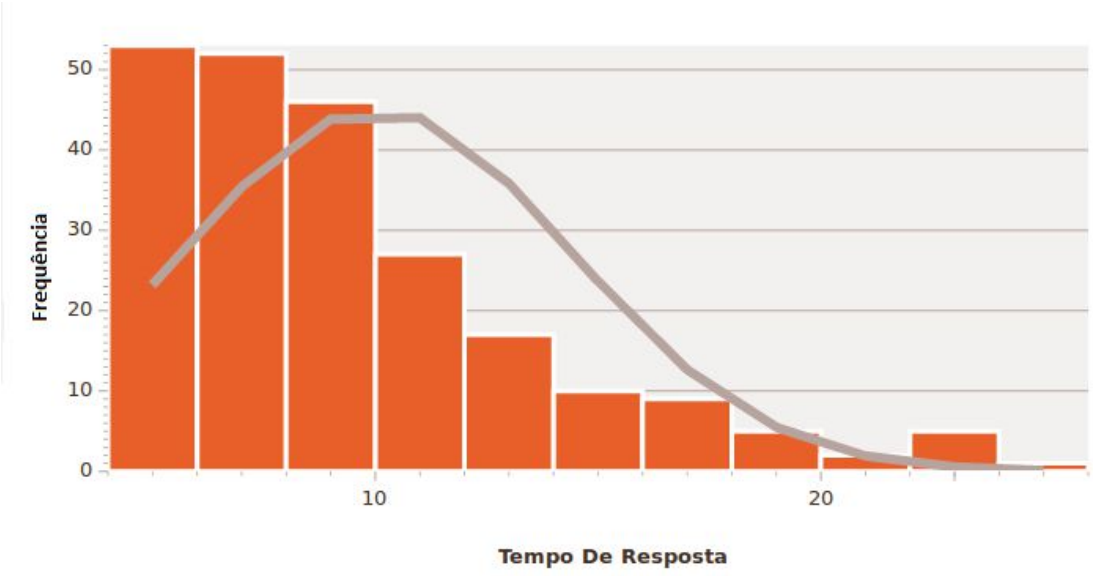


Figura 5.9: Histograma do experimento com 30Hz

A figura 5.9 apresenta o diagrama de frequência obtido com os dados de tempo de resposta do experimento com 30 Hz. Também foi possível obter os valores da média, mediana,

Tabela 5.1: Medidas do tempo de execução (ms)

	Moda	Média	Desvio Padrão	Mediana
Experimento com 1 Hz	5	8,093	3,827	7
Experimento com 30 Hz	5	9,039	4,361	8

moda e desvio padrão que são respectivamente: 9.039 ms, 8 ms, 5 ms e 4.361 ms. A curva no diagrama apresenta uma distribuição assimétrica positiva assim como no experimento de 30 Hz. Este experimento entretanto possui uma amostra melhor que o experimento anterior, tendo assim mais representatividade. Para apresentar os valores obtidos de forma mais clara, os dados são apresentados na tabela 5.1.

Estes experimentos mostram que frequências de controle mais elevadas geram maior variabilidade do tempo de execução do ciclo de controle mas o tempo de resposta da ponte em relação ao Ptolemy varia entre 8.093 ms e 9.039 ms para uma gama de frequências relativamente vasta. Entretanto, considerando o tempo de resposta quando a mensagem demora mais tempo, temos um tempo de resposta de 26 ms, obtido na simulação de 30 Hz. Dessa forma uma simulação em que o ROS envie dados ao Ptolemy em um intervalo menor que 26 ms (frequência de 38Hz), poderá fazer com que diversas mensagens fiquem acumuladas esperando para serem processadas pelo Ptolemy. Podendo assim comprometer o resultado da simulação.

5.3.2 Simulação com diversas frequências

Dois experimentos foram realizados para verificar a possibilidade de encontrar divergências no algoritmo virtual e real, um de forma distribuída enquanto que outro foi realizado de forma centralizada. Uma modificação foi realizada no processo *monitor*, esta mudança fez com que além de ser responsável por receber os dados de posição para análise posterior, também se tornou reponsável por detectar divergências entre o robô espelho e o robô real. Ao encontrar uma divergência abrupta nessas trajetórias, finaliza as pontes responsáveis pelo envio de mensagens ao Ptolemy bem como mostra uma mensagem ao usuário informando que foi encontrada uma divergência. Uma segunda mudança necessária para estes experimentos foi que a partir de uma dada quantidade de iterações, o controle do robô real passa a enviar comandos que farão o robô andar em circulos (simulando a falha de hardware). Este

número de iterações varia de acordo com a frequência utilizada no experimento.

5.3.3 Simulando localmente

Este experimento consistiu na utilização de algoritmos de controle em ambientes diferentes (Ptolemy e ROS), serem utilizados com os mesmos dados de entrada e aplicando a saída a dois robôs diferentes para assim verificar o erro entre os trajetos desses robôs. O experimento foi realizado localmente e utilizou duas frequências diferentes, sendo elas 1Hz, 30Hz.

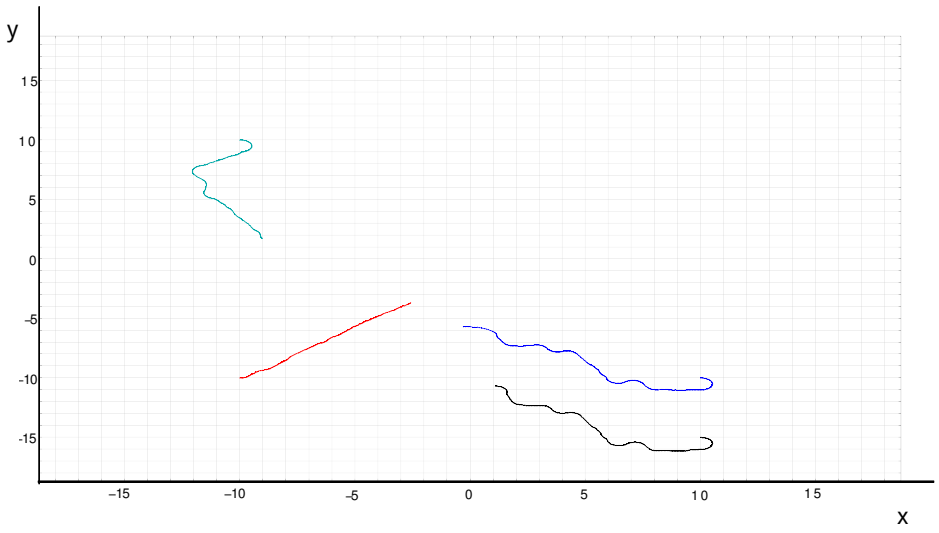


Figura 5.10: Cenário com 1Hz: Caminho dos robôs

A figura 5.10 Apresenta as trajetórias dos robôs obtidas em uma simulação de 1Hz, nela percebemos que a trajetória do robô espelho (em preto) ocorria de forma similar a do robô real (trajetória azul). Entretanto em um dado momento o robô azul divergiu e começou a fazer uma trajetória circular. O circulo não ficou aparente na figura porque antes de poder completar o circulo o algoritmo monitor detectou a variação abrupta de posição (apresentada na figura 5.11), encerrou as instâncias do controle que estavam em execução mostrando uma janela no computador informando da divergência entre os robôs.

Em seguida foi realizada a simulação para o segundo cenário, onde todos os nós utilizavam uma frequência de 30 Hz. A partir da informação obtida pelo monitor, temos as figuras

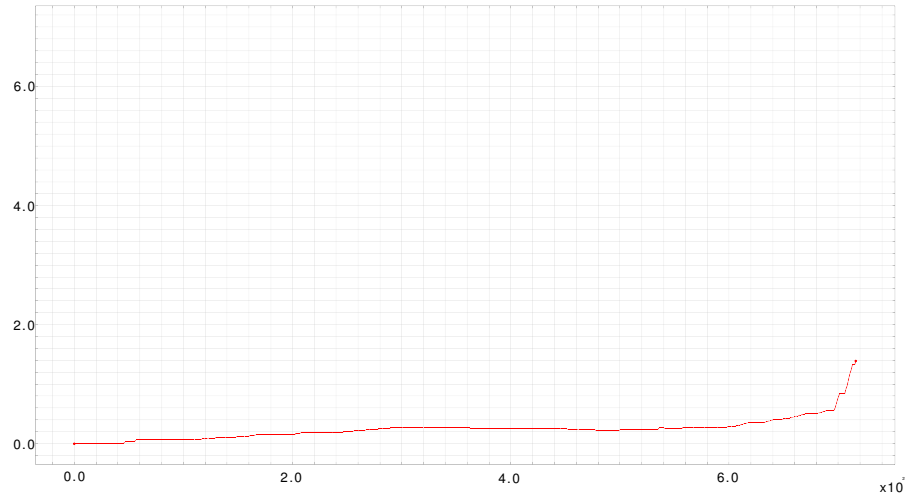


Figura 5.11: Cenário com 1Hz: Distância entre robô espelho e robô real

5.12 e 5.13 que apresentam respectivamente as trajetórias dos robôs e a variação de posição entre os dois robôs, o robô real e seu espelho.

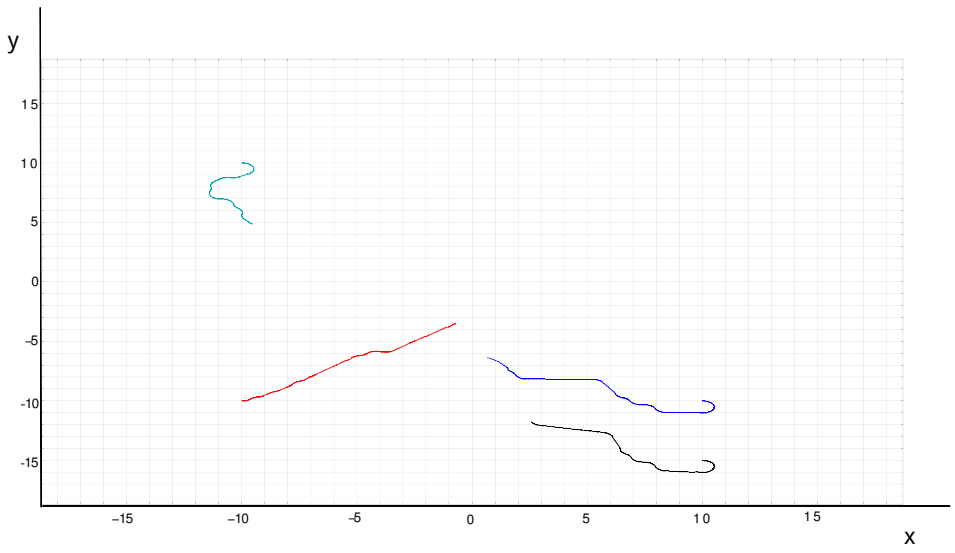


Figura 5.12: Cenário com 30Hz: Caminho dos robôs

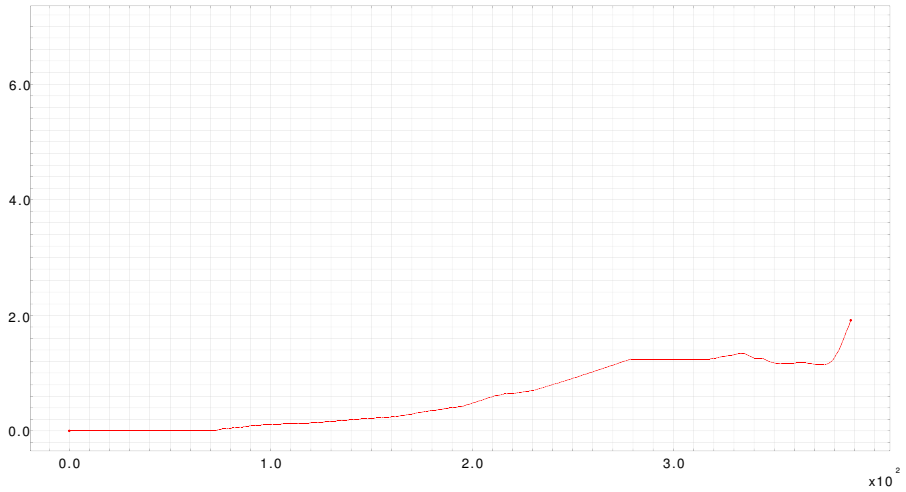


Figura 5.13: cenário com 30 Hz: Distância entre robô espelho e robô real

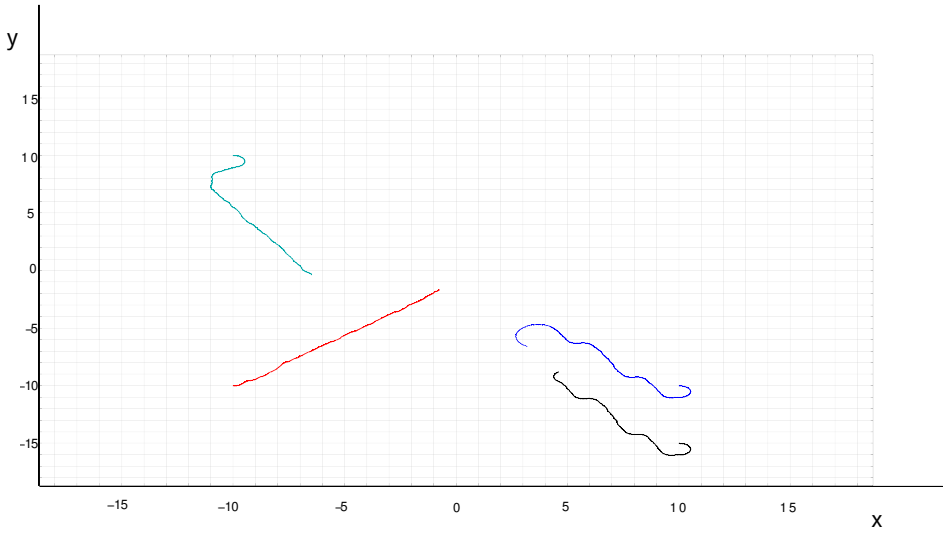
As trajetórias no gráfico mostram que em dado momento o robô real começou a agir de forma muito distinta em relação a seu espelho, que recebia as mesmas entradas. Então o algoritmo monitor detectou esta variação de trajetoria e encerrou a simulação. A figura 5.13 que apresenta a variação da distância de posições entre os robôs espelho e real no gráfico é possível perceber o momento em que houve um acréscimo abrupto na distância entre os robôs, isso permitiu ao código monitor identificar a simulação de falha.

5.3.4 Simulação distribuída

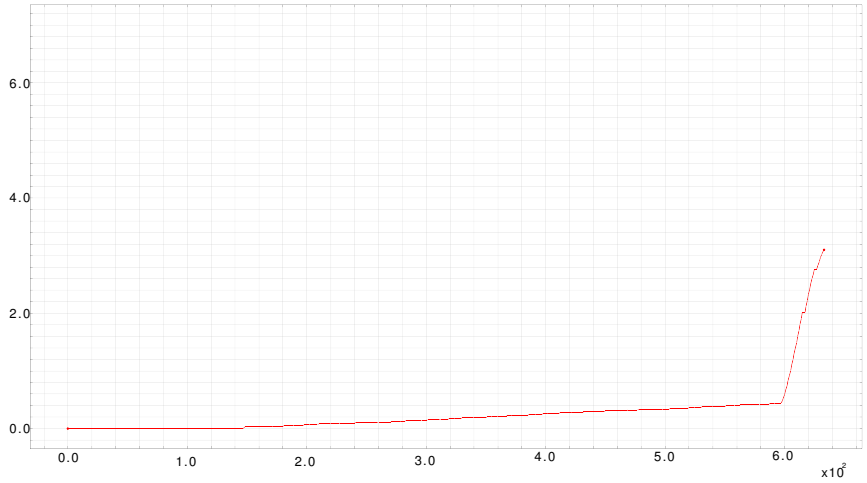
O experimento II consiste na realização de simulações para verificar o comportamento dos robôs em um ambiente distribuído. O experimento também utilizou dois cenários onde variam as frequências dos robôs em 1Hz e 30 Hz.

O primeiro cenário utilizou 1 Hz de frequência, a figura 5.14 apresenta os dados obtidos.

Na figura 5.14(a) percebemos a tendência do robô azul em realizar um circulo enquanto que o robô espelho continua sua trajetória. A distância entre a trajetória dos robôs simulado e real apresentada na figura 5.14(b) permite verificar uma variação constante até um momento em que a variação ocorre de forma abrupta ativando a deteção de divergencias entre os robôs e encerrando a simulação. Por simplicidade, nestes experimentos fizemos a deteção da vari-



(a) Caminho dos robôs

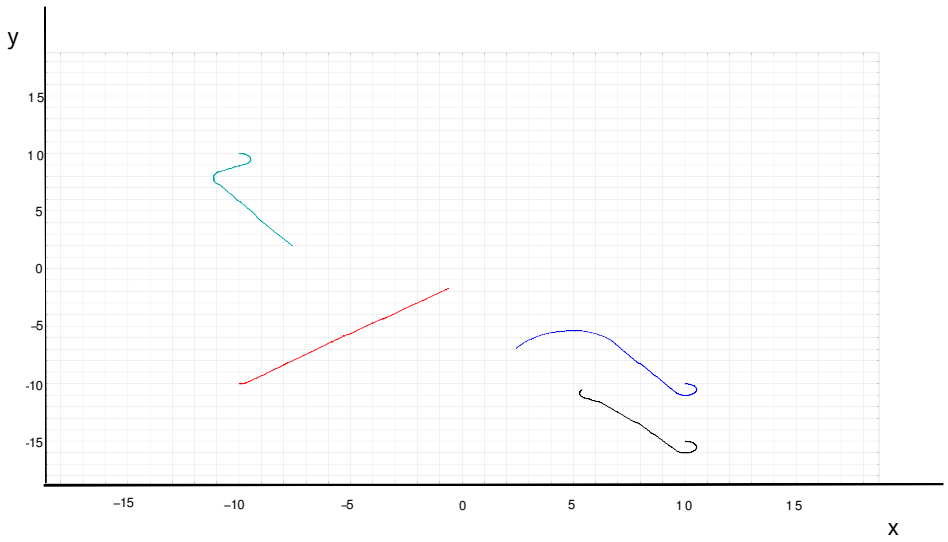


(b) Distância entre robô espelho e robô real/simulado

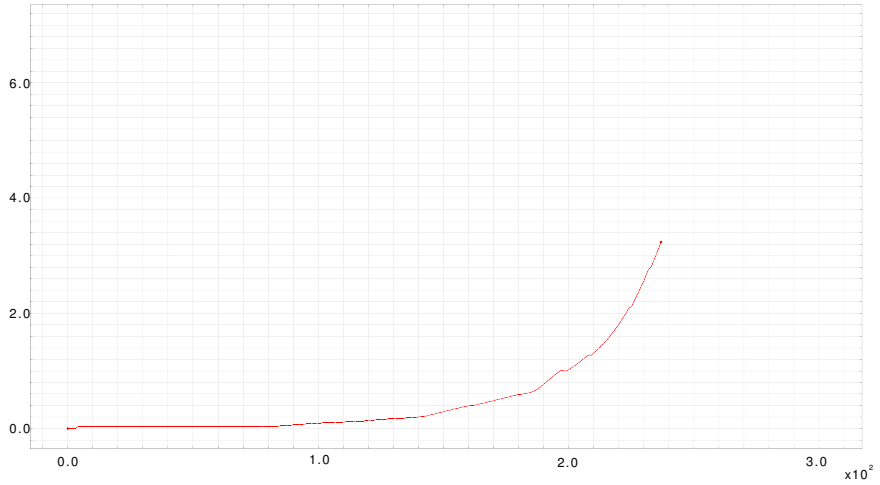
Figura 5.14: Experimento II: cenário com 1 Hz de frequência

ação por limiar do erro de posição relativa entre os robôs real e virtual. Contudo, para uma deteção robusta, poderia ser utilizado um algoritmo de Inteligência Artificial ou um filtro para reduzir ou aumentar este erro.

O segundo cenário utilizou 30 Hz como frequência. A figura 5.15(a) apresenta a trajetória dos robôs enquanto que a figura 5.15(b) apresenta a distância entre os robôs.



(a) Caminho dos robôs



iterações x distância

(b) Distância entre robô espelho e robô real/simulado

Figura 5.15: Experimento II: cenário com 30 Hz de frequência

Embora neste cenário já haja um pouco mais de erro de odometria, é possível perceber o momento em que o robô real começa a fazer uma manobra simulando uma falha de *hardware* a partir do seu movimento circular. Ao realizar esta ação, a distância entre a posição que o robô real aumenta abruptamente, ativando o algoritmo de deteção de divergências. A figura 5.15(b) apresenta esta a diferença entre a posição que o robo deveria estar e a posição que

ele se encontra.

Nos dois experimentos de deteção de erro foi possível identificar os momentos em que ocorreram variações abruptas das distâncias entre as posições que o robô se encontrava e onde ele deveria estar.

Capítulo 6

Experimentos com Turtlebot

Este capítullo apresenta os experimentos realizados para verificar o comportamento do ambiente proposto utilizando robôs reais. Este experimento envolve a utilização do Turtlebot e dos simuladores Stage e Ptolemy. Foram realizados três experimentos, o primeiro para verificar a trajetória do robô ao detectar um obstáculo virtual no ambiente de cossimulação, o segundo para simular uma falha de *hardware* e o terceiro para mitigar a falha de *hardware* por meio de modificações no controle do robô. Todos os experimentos foram realizados em uma mesma máquina que se encontrava no próprio turtlebot. Os processos utilizados pelo ambiente proposto são descritos a seguir:

- RTIG - *Runtime Infraestructure Gateway*: Processo responsável por prover os serviços da Arquitetura de Alto Nível aos federados.
- ROS core - Núcleo do ROS, responsável por configurar a comunicação entre os nós do ROS.
- Instâncias do Ptolemy - Instâncias do simulador Ptolemy com a implementação do algoritmo de controle em Python. Estas instâncias utilizam a ponte para se comunicar com o ROS.
- Simulador Stage - Simula os diversos robôs das simulações.
- Pontes - Responsáveis por realizar a comunicação entre o ambiente ROS e Ptolemy.
- Monitor - Aplicação utilizada para monitorar as informações compartilhadas no ROS.

- Script de comunicação do ROS com o Turtlebot- Este script é carregado a partir do comando Roslaunch do ROS e permite ao ROS comunicação com os sensores e atuadores do turltebot.

A máquina utilizada nas simulações tinha como configuração: um processador Intel Core i5, 4 GigaBytes de memória RAM, linux Ubuntu 14.04.

6.0.5 Integração do Turltebot com ambiente

O objetivo deste experimento é utilizar um turtlebot real em um ambiente virtual, inclusive com sensores virtuais. Este cenário se faz interessante quando o sensor do robô é muito caro para ser adquirido, ou quando não está disponível. Nesta simulação será utilizado o sensor de distância, mas poderia ser outro sensor qualquer que esteja disponível no Simulador Stage.

O robô real e virtual espelho (que reflete as ações do robô real) estarão enviando informações dos sensores a mesma instância de controle no Simulador Ptolemy, enquanto que os dois receberão a mesma saída na forma de velocidade angular e linear que deve ser utilizada para movimentar o robô. O robô virtual permite o uso do sensor de distância, enquanto que o robô real permite a odometria do robô. Ao lado do robô real, para fins de comparação foi utilizado um robô com o controle realizado pelo ambiente ROS, isso permite uma comparação de trajetórias entre os dois robôs, simulado e real. A figura 6.1 mostra a disposição dos robôs no ambiente de simulação.

O objetivo do robô simulado é: seguir em frente, ao detectar um obstáculo (caixa em preto) desviar do obstáculo e depois seguir a trajetória. O gráfico da figura 6.2 apresenta a trajetória dos robôs durante a simulação, tendo a posição incial dos robôs como x = 0 e y = 0. A trajetória em verde é referente a odometria do robô real, percebe-se que ela acompanha o trajeto em azul que é dado pela outra instância de controle do ptolemy, seria a trajetória esperada no caso de uma simulação sem HiL. Já a trajetória em vermelho corresponde a trajetória do robô simulado recebendo os mesmos estímulos do robô real. O trajeto parece mais distinto das demais porque o atrito do modelo do turtlebot simulado no Stage não corresponde ao do robô real no piso em que a simulação ocorreu. Isso é mais perceptível nas curvas, durante a simulação o robô real demorava bastante a realizar a curva enquanto que o simulado girava rapidamente em torno do próprio eixo.

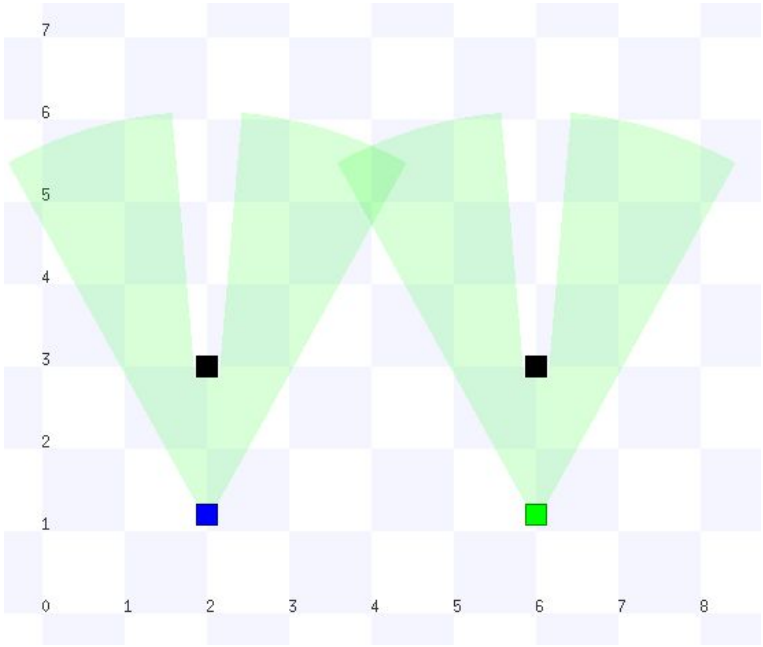


Figura 6.1: Disposição dos robôs

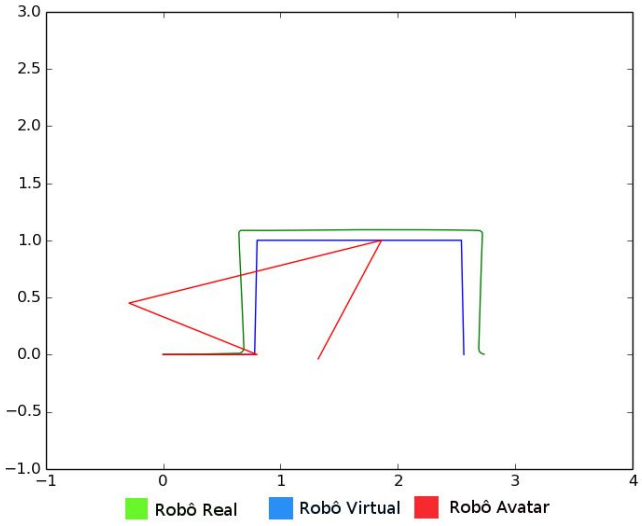


Figura 6.2: Experimento com Turtlebot

Em um segundo cenário, a roda do turtlebot real foi modificada para simular um erro de *hardware* e assim fosse possível verificar o erro durante a simulação. O gráfico com as trajetórias é apresentado na figura 6.3. Mais uma vez é possível perceber a trajetória em vermelho referente ao robô espelho distinta das demais. Já a trajetória em verde (robô real) ficou com uma angulação diferente do primeiro experimento, isso foi provocado pela diferença na roda do robô.

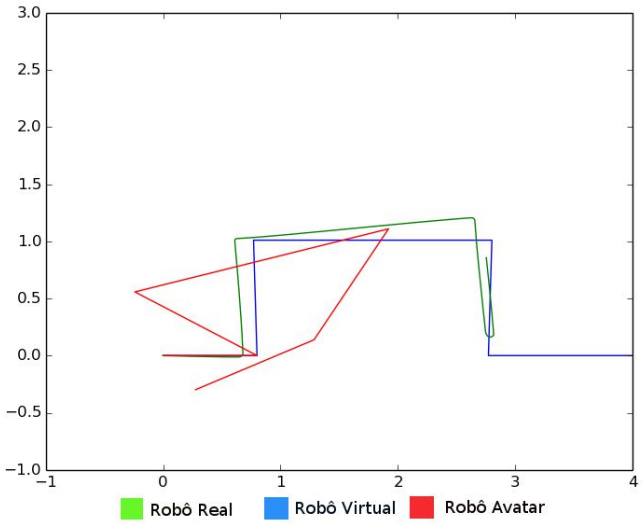


Figura 6.3: Simulação de falhar de hardware

Além do erro provocado pela roda, ao terminar de evitar o obstáculo virtual o robô real em vez de continuar o percurso a frente, mudou a orientação. Isso ocorreu porque o sensor de distância do robô virtual neste momento havia detectado novamente o obstáculo virtual, pois ao terminar a trajetória se posicionava em um local semelhante a posição que detectou o obstáculo pela primeira vez.

No terceiro cenário, permanece a simulação de erro com a modificação na roda do turtlebot. Entretanto, uma modificação foi feita no algoritmo de controle para que o robô modificasse também sua velocidade angular enquanto andava em linha reta. Isso permite a correção da trajetória para se adequar ao ângulo esperado.

Código Fonte 6.1: Código do controle

```
1 if etapa== 2 or etapa == 4 or etapa == 6:
2         oldx, oldy = posInicial
3         linear = 0.1
4         angular = 0
5         deltaS = sqrt((float(posx)-oldx)**2) + ((float(posy)-oldy)**2)
6         if (etapa != 4 and deltaS > 1) or (etapa == 4 and deltaS > 2):
7                 posInicial = None
8                 etapa= (etapa + 1) % 8
```

O trecho de código reponsável por andar em linha reta após as curvas é apresentado no código 6.1. Ele se encontra no *loop* principal do controle, nele há um if responsável por verificar se a variável etapa é igual a 2, 4 ou 6. Esta variável representa o estado do robô.

Nestes três estados o robô anda a frente a partir da modificação da velocidade linear do robô. Em seguida é calculada a distância do robô no instante atual (variáveis posx e posy)e no instante em que foi inciada a trajetória (variáveis oldx e oldy), no caso do estado ser 2 ou 6 e a distância > 1 o percurso em linha reta é finalizado. No caso do estado ser 4, e etapa é finalizada apenas se distância > 2, isso porque este trajeto requer que o robô ande mais para ultrapassar o obstáculo. Em seguida a posição inicial recebe nulo e o estado atual é incrementado, no caso do estado (etapa) ser maior ou igual a 8 o valor é zerado. Isso acontece porque este algoritmo possui apenas 8 estados.

Código Fonte 6.2: Código do controle após modificações

```
1  if etapa== 2 or etapa == 4 or etapa == 6:
2          oldx, oldy = posInicial
3          linear = 0.1
4          deltaS = sqrt((float(posx)-oldx)**2) + ((float(posy)-oldy)**2)
5          if (etapa != 4 and deltaS > 1) or (etapa == 4 and deltaS > 2):
6                  posInicial = None
7                  etapa= (etapa + 1) % 8
8          if ((etapa == 2 or etapa == 4 or etapa == 6) and (int (angulos[etapa]) != int (
               anguloAtual))):
9                  valor = angulos[etapa] - anguloAtual
10                 if (valor >=0):
11                         if (valor > 180):
12                                 angular = -0.08
13                         else:
14                                 angular = 0.08
15                 if (valor < 0):
16                         if abs(valor) < 180:
17                                 angular = -0.08
18                         else :
19                                 angular = 0.08
```

O código 6.2 apresenta o código após as modificações para correção durante o trajeto em linha reta. A variável angulos[x] é um mapa que possuí os valores dos ângulos que o robô deveria estar dada a etapa x. A partir dessa variável foi possível comparar o ângulo atual do robô com o que ele deveria estar e dessa forma aplicar valores de 0.08 ou -0.08 na velocidade angular, permitindo assim a correção na trajetória.

A figura 6.4 apresenta o gráfico das trajetórias dos robôs, nela é possível perceber que a modificação no algoritmo de controle provocou uma melhora significativa na trajetória do robô real (em verde) em relação ao experimento anterior.

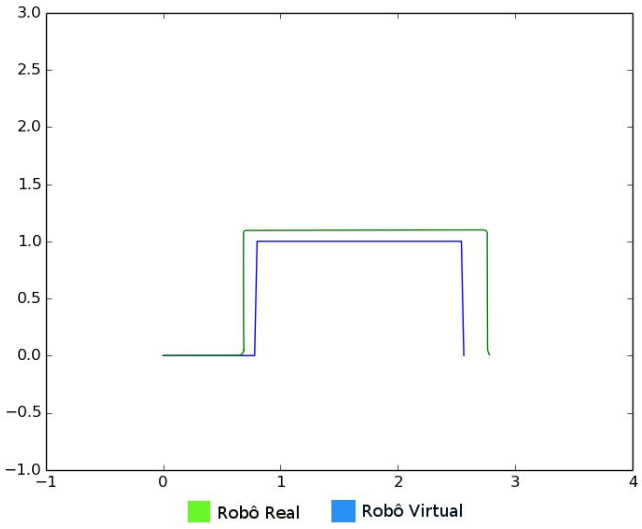


Figura 6.4: Resultado após modificação do algoritmo de controle

Este conjunto de experimentos mostra que é possível ter um ambiente de simulação com um robô real sendo espelhado por um robô avatar e ainda a utilização de um robô completamente virtual, formando assim um ambiente multi-robôs.

Capítulo 7

Conclusões e trabalhos futuros

Este trabalho integrou dois simuladores de diferentes focos, o Stage e o Ptolemy, proporcionando um ambiente de simulação que permite testar sistemas robóticos tanto em relação ao sistema embarcado quanto a física do robô. A co-simulação entre estes simuladores foi realizada com a utilização da Arquietura de Alto Nível. Esta arquitetura administrou tanto o compartilhamento da informação entre os simuladores como o avanço do tempo de simulação nos robôs e no ambiente HLA.

No capítulo 4 foram apresentadas diversas abordagens de cossimulação entre o ROS e o Stage que podem ser úteis dependendo do ambiente que se deseja testar. Também descreve a criação do arquivo FED bem como as variáveis que foram utilizadas para compartilhar as informações entre os simuladores e o experimento para verificar o funcionamento da comunicação entre o Ptolemy e HLA, que permitirá o uso de dispositivos de hardware na simulação, completando a tarefa T1 apresentada na metodologia.

A tarefa T2 era referente a comunicação entre ROS e Ptolemy por meio da HLA. Esta tarefa também foi completada no capítulo 4 por meio do experimento que verifica a comunicação entre os dois ambientes (Stage e Ptolemy).

O estudo do ambiente apresentado no capítulo 5 atende a tarefa T3 que consiste no estudo do ambiente que foi desenvolvido por meio de experimentos e coleta de informações como tempo de resposta. A tarefa T4 foi completada pelos experimentos com o robô espelho apresentados no capítulo 5, mostrando que é possível utilizar um avatar do robô real no ambiente simulado para detecção de falhas de hardware. A tarefa T5 permitiu testar o ambiente com a utilização de robôs reais por meio do ROS, completando assim todas as atividades que foram

propostas na metodologia.

Como resultados, temos o desenvolvimento de um modelo de dados para compartilhamento de informações que foi utilizado nas co-simulações especificando o tipo de informação que era compartilhada entre os simuladores. Para a utilização deste modelo de dados também foi necesário o desenvolvimento de atores no simulador Ptolemy. Com o desenvolvimento destes atores, foi possível a utilização do modelo de dados para uma simulação em que os dados eram compartilhados entre um robô real e a Arquitetura de Alto Nível. Em seguida foi realizada a cossimulação entre Stage e o Ptolemy e consequentemente uma verificação de aspectos de sincronização de simuladores que foram realizados com base nos experimentos de navegação e deteção de falhas de hardware. O algoritmo de controlo utilizado pelos robôs também pode ser considerado como um resultado, podendo ser utilizado por outras aplicações em Python que também utilizem a posição dos robôs para calcular o ponto de destino.

Os resultados experimentais mostraram a eficácia do HLA a sincronizar os vários simuladores envolvidos, independentemente da plataforma de hardware utilizada. Finalmente também verificámos através de um caso simples, como esta abordagem de cossimulação pode ser usada para detectar falhas de hardware, em particular, mecânicas. Também foram realizadas simulações com um turtlebot real integrado ao ambiente, verificamos que a divergência entre o robô real e seu avatar eram notáveis. Mas ainda assim foi possível detectar falhas mecânicas provacas pela roda e reduzir esta falha através de mudanças no algoritmo de controle.

Como trabalhos futuros é possível que novos simuladores sejam integrados ao ambiente de cossimulação. Isso é possível tendo em vista que os simuladores utilizados podem se comunicar com a Arquitetura de Alto Nível, então outros simuladores que também tenham esta capacidade e sejam adaptados para utilizar o mesmo modelo de dados possam assim acessar a informação do ambiente proposto. Poderiam ser integrados simuladores de redes de computadores ou utilizar um simulador robótico 3D para maior exatidão da simulação. Estudos mais aprofundados também podem ser realizados em relação a utilização de *Robot-in-the-Loop*, a modificação do algoritmo de controle que foi utilizado para um algoritmo mais complexo que envolva mais parâmetros e conceitos de visão robótica pode ser incluída ao ambiente.

É possível ainda a implementação de uma função que modifique a posição do robô no ambiente stage a partir da posição obtida pela odometria do robô real. Isso seria importante para dar mais precisão a localização do robô avatar no ambiente de simulação e assim permitir a comparação do trajeto do robô real com o simulado com mais eficácia. Também seria interessante a continuidade do estudo de sincronização do ambiente com a utilização de robôs reais, isso permitiria verificar se não há interferências no ambiente provocadas pelo uso dos robôs reais.

Bibliografia

ARDUINO. *What is arduino*. nov. 2014. Disponível em: <http://arduino.cc/en/Guide/Introduction>.

BOARD, B. *Beaglebone Black*. nov. 2014. Disponível em: <http://beagleboard.org/black>.

BRITO, A. et al. Development and evaluation of distributed simulation of embedded systems using ptolemy and hla. In: *Distributed Simulation and Real Time Applications (DS-RT), 2013 IEEE/ACM 17th International Symposium on*. [S.l.: s.n.], 2013. p. 189–196. ISSN 1550-6525.

BROOKS, C. e. a. *Ptolemy II, Heterogeneous Concurrent Modeling and Design in Java*. fev 2014. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS\discretionary{-}{}{}2007\discretionary{-}{}{}7.pdf>.

FOUNDATION, R. pi. *What is Raspberry pi*. jan. 2015. Disponível em: <http://www.raspberrypi.org/help/what-is-a-raspberry-pi/>.

GAJSKY, D. D. et al. *Embedded System Design: modeling, synthesis and verification*. [S.l.]: Springer, 2013.

GALLA, T. M. *Cluster Simulation in Time Triggered Real-Time Systems*. jan. 2015. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.199.1439&rep=rep1&type=pdf>.

GAZEBO. *Gazebo*. nov. 2014. Disponível em: <http://playerstage.sourceforge.net/>.

GERKEY, B. P.; VAUGHAN, R. T.; HOWARD, A. The player/stage project: Tools for multi-robot and distributed sensor systems. In: *In Proceedings of the 11th International Conference on Advanced Robotics*. [S.l.: s.n.], 2003. p. 317–323.

HU, X. Applying robot-in-the-loop-simulation to mobile robot systems. In: *Advanced Robotics, 2005. ICAR '05. Proceedings., 12th International Conference on*. [S.l.: s.n.], 2005. p. 506–513.

HU, X.; ZEIGLER, B. P. *Measuring Cooperative Robotic Systems Using Simulation-Based Virtual Environment*. jan. 2015. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.7625&rep=rep1&type=pdf>.

IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)–Federate Interface Specification. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, p. 1–378, Aug 2010.

IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, p. 1–38, Aug 2010.

IEEE Standard for Modeling and Simulation (M amp;S) High Level Architecture (HLA)– Object Model Template (OMT) Specification - Redline. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000) - Redline*, p. 1–112, Aug 2010.

IEEE Standard Glossary of Modeling and Simulation Terminology. *IEEE Std 610.31989*, 1989.

LANE, D. et al. Interoperability and synchronisation of distributed hardware-in-the-loop simulation for underwater robot development: issues and experiments. In: *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. [S.l.: s.n.], 2001. v. 1, p. 909–914 vol.1. ISSN 1050-4729.

MARTIN, A.; EMAMI, M. An architecture for robotic hardware-in-the-loop simulation. In: *Mechatronics and Automation, Proceedings of the 2006 IEEE International Conference on*. [S.l.: s.n.], 2006. p. 2162–2167.

MARTIN, A.; SCOTT, E.; EMAMI, M. Design and development of robotic hardware-in-the-loop simulation. In: *Control, Automation, Robotics and Vision, 2006. ICARCV '06. 9th International Conference on*. [S.l.: s.n.], 2006. p. 1–6.

MORSE. *Morse*. nov. 2014. Disponível em: <https://www.openrobots.org/morse/doc/latest/what_is_morse.html>.

NEGREIROS, A. L. V. de; BRITO, A. V. Análise da aplicação de simulação distribuída no projeto de sistemas embarcados. 2013.

NEGREIROS, A. Vidal de; BRITO, A. V. The development of a methodology with a tool support to the distributed simulation of heterogeneous and complexes embedded systems. In: *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*. [S.l.: s.n.], 2012. p. 37–42. ISSN 2324-7886.

NONGNU. *PyHLA — Python Bindings for M&S HLA*. nov. 2014. Disponível em: <http://www.nongnu.org/certi/PyHLA/>.

PTOLEMY. *The Ptolemy projetct*. jan. 2014. Disponível em: <http://ptolemy.eecs.berkeley.edu/index.htm>.

QUIGLEY, M. et al. Ros: an open-source robot operating system. In: *ICRA Workshop on Open Source Software*. [S.l.: s.n.], 2009.

ROS. *ROS*. nov. 2014. Disponível em: <http://wiki.ros.org/pt>.

ROS. *Learn turtle and ROS*. jun. 2015. Disponível em: <http://learn.turtlebot.com/2015/02/01/1/>.

SAVANNAH. *CERTI resumo*. nov. 2014. Disponível em: <http://savannah.nongnu.org/projects/certi>.

SCHAUMONT, P.; VERBAUWHEDE, I. Interactive cosimulation with partial evaluation. In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. [S.l.: s.n.], 2004. v. 1, p. 642–647 Vol.1. ISSN 1530-1591.

SCHLAGER, M. *Hardware-in-the-Loop Simulation: A Scalable, Component-based, Time-triggered Hardware-in-the-loop Simulation Framework*. [S.l.]: VDM Verlag Dr. Müller, 2008.

SCHLAGER, M.; ELMENREICH, W.; WENZEL, I. Interface design for hardware-in-the-loop simulation. In: *Industrial Electronics, 2006 IEEE International Symposium on*. [S.l.: s.n.], 2006. v. 2, p. 1554–1559.

TERASIC. *DEi2-150 FPGA development kit*. nov. 2014. Disponível em: <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=529>.

Apêndice A

Código do arquivo FED

```
1
2  (FED
3  (Federation TestFed)
4  (FEDversion v1.3)
5  (spaces
6  )
7  (objects
8  (class ObjectRoot
9  (attribute privilegeToDelete reliable timestamp)
10 (class RTIprivate)
11         (class robot
12                 (attribute id reliable timestamp)
13                 (attribute battery reliable timestamp)
14                 (attribute temperature reliable timestamp)
15                 (attribute sensor1 reliable timestamp)
16                 (attribute sensor2 reliable timestamp)
17                 (attribute sensor3 reliable timestamp)
18                 (attribute gps reliable timestamp)
19                 (attribute compass reliable timestamp)
20                 (attribute goto reliable timestamp)
21                 (attribute rotate reliable timestamp)
22                 (attribute activate reliable timestamp)
23         )
24         (class robot1
25                 (attribute id reliable timestamp)
26                 (attribute battery reliable timestamp)
27                 (attribute temperature reliable timestamp)
28                 (attribute sensor1 reliable timestamp)
29                 (attribute sensor2 reliable timestamp)
30                 (attribute sensor3 reliable timestamp)
31                 (attribute gps reliable timestamp)
32                 (attribute compass reliable timestamp)
```

```
33                  (attribute goto reliable timestamp)
34                  (attribute rotate reliable timestamp)
35                  (attribute activate reliable timestamp)
36          )
37          (class robot2
38                  (attribute id reliable timestamp)
39                  (attribute battery reliable timestamp)
40                  (attribute temperature reliable timestamp)
41                  (attribute sensor1 reliable timestamp)
42                  (attribute sensor2 reliable timestamp)
43                  (attribute sensor3 reliable timestamp)
44                  (attribute gps reliable timestamp)
45                  (attribute compass reliable timestamp)
46                  (attribute goto reliable timestamp)
47                  (attribute rotate reliable timestamp)
48                  (attribute activate reliable timestamp)
49          )
50          (class robot3
51                  (attribute id reliable timestamp)
52                  (attribute battery reliable timestamp)
53                  (attribute temperature reliable timestamp)
54                  (attribute sensor1 reliable timestamp)
55                  (attribute sensor2 reliable timestamp)
56                  (attribute sensor3 reliable timestamp)
57                  (attribute gps reliable timestamp)
58                  (attribute compass reliable timestamp)
59                  (attribute goto reliable timestamp)
60                  (attribute rotate reliable timestamp)
61                  (attribute activate reliable timestamp)
62          )
63
64  )
65  )
66  (interactions
67  )
68  )
```

Apêndice B

Código do Controle

```
1
2  # -*- coding: utf-8 -*-
3  import math
4
5
6  class Controlo :
7          def __init__ (self):
8                  self.VEL_MAX= 0.6
9                  self.lex= 0
10                 self.ley= 0
11                 self.mId= 0
12                 self.mmx=0
13                 self.mmy=0
14                 self.mmz=0
15                 self.targets = [[10, 10], [-10, 10], [-10, -10] , [10, -10]]
16                 self.myTarget = 0
17
18         def myPosition(self):
19                 return self.mmx, self.mmy, self.mmz
20         def walkon(self):
21                 return self.VEL_MAX, 0 #linear and angular modificado 1
22         def walkhorario(self):
23                 return 0, -self.VEL_MAX #linear and angular modificado -0.5
24         def walkantihorario(self):
25                 return 0, self.VEL_MAX#linear and angular modificado 0.5
26         def degrees(self, value):
27                 return ((value* 180.0)/math.pi)
28         def changeTarget(self):
29                 self.myTarget = (self.myTarget + 1) % (len (self.targets))
30                 print str (self.myTarget)
31         #Anda em frente e direciona aos lados
32         def walkhorarioon(self, vel):
```

```
33              return 0.3, -self.VEL_MAX #linear and angular modificado -0.5
34          def walkantihorarioon(self, vel):
35              return 0.3, self.VEL_MAX #linear and angular modificado 0.5
36          def walkonhorario(self):
37              return self.VEL_MAX, -0.3 #linear and angular modificado 0.5
38          def walkonantihorario(self):
39              return self.VEL_MAX, 0.3 #linear and angular modificado 0.5
40
41
42
43          def whereImGoing(self):
44              lx= float(self.lex)
45              ly= float (self.ley)
46              myId = self.mId
47              distance = 2
48              if (myId == "0"or myId == "00"): # Im the leader
49                  return self.targets[self.myTarget][0], self.targets[self.myTarget
                        ][1]
50              elif (myId == "1" or myId == "11"):
51                  return lx -distance , ly - distance
52              elif (myId == "2" or myId == "22"):
53                  return lx - distance - distance , ly  - distance - distance
54              elif (myId == "3"or myId == "33"):
55                  return lx - distance - distance -distance , ly  - distance -
                        distance -distance
56              return 0, 0
57
58
59          def isOriented(self):
60              x, y = self.whereImGoing()
61              mx, my, mz = self.myPosition()
62              if (not (mx == 0 and my == 0)):
63                  #calcula a distancia entre meu ponto e o ponto que quero ir
64                  hip = math.hypot (x - mx, y - my)
65                  #calcula a distancia dos dois catetos
66                  tmp1 = math.hypot( x -mx, 0 )
67                  tmp2 = math.hypot( 0, y -my )
68                  #seleciona o maior cateto
69                  cat = max ([tmp1, tmp2])
70                  #calcula o angulo que preciso estar para
71                  anguloEsperado = self.degrees(math.cos(float(cat)/hip))
72                  deltax = x - mx
73                  deltay = y - my
74                  deltax = abs(deltax)
75                  deltay = abs(deltay)
76                  if (deltax <0.19) and y > my:
77                          anguloEsperado = 90
```

```
78                      elif (deltax<0.19) and y < my:
79                              anguloEsperado = 270
80                      elif (deltay<0.19) and mx < x:
81                              anguloEsperado = 0
82                      elif (deltay<0.19) and mx > x:
83                              anguloEsperado = 180
84                      elif mx < x and my <=y:
85                              pass#anguloEsperado += 180
86                      elif mx < x and my >= y:
87                              anguloEsperado= anguloEsperado +270# = 180 - anguloEsperado
88                      elif mx > x and my <= y:
89                              anguloEsperado= anguloEsperado + 90# = 360 - anguloEsperado
90                      elif mx > x and my >= y:
91                              anguloEsperado =anguloEsperado + 180# anguloEsperado
92                      a = max ([anguloEsperado, mz])
93                      b = min ([anguloEsperado , mz])
94                      limin = 3
95                      if ((a - b) < limin or ((a-b)>(360-limin))):
96                              return True , anguloEsperado, mz, hip
97                      return False, anguloEsperado, mz, hip
98              return False, 1000, 800, 1000
99
100
101     def inPosition(self):
102             x, y = self.whereImGoing()
103             mx, my, mz = self.myPosition()
104             if ((math.hypot(x-mx, y-my))< 0.3):
105                     return True
106             return False
107
108
109     def walk (self):
110             x, y = self.whereImGoing()
111             mx, my, mz = self.myPosition()
112             myId= str (self.mId)
113             if (not self.inPosition()):
114                     orient, ang, mz, hip = self.isOriented()
115                     if (orient):
116                             #muito Orientado
117                             if (int (mz) == int (ang)):
118                                     #self.log.broadcast(ptolemy.data.StringToken("em
                                            frente "))
119                                     return self.walkon()
120                             #Orientado mas necessita de ajustes
121                             else:
122                                     if ((ang-mz)>= 0):
123                                             if (ang-mz)< 180:
```

```
124                                                     return self.walkonantihorario()
125                                             else:
126                                                     return self.walkonhorario()
127                                     else:
128                                             if (ang-mz)< 180:
129                                                     return self.walkonhorario()
130                                             else:
131                                                     return self.walkonantihorario()
132
133                     if hip < 1.5:
134                             if ((ang - mz) >= 0):
135                                     if ((ang- mz) < 180):
136                                             return self.walkantihorario()
137                                     else:
138                                             return self.walkhorario()
139                             elif((ang- mz) < 0):
140                                     if (abs((ang- mz)) < 180):
141                                             return self.walkhorario()
142                                     else:
143                                             return self.walkantihorario()
144                     else:
145                             velocidade = 0.5
146                             if ((ang - mz) >= 0):
147                                     if ((ang- mz) < 180):
148                                             return self.walkantihorarioon(velocidade)
149                                     else:
150                                             return self.walkhorarioon(velocidade)
151                             elif((ang- mz) < 0):
152                                     if (abs((ang- mz)) < 180):
153                                             return self.walkhorarioon(velocidade)
154                                     else:
155                                             return self.walkantihorarioon(velocidade)
156             self.changeTarget()
157             return 0, 0
158
159     def start(self,mid,  x, y , mx, my, mz) :
160             self.mId = mid
161             #leader position
162             self.lex  = x
163             self.ley  = y
164             #my position
165             self.mmx  = mx
166             self.mmy  = my
167             self.mmz  = mz
168             return self.walk()
```

Apêndice C

Código do componente ponte

```
1
2  ################
3  ## Main loop  ##
4  ################
5
6  try:
7          while not rospy.is_shutdown():
8                  iteracoes+= 1
9                  ######################################
10                 ### Bridge handling data from  HLA  ##
11                 ######################################
12                 if mya.hasData():
13                         evento = mya.getData()
14                         _goto = evento["goto"]
15                         _tempo =evento["time"]
16                         _rid = evento["id"]
17                         if (_rid.count(str (mId[0]) ) >0): # If _rid is equivalent to myId
18                                 #Walk
19                                 if (_goto.count("none")<1 and _goto.count(";")== 1):
20                                         global mapaFim
21                                         _goto = _goto.replace("\\", "")
22                                         _goto = _goto.replace("\"", "")
23                                         lin, ang = _goto.split(";") # Linear an Angular
24
25                                         #removing bad chars
26                                         safe_chars = string.digits + '-.'
27                                         ang = ''.join([char if char in safe_chars else ''
                                               for char in ang])
28                                         lin = ''.join([char if char in safe_chars else ''
                                               for char in lin])
29
30                                         twist = Twist()
```

```
31                                      twist.linear.x = round(float(lin), 2)
32                                      twist.angular.z = round(float(ang), 2)
33                                  p.publish(twist)
34                                  pSimulado.publish(twist)
35
36                  ###################################
37                  ### Bridge Sending Data to HLA  ###
38                  ###################################
39                  if hasDataToHLA():
40                          laserSensor  = getLaserData()
41                          posRealX, posRealY, posRealZ = getRealData()
42                          posVirtualX, posVirtualY, posVirtualZ = getVirtualData()
43
44                          #send just some information
45                          sendData(mId, "", "", str(laserSensor), str(posRealZ), "<"+str(
                                posVirtualX)+";"+str(posVirtualY)+";"+str(posVirtualZ)+">"
                                , "<"+str(posRealX)+";"+str(posRealY)+";"+str(posRealZ)+
                                ">", "", "", "", str( newConter ))
46                  IteracoesROS += 1
47
48                  ######  Time Management  ########
49                  timeHLA = rtia.queryFederateTime() + 1
50                  rtia.timeAdvanceRequest(timeHLA)
51                  #print("Tempo Atual no HLA = ", timeHLA)
52                  while(mya.advanceTime == False):
53                          rtia.tick()
54                  mya.advanceTime = False
55                  ###################################
56
57                  if(SLEEP):
58                          r.sleep()
59
60  except Exception as e:
61          raise e
```

Código do Ambassador

```
1  #!/usr/bin/env python
2
3  #ROS
4  import rospy
5  from sensor_msgs.msg import LaserScan
6  from geometry_msgs.msg import Twist
7  from nav_msgs.msg import Odometry
8  import random
9
10 #CERTI - HLA
11 import hla.rti
```

```
12 import hla.omt as fom
13 import struct
14
15
16 ################################
17 #    The   Ambassador class      #
18 ################################
19
20 import time
21 getTime = lambda: int(round(time.time() * 1000))
22
23 class MyAmbassador(hla.rti.FederateAmbassador):
24
25         def initialize(self, value, number=""):
26                 self._rtia = value
27                 #Variables
28                 self.time = 0
29                 self.advanceTime = False
30                 self.isRegistered = False
31                 self.isAnnounced = False
32                 self.isReady = False
33                 self.isConstrained = False
34                 self.isRegulating = False
35                 self.posx = None
36                 self.posy = None
37                 self.id = None
38
39                 self.listaEventos= []
40
41                 self.classHandle = self._rtia.getObjectClassHandle("ObjectRoot.robot"+str(
                       number))
42                 self.log ("Using object ObjectRoot.robot"+str(number))
43                 self.idHandle = self._rtia.getAttributeHandle("id", self.classHandle)
44                 self.batteryHandle = self._rtia.getAttributeHandle("battery", self.
                       classHandle)
45                 self.temperatureHandle = self._rtia.getAttributeHandle("temperature", self.
                       classHandle)
46                 self.sensor1Handle = self._rtia.getAttributeHandle("sensor1", self.
                       classHandle)
47                 self.sensor2Handle = self._rtia.getAttributeHandle("sensor2", self.
                       classHandle)
48                 self.sensor3Handle = self._rtia.getAttributeHandle("sensor3", self.
                       classHandle)
49                 self.gpsHandle = self._rtia.getAttributeHandle("gps", self.classHandle)
50                 self.compassHandle = self._rtia.getAttributeHandle("compass", self.
                       classHandle)
51                 self.gotoHandle = self._rtia.getAttributeHandle("goto", self.classHandle)
```

```
52                  self.rotateHandle = self._rtia.getAttributeHandle("rotate", self.
                        classHandle)
53                  self.activateHandle = self._rtia.getAttributeHandle("activate", self.
                        classHandle)
54                  #Subscribe HLA
55                  self._rtia.subscribeObjectClassAttributes(self.classHandle,[self.idHandle,
                        self.batteryHandle, self.temperatureHandle, self.sensor1Handle, self.
                        sensor2Handle, self.sensor3Handle, self.gpsHandle, self.compassHandle,
                        self.gotoHandle, self.rotateHandle, self.activateHandle])
56                  #Publish HLA
57                  self._rtia.publishObjectClass(self.classHandle,[self.idHandle, self.
                        batteryHandle, self.temperatureHandle, self.sensor1Handle, self.
                        sensor2Handle, self.sensor3Handle, self.gpsHandle, self.compassHandle,
                        self.gotoHandle, self.rotateHandle, self.activateHandle])
58                  self.myObject = self._rtia.registerObjectInstance(self.classHandle)#, "
                        ROBO_2")
59
60          ############################
61          #Calbacks from CERTI - HLA#
62          ############################
63
64          def reflectAttributeValues(self, object, attributes, tag, order, transport, time=
                None, retraction=None):
65                  #self.attMap["time"] = self._rtia.queryFederateTime()
66
67                  attMap= {}
68                  attMap["time"] = getTime()
69                  attMap["id"] = attributes[self.idHandle]
70                  attMap["battery"]= attributes[self.batteryHandle]
71                  attMap["temperature"]= attributes[self.temperatureHandle]
72                  attMap["sensor1"]= attributes[self.sensor1Handle]
73                  attMap["sensor2"]= attributes[self.sensor2Handle].replace("sensor2:","").
                        replace("\x00","")
74                  attMap["sensor3"]= attributes[self.sensor3Handle]
75                  attMap["gps"] = attributes[self.gpsHandle]
76                  attMap["compass"] = attributes[self.compassHandle]
77                  attMap["goto"] = attributes[self.gotoHandle]
78                  attMap["rotate"]= attributes[self.rotateHandle]
79                  attMap["activate"]= attributes[self.activateHandle]
80                  self.listaEventos.append(attMap)
81  #               self.hasData= True
82  #print (attributes[self.gotoHandle])
83
84          def getData(self):
85                  return self.listaEventos.pop(0)
86
87          def hasData(self):
```

```
88              return (len(self.listaEventos)>0)
89
90      def log (self, valor):
91              print ("\033[34m" + valor + "\033[0;0m")
92
93      def terminate(self):
94              self._rtia.deleteObjectInstance(self.myObject, "ROBO_2")
95
96      def startRegistrationForObjectClass(*params):
97              print("START", params)
98
99      def provideAttributeValueUpdate(*params):
100             print("PROVIDE UAV", params)
101
102     def synchronizationPointRegistrationSucceeded(self, label):
103             self.isRegistered = True
104             print ("MyAmbassador: Registration Point Succeeded")
105
106     def announceSynchronizationPoint(self, label, tag):
107             self.isAnnounced = True
108             print ("MyAmbassador: Announce Synchronization Point")
109
110     def federationSynchronized (self,  label):
111             self.isReady = True
112             print ("MyAmbassador: Ready to run ")
113
114     def timeConstrainedEnabled (self, time):
115             self.isConstrained = True
116
117     def timeRegulationEnabled (self, time):
118             self.isRegulating = True
119
120     def discoverObjectInstance(self, object, objectclass, name):
121             print("DISCOVER", name)
122             self._rtia.requestObjectAttributeValueUpdate(object,[self.idHandle, self.
                    batteryHandle, self.temperatureHandle, self.sensor1Handle, self.
                    sensor2Handle, self.sensor3Handle, self.gpsHandle, self.compassHandle,
                    self.gotoHandle, self.rotateHandle, self.activateHandle])
123     def timeAdvanceGrant (self, time):
124             self.advanceTime = True
```