System for Creation, Programming and Availability of Distributed Robot Teams using Collective Knowledge

P. H. L. Silva and A. M. F. Burlamaqui

Abstract— This paper addresses the problem of programming, control and availability of distributed robots teams. Systems based on robots working together collaboratively has been the subject of much research in recent years. For distributed robots systems there is a latent need for research to enable the use of such systems by end users with no or almost no experience with programming or robotics. Thus, this paper proposes a system architecture that allows the creation and programming of robot teams, allowing and users to manage them as a single system and simple way using collective knowledge acquired both programmers as of the and users. The tests are performed by an application that was developed under the concepts of this architecture. It provides a set of special operations by which and users can connect, create and control robots teams at high level. The process of creation, programming and control of two teams is presented in well defined steps. The results are satisfactory and show that the operating strategy of the proposed system meets fundamental aspects at the system level such as modularity, flexibility and ease of access for end users.

Keywords— Distributed Robots, Creation of Robot Teams, Programming Robot Teams, Collective Knowledge.

I. INTRODUÇÃO

ATUALMENTE, a robótica tem um papel muito importante na realização de diversas tarefas do cotidiano dos seres humanos. Seu campo de atuação vem se expandindo com muita rapidez. Já é possível observar a presença de robôs em um número cada vez maior de ambientes. Com isso, há uma tendência nas pesquisas dessa área na busca por soluções que aperfeiçoem ainda mais a interação dos robôs nos mais variados níveis da sociedade [1].

Vários ramos da ciência e tecnologia conduzem pesquisas na busca por avanços que irão aperfeiçoar a civilização, trazendo benefícios na indústria [2], medicina [3] e educação [4], por exemplo. Isto, por consequência, tende a melhorar a qualidade de vida das pessoas. Na robótica, estudos sobre a utilização de robôs pessoais, para casa ou ambiente de trabalho, tem avançado bastante [5,6].

Com a rápida disseminação das redes de computadores e Internet, vários projetos foram iniciados na perspectiva de que vários recursos dispersos e heterogêneos pudessem cooperar em uma aplicação específica, dando-lhe maior flexibilidade, poder computacional e, por conseguinte, um maior alcance [7,8].

Os sistemas multi-robôs distribuídos tem sido alvo de

intensa pesquisa nos últimos anos. Missões de exploração, busca e resgate em ambientes perigosos e inacessíveis são exemplos onde uma coleção de robôs trabalhando de forma colaborativa se tornam uma alternativa eficaz [9,10]. Apesar dos sistemas de robô único oferecerem um ótimo desempenho, essas tarefas são mais complexas do seu ponto de vista.

Entretanto, apesar de suas vantagens claras, o gerenciamento de times de robôs distribuídos adiciona aspectos como comunicação eficiente, estado do sistema, tolerância à falhas. Isso faz com que seu desenvolvimento seja mais complexo, exigindo soluções mais sofisticadas. Alguns resultados interessantes podem ser mencionados em temas como controle e formação de times de robôs móveis com sistemas multiagentes [11], tolerância e isolamento de falhas [12] e planejamento dinâmico de rota de times de robôs [13].

Além de estratégias e algoritmos, uma grande quantidade de trabalhos envolvendo a especificação de padrões, modelos e middlewares foram propostos na tentativa oferecer uma solução eficiente para os problemas inerentes ao desenvolvimento de software e interação com usuários finais em sistemas de robôs distribuídos.

Projetos como OPRoS [14], ROS [15] e *RoboEarth* [16] são exemplos de sistemas que oferecem abstrações de software para ocultar do desenvolvedor, em sistemas de robô único ou multi-robôs, a heterogeneidade de hardware, software e protocolos de comunicação existente em robótica. São soluções eficazes no que se propõem a realizar, contudo, requerem que o especialista em desenvolvimento de software e robótica se familiarize com os conceitos do modelo ou do middleware e isso, em muitos desses casos, pode levar algum tempo.

Diante do exposto, concluímos que ainda existem problemas no desenvolvimento de sistemas para gerenciamento de times de robôs. Em particular, existe espaço para buscar soluções em sistemas específicos para usuários finais acessarem e programarem tarefas para conjuntos de robôs distribuídos de uma forma mais simplificada.

Portanto, este trabalho tem como objetivo contribuir com o estado da arte no gerenciamento de times de robôs distribuídos. A proposta é de uma arquitetura de sistema que permite a criação, programação e acesso a times de robôs. Ela fornece mecanismos para que usuários finais acessem um time de robôs como se fosse um único robô e de maneira simples. Assim sendo, usuários finais não precisam ser especialistas em robótica ou programação. Além do mais, o sistema propõe um mecanismo que armazena o conhecimento oriundo de programadores e usuários finais na configuração de times.

Os testes para validar a solução proposta são realizados por meio de uma aplicação que implementa os conceitos da

P. H. L. Silva, Universidade Federal Rural do Semiárido (UFERSA), paulohlsilva@gmail.com.

A. M. F. Burlamaqui, Universidade Federal do Rio Grande do Norte (UFRN), aquilesburlamaqui@gmail.com.

arquitetura. Trata-se de um middleware, desenvolvido em Java, que interliga os componentes da arquitetura via conexões TCP/IP. Dois times, usando robôs da *LEGO Mindstorms®*, foram criados. São apresentados, para os dois times, o processo de criação, programação e disponibilização de acordo com o sistema proposto.

Esse artigo está organizado em seções bem definidas que são brevemente descritas a seguir. A seção II apresenta a arquitetura proposta para o sistema de criação, programação e disponibilização de times de robôs. A seção III mostra o processo de desenvolvimento de um middleware que implementa as regras conceituais desse sistema. Na seção IV, mostra o processo de validação do sistema por meio da realização de dois experimentos com times de robôs. A seção V é destinada a discussão sobre os resultados e impressões dos experimentos realizados. Na seção VI, são feitas as considerações finais e propostas de trabalhos futuros.

II. ARQUITETURA DO SISTEMA

Um time pode ser formado por um conjunto de dois ou mais robôs. A arquitetura do sistema é dividida em componentes com funções bem definidas. A definição de um modelo para a arquitetura proposta visa apresentar suas funcionalidades e ao mesmo tempo separá-las em partes independentes. Cada parte executa funcionalidades diferentes, mas complementares a nível de sistema.

A Fig. 1 mostra a arquitetura do sistema que permite a criação, programação e execução de funções para um time de robôs. Cada componente possui um canal bidirecional de comunicação com o núcleo do sistema, por onde ocorrem trocas de mensagens sob um protocolo específico. Essas comunicações ocorrem como eventos independentes.

Figura 1. Componentes do sistema.

Além disso, essa definição conceitual dos componentes da arquitetura nos permite vislumbrar seus objetivos sem se fixar a nenhuma tecnologia de desenvolvimento.

A. Núcleo

O **núcleo** é responsável por gerenciar as comunicações com os outros componentes do sistema e realizar a formação de times de robôs. Cada componente da arquitetura interage de

uma maneira diferente com o núcleo, ou seja, possui visões diferentes do sistema.

Para poder formar times de robôs, o núcleo possui um registro único de descrição de robôs. Essa descrição funciona como uma referência global e é diferenciada pelo tipo, formato e as funções que este robô pode ter. Por exemplo, um robô com formato de garra, geralmente, possui funções para agarrar e soltar objetos. Essa é uma descrição genérica que pode ser usada para qualquer robô do tipo garra, independente de seu *hardware* de fabricação.

Assim, é possível para o programador configurar o comportamento dos times por meio dessas descrições. As configurações de times são armazenadas em uma base de conhecimento. Antes de instanciar um time, o núcleo verifica se existe alguma configuração do mesmo na base de conhecimento, por meio de fatos e regras bem definidas, baseadas na lógica de predicados.

A Fig. 2 mostra uma representação gráfica de um time de robôs genérico. Ele é um componente virtual com canais de comunicação bidirecionais com cada robô membro e é executado no núcleo do sistema.

Figura 2. Modelo de um time de robôs.

Estruturas de dados como grafos ou árvores podem ser usadas para modelar e implementar um time de robôs.

B. Robô

Um **robô** interage com o núcleo por meio da disponibilização de informações essenciais para a formação de times. Cada robô, ao se conectar com o núcleo, necessita enviar-lhe seu nome, seu tipo, seu formato e uma descrição de suas funções.

Dessa forma, o núcleo passa a conhecer as características do robô recém conectado. Após isso, o robô fica aguardando chamadas de funções enviadas pelo núcleo.

C. Programação de times

É por meio do componente de **programação de times** que um desenvolvedor define as configurações de um time. Essa configuração compreende quais robôs formam aquele time, bem como o nome do time e suas funções. Para isso, o núcleo envia ao programador de times as descrições dos tipos de robôs que possuem suporte pelo sistema.

A programação de times pode ocorrer a qualquer instante. As descrições enviadas pelo núcleo funcionam como referências abstratas de robôs, permitindo que se possa programar as ações desejadas para um time antes mesmo de haverem robôs conectados ao sistema. A ideia é que cada robô seja mapeado para o sistema por meio de uma dessas descrições.

Por exemplo, considere um robô que se locomove com rodas. Geralmente, um robô com essa estrutura possui funções para ir para frente, virar à direita, etc. O mecanismo que a arquitetura oferece para padronizar a descrição de robôs com rodas segue o que está especificado na interface genérica mostrada na Fig. 3.

```
interface RoboComRodas{
    void irParaFrente(<lista-de-argumentos>);
    void irParaTras(<lista-de-argumentos>);
    void virarAEsquerda(<lista-de-argumentos>);
    void virarADireita(<lista-de-argumentos>);
    void parar(<lista-de-argumentos>);
}
```

Figura 3. Descrição de um robô com rodas.

Outro exemplo interessante é um robô com garra. Nessa arquitetura, um robô com garra pode possuir a descrição genérica mostrada na Fig. 4.

```
interface RoboComGarra{
    void pegarObjeto(<lista-de-argumentos>);
    void soltarObjeto(<lista-de-argumentos>);
    void moverParaEsquerda(<lista-de-argumentos>);
    void moverParaDireita(<lista-de-argumentos>);
    void moverParaCima(<lista-de-argumentos>);
    void moverParaBaixo(<lista-de-argumentos>);
}
```

Figura 4. Descrição de um robô com garra.

Com o sistema para a formação de times é possível estipular, para cada time, funções que são compostas por outras funções. Ou seja, a programação das funções do time ocorre pela associação de funções dos robôs que o compõe. Cada time terá um identificador único no sistema. Além disso, as configurações de cada time formado podem ser compartilhadas para outros programadores de times.

D. Interação com usuário final

O usuário final se conecta ao sistema por meio do componente de **interação com o usuário**. Um dos objetivos da arquitetura proposta é fazer com que o usuário final tenha a visão do time de robôs como sendo um robô único, fazendo com que ele tenha acesso ao time com a sensação de que está acessando as funções de, somente, um robô.

Quando um usuário final faz chamadas às funções do time, o que ocorre, em segundo plano, é emissão de comandos ao núcleo, que, por sua vez, os transformará em subcomandos específicos a serem enviados ao robôs que formam o time.

Este usuário terá conhecimento apenas dos times e não dos dispositivos que o compõem, cabendo esta função ao núcleo do sistema. Além disso, é ocultado dele, também, as regras de formação de times. A ele, é enviada somente uma referência do time e de suas funções.

III. UM MIDDLEWARE PARA GERENCIAMENTO DE TIMES DE ROBÔS

Para realizar experimentos práticos e validar o modelo proposto é necessário ter uma aplicação que funciona como um middleware para interligar robôs e formar times, além de gerenciar conexões com usuários e programadores. Trata-se de

uma aplicação desenvolvida para executar as funcionalidades que o modelo arquitetural especifica.

Para o seu desenvolvimento, a linguagem Java foi escolhida. Ela oferece características importantes como portabilidade, segurança, acesso a protocolos de Internet, programação concorrente e integração com outras linguagens [17], permitindo, assim, que o middleware possa alcançar os objetivos propostos pelo modelo apresentado. Ademais, se trata de um projeto que recebe manutenções e atualizações constantes.

A. Implementação dos componentes no middleware

O middleware foi desenvolvido de acordo com os conceitos da arquitetura cliente/servidor, onde processos se comunicam via rede por meio do padrão requisição/resposta. Cada componente é implementado como um processo do sistema.

As ações do **núcleo do sistema** são executadas por um servidor que recebe conexões via *sockets* TCP/IP. Dessa forma, esse servidor é responsável por gerenciar conexões com robôs, programador de times e usuário final, oferecendo a cada um deles uma visão específica do sistema. Este servidor é *multithread*. Portanto, permite a conexão de vários clientes ao mesmo tempo.

A base de conhecimento situada no núcleo é desenvolvida usando fatos e regras da linguagem Prolog [18]. Ela permite a representação de conhecimento sobre times de uma maneira concisa e escalável.

O componente de **programação de times** é responsável por permitir que desenvolvedores possam programar times de robôs, definindo seu nome e suas funções. É implementado como um cliente que realiza a comunicação com o núcleo por uma conexão TCP/IP.

É por meio do componente de **interação com usuário** que usuários finais podem acessar times de robôs via núcleo do sistema. Ele é implementado como um cliente Java que acessa o núcleo por meio de uma conexão TCP/IP. Sua interface foi desenvolvida em modo texto, que habilita, com simplicidade, o envio e recebimento de mensagens do núcleo.

B. Comunicação entre os componentes

A comunicação entre os componentes do sistema ocorre por meio de um protocolo específico de troca de mensagens. Contudo, antes de descrever o protocolo, é necessário descrever uma estrutura de dados que reúne informações essenciais para a conexão de robôs e formação de times.

A Fig. 5 mostra a estrutura da classe *RobotData*. É a forma pela qual um robô torna-se visível para o sistema. Tem papel fundamental tanto na criação e formação de times de robôs, quanto nas comunicações entre os componentes do sistema. A partir disso, o protocolo estabelece operações para troca de mensagens (*sendRobotData()* e *receiveRobotData()*).

```
class RobotData{
    String robotName;
    String robotType;
    String robotFormat;
    ArrayList<String> robotFunctions;
}
```

Figura 5. Classe *RobotData*.

Vale salientar ainda que antes da troca de mensagens ser efetivada, as informações contidas em um objeto *RobotData* são convertidas para um objeto JSON (*JavaScript Object Notation*) [19]. Sua escolha se justifica pelo fato de ser um padrão aberto e compacto, além de possuir integração com muitas linguagens e tecnologias de programação.

C. Formação de um time de robôs

Antes de um time poder ser formado, é preciso mencionar como os **robôs** são integrados ao sistema. Essa integração diz respeito ao mapeamento das funções nativas do robô para as funções especificadas nas descrições de robôs que estão no núcleo. Ou seja, os robôs são incorporados por meio de classes que podem implementar essas descrições (interfaces em Java) e realizar o encapsulamento de suas funções nativas em funções (métodos) associadas a eles no sistema. Além disso, possui uma variável do tipo *RobotData*, que é fundamental para a troca de informações entre os componentes do middleware.

A Fig. 6 mostra a representação de uma classe com as características mencionadas. Considere um robô que possui dois motores (A e B) que são ligados, cada um, a uma roda. A classe *Robo1* é responsável por integrar este robô ao sistema. Ela implementa os métodos da interface *RoboComRodas* e possui uma variável *RobotData*. Por exemplo, o método *virarAEsquerda()* encapsula funções nativas do robô que fazem com que ele realize a ação de virar à esquerda. Para cada método implementado são encapsuladas funções nativas que correspondem a uma determinada ação.

Figura 6. Implementação de um robô no sistema.

O processo de formação de um time ocorre durante a interação entre o programador de times e o núcleo. Tal processo é implementado de acordo com o que é especificado na arquitetura do sistema. Tomemos como exemplo simples um time formado por dois robôs que se locomovem com rodas, mostrado na Fig. 7.

Em sua estrutura, cada robô possui dois motores, um do lado direito e um do lado esquerdo, sendo que cada motor está conectado a uma roda. Além disso, possui funções como *irParaFrente()*, *irParaTras()*, *virarADireita()* e *virarAEsquerda()*, por exemplo. Um time formado por esses dois robôs pode ter funções do tipo *irParaFrente()* e *irParaTras()*. Essas funções são compostas por funções de cada robô conectado. Por exemplo, a função *irParaFrente()* é composta pela função *irPraFrente()* do Robô 1 e *irPraFrente()* do Robô 2.

Figura 7. Formação de um time de robôs.

Para o caso das funções *virarADireita()* e *virarAEsquerda()*, trabalhamos com outra questão: a ação colaborativa do time. Essas ações devem provocar um comportamento colaborativo nos robôs. Quando essas funções são criadas, o programador de times especifica como cada robô deve se comportar (velocidade e sentido de movimentação) quando elas são chamadas por usuários finais. As estratégias usadas nesse trabalho para realizar essas ações são detalhadas na seção IV.

Após criar o time e suas funções, o componente de programação de times encapsula essas informações em um objeto *RobotData*, visto que para o sistema um time de robôs também representa um robô único, e envia-o para o núcleo. Este, por sua vez, registra a configuração do time recém criado em sua base de conhecimento.

Quando um usuário final deseja acessar um time de robôs, ele necessita realizar dois procedimentos. O primeiro é conectar os robôs, por meio de uma interface simples em modo texto, com os quais deseja formar um time, ao núcleo do sistema. Assim, os robôs enviam ao núcleo suas informações via objeto *RobotData*.

A Fig. 8 mostra um exemplo de como seria a base de conhecimento, com fatos e uma regra, relacionado ao exemplo apresentado.

```

robo(robo_com_rodas).
time(robo_com_rodas,robo_com_rodas).
forma_time(X, Y) :- robo(X), robo(Y), time(X,Y).

```

Figura 8. Exemplo da base de conhecimento em Prolog.

No exemplo, é criado um predicado para os robôs com rodas (*robo(robo_com_rodas)*). Além disso, um predicado é criado para o time formado pelos dois robôs com rodas (*time(robo_com_rodas, robo_com_rodas)*). Por último, temos uma regra que verifica se os argumentos são ou não são robôs e se formam ou não um time.

Vale salientar que este é um exemplo simples. A base de conhecimento aumenta de acordo com as novas conexões de desenvolvedores e robôs. Um time é formado por um conjunto

finito de robôs e a base de conhecimento, desenvolvida em Prolog, é alimentada dinamicamente.

Logo após isso, o usuário final se conecta ao núcleo do sistema e recebe deste as configurações de times possíveis para os robôs que ele conectou. Essa busca é realizada na base de conhecimento do núcleo. Uma vez que o usuário final seleciona qual time deseja formar, o núcleo imediatamente cria um componente virtual em que o time será executado. Além disso, o núcleo repassa para o time recém criado as conexões de seus membros, bem como suas funções. A Fig. 9 mostra esse processo.

Figura 9. Interação do usuário final com o time de robôs.

Dessa forma, o usuário final passa a comandar o time de robôs via emissão de comandos. Esses comandos são enviados ao núcleo, que os repassa para o time transformá-los em chamadas de funções (métodos) nos robôs membros do time (Ver Fig. 9). Isso ocorre porque estão especificadas nos parâmetros das funções do time, quais funções devem ser chamadas nos robôs, individualmente.

O middleware oferece ao usuário final duas opções para acesso ao time. A primeira opção se trata de uma operação de junção de funções em um bloco de comandos único. Esse bloco é implementado no sistema como uma função criada pelo usuário final, onde ele insere quais funções do time deseja executar em conjunto.

A outra opção se trata da seleção de qual função o usuário final deseja executar imediatamente. Na medida em que ele tem acesso ao time, vai selecionado sequencialmente quais funções deseja executar.

Com esse exemplo, podemos perceber que usuários finais possuem um acesso simplificado a um time de robôs. Isso se deve muito ao fato de a arquitetura proposta fornecer um conjunto bem definido de regras para criação, formação e disponibilização das funções do time.

No entanto, podem haver casos onde algum usuário final conecte um conjunto de robôs que não possuam configuração de time predefinida. Mesmo assim, o núcleo, de posse das informações enviadas pelos robôs conectados, alimenta sua base de conhecimento com essa nova configuração.

Nesse caso em específico, existe uma configuração de times, mas ainda não existe uma configuração das funções desse time. O núcleo, assim que alimenta sua base de conhecimento, envia ao componente de programação de times uma sinal para que algum desenvolvedor possa configurar as funções desse time.

IV. ESTUDO DE CASO

Para validar o modelo de sistema apresentado, escolhemos dois estudos de caso para mostrar o processo de criação, programação e disponibilização de times de robôs distribuídos. No primeiro, um time é formado a partir de dois

robôs que se locomovem com rodas. Já no segundo, um time é formado por um robô com rodas e um robô com garra.

O objetivo é mostrar, passo a passo, como o sistema funciona tanto com a visão do programador de times quanto com a visão do usuário final.

A. Metodologia

Os experimentos foram realizados obedecendo os seguintes passos:

- O programador de times define a configuração dos dois times;
- O núcleo recebe essas configurações e as armazena na base de conhecimento;
- O usuário final conecta os robôs ao sistema, fazendo com que o núcleo verifique se existe alguma configuração que permita a formação de um time com os robôs conectados;
- O núcleo envia ao usuário final uma referência ao time formado. Com isso, o usuário final passa a controlar o time por meio das operações fornecidas pelo sistema.

Para efeito de viabilização dos experimentos, os componentes de programação de times e do usuário final foram desenvolvidos com uma interface simples e em modo texto, como já mencionado. O objetivo nesse momento não é definir um mecanismo padrão de interface com usuários, mas, sim, apresentar o processo de funcionamento do sistema proposto para a formação de times de robôs distribuídos e disponibilização para usuários finais.

Os experimentos foram realizados no laboratório TEAM, localizado no Núcleo de Pesquisa e Inovação em Tecnologia da Informação (NPITI) do Instituto Metrópole Digital (IMD) da Universidade Federal do Rio Grande do Norte (UFRN).

Os dispositivos utilizados na realização dos experimentos foram:

- Dois kits de robôs do modelo EV3 da *LEGO Mindstorms®*.
- Dois adaptadores USB para conexão sem fio da IntelBras® (WBN 900) para conectar os robôs ao núcleo;
- Dois cartões de memória com capacidade de 8GB, onde o *firmware leJOS*, máquina virtual Java para robôs da LEGO, foi instalado;
- Um notebook com processador Intel® Core i5-4210U de 1.7 GHz e com 8GB de memória principal, onde o núcleo do sistema está hospedado.

Cada robô tem suporte a programação em Java e se conecta ao núcleo usando conexões sem fio. Nos experimentos, o formato de um dos robôs é alterado, mas a estrutura de conexão com o núcleo permanece inalterada.

B. Time formado por dois robôs com rodas

O primeiro experimento contempla a utilização de dois robôs com rodas para a formação de um time. A estrutura dos dois robôs é a seguinte:

- **Robôs 1 e 2:** Um controlador EV3 conectado a dois motores, cada qual sendo acoplado a suas respectivas rodas.

A Fig. 10 mostra o formato dos robôs usados no experimento. Os robôs com rodas são idênticos e formam o time usando uma estrutura (barra adaptada) que os interliga. Dessa maneira, a distância lateral entre os dois robôs se torna fixa, facilitando a adoção de estratégias para a movimentação do time sem ter que calcular, constantemente, as posições dos robôs no ambiente, bem como a distância entre eles. No entanto, nada impede que este time possa atuar sem uma estrutura que os interligue fisicamente.

Figura 10. Robôs usados no primeiro experimento.

O programador de times configura o time formado por esses dois robôs usando as descrições genéricas já mencionadas. Nesse primeiro experimento, a descrição genérica usada para os robôs com rodas é mostrada na Fig. 11.

```
interface RoboComRodas{
    void irParaFrente(int segundos, float velocidade);
    void irParaTras(int segundos, float velocidade);
    void virarAEsquerda(int segundos, float velocidade);
    void virarADireita(int segundos, float velocidade);
}
```

Figura 11. Descrição do robô com rodas usado no experimento.

O time formado nesse primeiro experimento recebeu o nome de *time alfa*. Quatro funções foram criadas para este time. As funções *irParaFrente()* e *irParaTras()* estão descritas na Fig. 12.

```
void irParaFrente(int segundos, float velocidade){
    Robo1.irParaFrente(segundos, velocidade);
    Robo2.irParaFrente(segundos, velocidade);
}

void irParaTras(int segundos, float velocidade){
    Robo1.irParaTras(segundos, velocidade);
    Robo2.irParaTras(segundos, velocidade);
}
```

Figura 12. Descrição das funções *irParaFrente()* e *irParaTras()* do *time alfa*.

A função *irParaFrente()* faz com que os dois robôs sejam acionados. Dessa forma, essa função resulta na chamada das funções *irParaFrente()* de cada robô com rodas do *time alfa*. O mesmo princípio de funcionamento se aplica a função *irParaTras()*.

Na Fig. 13, podemos observar a implementação das funções *virarADireita()* e *virarAEsquerda()* do *time alfa*. Nesse caso, onde temos dois robôs com rodas no mesmo time, a ação de virar à direita ou à esquerda deve acontecer de forma

colaborativa. Desenvolvemos, nesse trabalho, duas formas para que o time realize essas ações. Assim, descrevemos aqui como elas foram projetadas, na prática.

```
void virarADireita(int segundos, float velocidade){
    Robo1.virarADireita(segundos, velocidade);
    Robo2.virarADireita(segundos, velocidade);
}

void virarAEsquerda(int segundos, float velocidade){
    Robo1.virarADireita(segundos, velocidade);
    Robo2.virarAEsquerda(segundos, velocidade);
}
```

Figura 13. Descrição das funções *virarADireita()* e *virarAEsquerda()* do *time alfa*.

Usando a primeira estratégia, na ação de para virar à direita, por exemplo, o robô do lado direito se locomoverá com uma velocidade menor do que o robô do lado esquerdo. O resultado disso é um movimento em curva ou arco a direita em torno do eixo do robô do lado direito. Uma representação gráfica dessa estratégia é apresentada na Fig. 14. A mesma lógica de funcionamento é aplicada para a ação de virar à esquerda. Contudo, nesse caso, a diferença ocorre na configuração do sentido e velocidade dos motores dos robôs.

Figura 14. Representação gráfica da primeira estratégia usada para realizar a operação *virarADireita()* do *time alfa*.

Nessa representação, utilizamos uma associação envolvendo valores fixos (constantes) predefinidos e o valor de *velocidade* passado como argumento pelo usuário final para o time. A partir disso, temos a construção de ações colaborativas entre os dois robôs que compõem o time. Para realizar a ação de virar à direita, o time configura o valor de cada motor de cada robô. O primeiro motor do robô a esquerda (representado pela variável *B1*) tem sua velocidade configurada a partir da chamada do método *B1.setSpeed(720*v)*. Esse motor é acionado com a maior velocidade do conjunto. Nesse caso, o valor 720 configura, segundo a API do *leJOS*, a velocidade do motor para 2 RPM (Rotações Por Minuto). Além disso, ele se movimenta para frente com a chamada do método *B1.forward()*.

O segundo motor do primeiro robô (representado pela variável *C1*) também é acionado para ir para frente por meio da chamada do método *C1.forward()*, mas sua velocidade é configurada com um valor menor em relação ao primeiro motor. A chamada ao método *C1.setSpeed(360*v)* faz com que o valor recebido na variável *v* seja multiplicado pelo valor de 360, que representa 1 RPM no *leJOS*.

Em relação ao primeiro motor do segundo robô (variável *B2*), temos uma situação parecida com a do segundo motor do primeiro robô. Ele é acionado com uma velocidade representada pela chamada *B2.setSpeed(360*v)* e se movimenta para frente usando a chamada do método *B2.forward()*. Já com relação ao segundo motor do segundo robô (variável *C2*), temos uma diferença no sentido de sua movimentação. Ele se movimenta para trás (*C2.backward()*) com uma velocidade idêntica a do primeiro motor (*C2.setSpeed(360*v)*). Nessa estratégia, o time gira em torno do eixo do segundo robô (*Robô 2*). Com o primeiro motor desse robô se movendo para frente, a opção satisfatória, em termos de funcionamento, nesse caso, foi a de fazer o segundo motor se mover para trás.

Uma configuração alternativa para esse caso é alterar a velocidade do segundo motor em 2 RPM (*C2.setSpeed(720*v)*). Com esse motor se movendo para trás com uma velocidade maior, observamos que a operação de virar à direita, também, ocorre de maneira satisfatória. Dessa forma, é possível constatar que a movimentação do segundo robô ajuda na movimentação do primeiro, tornando possível realizar uma ação de virar à direita de forma colaborativa.

Para realizar a ação de virar à esquerda, temos um cenário análogo ao da ação de virar à direita. O que muda é a configuração dos valores de velocidade e sentido dos robôs. O motor representado pela variável *B1*, do *Robô 1*, seria configurado com valores de velocidade e sentido idênticos aos do motor representado pela variável *C2*, usados na operação de virar à direita. O resultado seria: *B1.setSpeed(360*v)* e *B1.backward()*. O motor *C2*, por sua vez, seria configurado com valores idênticos aos usados no motor *B1*, usados na operação de virar à direita. O resultado seria: *C2.setSpeed(720*v)* e *C2.forward()*. Assim, temos a operação de virar à esquerda do time em torno do eixo do *Robô 1*.

Assim como ocorre na operação de virar à direita, há também para a operação de virar à esquerda uma possibilidade de configuração alternativa. Se o motor *B1* tiver sua velocidade configurada para 2 RPM (*B1.setSpeed(720*v)*), observamos a realização dessa operação de forma mais rápida.

Na segunda estratégia, projetamos a realização das operações de virar à direita ou à esquerda em torno de um eixo localizado no ponto que divide a distância entre os robôs do time. Como essa distância entre é fixa, por causa da estrutura que os interliga, a opção de realizar essas operações fazendo os dois robôs girarem em torno do mesmo eixo também se torna aplicável. Nesse caso, para virar à direita, colocamos o *Robô 1* para se movimentar para frente e o *Robô 2* para se movimentar para trás. Na Fig. 15, podemos observar a representação gráfica dessa operação.

Na representação da segunda estratégia, o primeiro motor do *Robô 1* (*B1*) tem sua velocidade e seu sentido de movimentação definidos pelas chamadas aos métodos *B1.setSpeed(360*v)* e *B1.forward()*, respectivamente. As mesmas linhas de código são aplicadas ao segundo motor do *Robô 1* (*C1*). Assim, temos o *Robô 1* se movendo para frente. Já o *Robô 2*, se move para trás. Assim sendo, o motor *B2* tem sua velocidade e seu sentido de movimentação definidos pelas chamadas aos métodos *B2.setSpeed(360*v)* e *B2.backward()*, respectivamente. E o motor *C2* também tem sua velocidade e seu sentido de movimentação definidos pelas chamadas aos

métodos *C2.setSpeed(360*v)* e *C2.backward()*. Uma configuração alternativa seria aumentar para 2 RPM as velocidades do motor *B1* do *Robô 1* (*B1.setSpeed(720*v)*) e do motor *C2* do *Robô 2* (*C2.setSpeed(720*v)*). Com isso, a operação é realizada com uma rapidez maior.

Figura 15. Representação gráfica da segunda estratégia usada para realizar a operação *virarADireita()* do time *alfa*.

Podemos observar que na segunda estratégia, as velocidades dos motores são idênticas. O que é determinante para que a operação de virar à direita seja realizada é a movimentação dos robôs em sentidos contrários. Essa lógica também é aplicada para a operação de virar à esquerda. Contudo, para tanto, o *Robô 1* deve se movimentar para trás (*B1.setSpeed(360*v)*, *B1.backward()*, *C1.setSpeed(360*v)* e *C1.backward()*) e o *Robô 2* deve se movimentar para frente (*B2.setSpeed(360*v)*, *B2.forward()*, *C2.setSpeed(360*v)* e *C2.forward()*).

Com as duas estratégias apresentadas o sistema ganha em flexibilidade na realização das operações citadas em um times de robôs com rodas. As duas poderiam ser usadas para o caso, por exemplo, do time possuir algum sensor ou câmera para calcular a distância dele para os objetos e definir qual a melhor forma de virar à direita ou à esquerda.

Vale salientar também que, os valores testados para a configuração das velocidades dos motores estão multiplicados pela variável *v* (abreviação do parâmetro *velocidade* usado nas funções do time). Essa variável recebe um valor de velocidade escolhido pelo usuário final. Nos testes, trabalhamos com duas modalidades de velocidade que podem ser escolhidas pelo usuário final. A velocidade representada pela constante *NORMAL* recebe, internamente o valor 1. E a velocidade representada pela constante *RÁPIDO* recebe, internamente o valor 2. Com isso, os valores de velocidade aplicados internamente pelo sistema aos motores dos robôs podem variar de acordo com essas opções, mas sempre partindo dos valores-base estabelecidos nos testes.

O primeiro experimento mostra que cada função do time, ao ser acessada, resulta no acionamento dos dois robôs. Nesse caso, as funções chamadas nos robôs são idênticas. O que muda são os valores dos parâmetros para alguns casos. Nada impede que outras funções sejam criadas e que agrupem chamadas às funções distintas nos robôs membros do time.

C. Time formado por um robô com rodas e um robô com garra

O segundo experimento contempla a utilização de um robô com rodas e um robô com garra para a formação de um time.

A estrutura dos dois robôs é a seguinte:

- **Robô 1:** Um controlador EV3 conectado a um motor que, por sua vez, é acoplado a uma estrutura em forma de garra.
- **Robô 2:** Um controlador EV3 conectado a dois motores, cada qual sendo acoplado as suas respectivas rodas.

A Fig. 16 mostra o formato dos robôs usados no experimento. Nesse caso, os robôs foram agrupados na mesma estrutura física para que o robô garra pudesse se movimentar com a ajuda do robô com rodas.

Figura 16. Robôs usados no segundo experimento.

Para este experimento, a descrição usada para o robô com garra é mostrada na Fig. 17. Para o robô com rodas, a descrição usada foi a da Fig. 11.

```
interface RoboComGarra{
    void pegarObjeto(int segundos, float velocidade);
    void soltarObjeto(int segundos, float velocidade);
}
```

Figura 17. Descrição do robô com garra usado no experimento.

Nesse experimento, o time formado por esses dois robôs recebe o nome de *time beta*. Seis funções foram criadas para este time. As funções *irParaFrente()* e *irParaTras()* estão descritas na Fig. 18.

```
void irParaFrente(int segundos, float velocidade){
    Robo2.irParaFrente(segundos, velocidade);
}

void irParaTras(int segundos, float velocidade){
    Robo2.irParaTras(segundos, velocidade);
}
```

Figura 18. Descrição das funções *irParaFrente()* e *irParaTras()* do *time beta*.

Para que esse time se movimente para frente ou para trás é preciso que o robô com rodas seja acionado. Dessa forma, na definição dessas funções para o *time beta*, somente o *Robô 2* é acionado, por meio de sua função *irParaFrente()*.

A Fig. 19 mostra a descrição das funções *virarADireita()* e *virarAEsquerda()*. Mais uma vez, podemos perceber que nessas duas funções do *time beta*, somente um dos robôs é acionado (*Robô 2*). Contudo, as ações de virar à direita ou à esquerda exigem que os motores do *Robô 2* atuem de forma colaborativa.

```
void virarADireita(int segundos, float velocidade){
    Robo2.virarADireita(segundos, velocidade);
}

void virarAEsquerda(int segundos, float velocidade){
    Robo2.virarAEsquerda(segundos, velocidade);
}
```

Figura 19. Descrição das funções *virarADireita()* e *virarAEsquerda()* do *time beta*.

Nesse caso, utilizamos uma estratégia similar a segunda estratégia usada no experimento anterior. A diferença é que para realizar essas operações, nesse experimento, trabalhamos com um robô somente. Então, para o time virar a direita é necessário que o motor do lado direito acione a roda para se mover para trás e o do lado esquerdo acione a roda para se mover para frente. Essa ação conjunta é realizada por esta função.

A Fig. 20 mostra uma representação gráfica da operação *virarADireita()* do *time beta*. No *Robô 2*, o motor representado pela variável *B* tem a velocidade configurada para 1 RPM (*B.setSpeed(360*v)*) como valor base. Ela pode aumentar de acordo com o valor da variável *v*, que, assim como no experimento anterior, representa o valor passado pelo usuário final como argumento para velocidade dessa operação do time. Os modos de velocidade que podem ser usados pelo usuário final também são os mesmos em relação ao experimento anterior (*NORMAL* = 1, *RÁPIDO* = 2). O sentido de movimentação do motor *B* é configurado pela chamada ao método *B.forward()*.

Figura 20. Representação gráfica da estratégia usada para realizar a operação *virarADireita()* do *time beta*.

Já em relação ao motor *C*, para completar a operação, a configuração é a seguinte: velocidade de 1 RPM (*C.setSpeed(360*v)*) e sentido de movimentação para trás (*C.backward()*).

Para realizar a operação *virarAEsquerda()*, o programador do time configura o motor *B* com uma velocidade de 1 RPM (*B.setSpeed(360*v)*) e para se mover para trás (*B.backward()*). E, o motor *C* é configurado, também, com velocidade de 1 RPM (*C.setSpeed(360*v)*) e se movendo para frente (*C.forward()*).

As duas últimas funções do *time beta* são apresentadas na Fig. 21. A função *pegarObjeto()* do *time beta* aciona somente o robô com a garra (*Robô 1*). A mesma coisa acontece para a função *soltarObjeto()*. As duas recebem como parâmetros um valor da quantidade de tempo para aquela ação ser realizada e a velocidade, que serve para acionar o motor ligado a garra. A

velocidade de movimentação do motor ligado a garra também foi configurada para 1 RPM, podendo variar de acordo com a variável v , assim como nas operações anteriores.

```
void pegarObjeto(int segundos, float velocidade){
    Robo1.pegarObjeto(segundos, velocidade);
}

void soltarObjeto(int segundos, float velocidade){
    Robo1.soltarObjeto(segundos, velocidade);
}
```

Figura 21. Descrição das funções *pegarObjeto()* e *soltarObjeto()* do *time beta*.

No segundo experimento, podemos observar que o resultado da execução de uma função de um *time* pode ser o acionamento de um dos robôs que o compõe. O desenvolvedor de *times* é livre para programar as ações que desejar da forma que desejar.

A Fig. 22 mostra a base de conhecimento com informações relativas aos experimentos realizados. Temos dois robôs cadastrados, dois predicados descrevendo quais robôs formam um *time* e uma regra, que verifica se os argumentos enviados como parâmetros são robôs e formam um *time*.

```
robo(robo_com_rodas).
robo(robo_com_garra).

time(robo_com_rodas,robo_com_rodas).
time(robo_com_rodas,robo_com_garra).

forma_time(X, Y) :- robo(X), robo(Y), time(X,Y).
```

Figura 22. Base de conhecimento formada com os experimentos.

Como já mencionado, *times* podem ser formados por um conjunto finito de robôs. Com isso, os predicados que informam quais robôs formam *times* podem ter mais de dois parâmetros, fazendo com que as regras que avaliam se determinados robôs formam um *time* também sejam criadas para mais de dois parâmetros.

V. RESULTADOS E DISCUSSÃO

Antes de mais nada, é preciso descrever algumas questões de implementação. Inicialmente, projetamos uma estratégia para a formação e gerenciamento de *times*. Ela se baseia na escolha de um líder entre os robôs membros de um *time* para que este pudesse se comunicar com o núcleo. Ou seja, a responsabilidade de receber uma chamada para uma função do *time* e transformá-la em subcomandos para os membros do *time* seria de um dos próprios membros.

Entretanto, essa estratégia apresenta um número maior de troca de mensagens entre os robôs tanto na eleição de um líder quanto nas comunicações para envio de subcomandos ao robôs. Além disso, o risco de falhas se torna alto pelo fato de robô líder ter restrições de energia.

Dessa forma, projetamos a execução dos *times* como entidades virtuais situadas no núcleo pelo fato deste desempenhar funções importantes, como já pôde ser observado. Essa decisão implica na diminuição do risco de falha, pois o núcleo deve ter um nível de disponibilidade alto. Além disso, as informações acerca de um *time* são centralizadas em uma entidade virtual, diminuindo latências de comunicação com outros processos.

Mesmo assim, o sistema não está livre do risco de falha por queda de energia ou problemas na rede. Contudo, a estratégia de executar o *time* como uma entidade virtual se tornou mais viável nesse momento.

Os experimentos realizados tiveram o objetivo de apresentar, para dois exemplos concretos, o processo de criação, programação e disponibilização de *times* de robôs distribuídos. O resultado desse processo foi a abertura de um canal pelo qual usuários finais podem programar ações para *times* de robôs de uma forma simples, sem a obrigação de serem especialistas em robótica ou programação.

Simultaneamente, os experimentos mostraram algumas características positivas do sistema proposto, como: padronização das comunicações entre os componentes do sistema por meio do objeto *RobotData* transformando em JSON, ocultação do usuário final do processo de gerenciamento dos *times* de robôs, flexibilidade quanto as configurações e formatos que *times* de robôs podem ter, programação de *times* por meio de associação de funções, além da padronização de regras para usuários finais acessarem *times* de robôs.

Podemos listar também alguns aspectos que podem ser aprimorados no sistema como: integração de robôs com *hardwares* heterogêneos, desenvolvimento de estratégias para tratamento de falhas no gerenciamento dos *times* de robôs e aperfeiçoamento da interface com o usuário final.

Outra característica importante é que o sistema proposto pode aprender sobre configurações de *times* tanto a partir do programador de *times* quanto a partir do usuário final. Por exemplo, se um usuário final desejar criar um *time* com três robôs (dois robôs com rodas e um robô com garra) e o núcleo do sistema constatar a inexistência dessa configuração na base de conhecimento, o núcleo armazena essa nova configuração na base de conhecimento e envia ao usuário final uma configuração de *time* que mais se aproxime da que ele desejava criar. Nesse caso, ele iria enviar a configuração do *time beta* (um robô com rodas e um robô com garra).

Posteriormente, quando algum programador de *times* se conectar ao sistema, o núcleo envia uma notificação sobre a necessidade de criar funções para um *time* contendo dois robôs com rodas e um robô com garra. Com isso, este *time* passa a ter uma configuração completa no sistema.

Além disso, o sistema especifica duas operações especiais para que usuários finais acessem o *time* de robôs: execução de funções do *time* por meio de um bloco de comandos único ou seleção imediata e sequencial de funções para execução. Uma implementação simples (modo texto) foi desenvolvida para viabilizar a realização dos experimentos. No entanto, há margem para a especificação de novas operações, bem como a implementação das que já existem usando outros mecanismos de interação de usuários finais com o sistema.

VI. CONSIDERAÇÕES FINAIS E PROPOSTAS DE TRABALHOS FUTUROS

Esse trabalho aborda o problema da programação e colaboração de robôs distribuídos propondo uma arquitetura de sistema que permita a criação de *times* de robôs de forma

direta e predefinida, além de disponibilizá-los aos usuários finais de forma simplificada.

Uma característica importante desse sistema é o aprendizado coletivo. As configurações de times podem ser enviadas tanto pelo programador de times quanto pelo usuário final. Quando esse último conecta robôs ao sistema e o núcleo verifica que não há nenhuma configuração de time cadastrada para esses robôs, o mesmo armazena aquela configuração e envia a configuração mais aproximada daquela solicitada pelo usuário final.

Esse fato faz com que o sistema se adapte ao contexto rapidamente. Quando algum programador de times se conectar ao núcleo, este envia essa configuração para que o programador especifique as funções desse time.

Os experimentos mostraram resultados satisfatórios. O processo de funcionamento do sistema para a formação e disponibilização de times é apresentada de maneira ordenada, onde é possível observar que cada parte possui um conjunto de funcionalidades específicas, mas complementares. Além do mais, os times de robôs criados apresentaram funcionamento desejado, correspondente ao seu *hardware* de fabricação.

Como propostas de trabalhos futuros, estudamos a possibilidade usar as tecnologias de *web services* e nuvens computacionais no desenvolvimento dos componentes do sistema, além de realizar a integração de robôs de *hardwares* de fabricação diferentes e outros dispositivos como sensores, *smartphones*, *tablets*, minicomputadores de baixo custo.

Além disso, projetamos a criação de operações especiais de comunicação coletiva entre os robôs para permitir que eles possam coordenar suas ações na execução das funções do time.

Existe muito espaço, também, para pesquisas em mecanismos para interface com usuários finais. Estudar quais técnicas seriam mais adequadas para usuários finais de diferentes faixas etárias acessarem e programarem, em alto nível, times de robôs distribuídos se coloca como uma tarefa bem interessante e que pode auxiliar projetos realizados no grupo TEAM da UFRN.

REFERÊNCIAS

- [1] G. C. Fernandez, S. M. Gutierrez, E. S. Ruiz, F. M. Perez and M. C. Gil, "Robotics, the New Industrial Revolution," in *IEEE Technology and Society Magazine*, vol. 31, no. 2, pp. 51-58, June 2012.
- [2] A. M. Zanchettin, N. M. Ceriani, P. Rocco, H. Ding and B. Matthias, "Safety in Human-Robot Collaborative Manufacturing Environments: Metrics and Control," in *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2, pp. 882-893, April 2016.
- [3] M. S. Nosrati, R. Abugharbieh, J. M. Peyrat, J. Abinayed, O. AlAlao, A. Al-Ansari, and G. Hamarneh, "Simultaneous multi-structure segmentation and 3d nonrigid pose estimation in image-guided robotic surgery," *IEEE Transactions on Medical Imaging*, vol. 35, no. 1, pp. 1-12, Jan 2016.
- [4] Z. Dogmus, E. Erdem, and V. Patoglu, "React!: An interactive educational tool for ai planning for robotics," *IEEE Transactions on Education*, vol. 58, no. 1, pp. 15-24, Feb 2015.
- [5] T. S. Tadele, T. de Vries, and S. Stramigioli, "The safety of domestic robotics: A survey of various safety-related publications," *IEEE Robotics Automation Magazine*, vol. 21, no. 3, pp. 134-142, Sept 2014.
- [6] T. J. Wiltshire and S. M. Fiore, "Social cognitive and affective neuroscience in human-machine systems: A roadmap for improving training, human-robot interaction, and team performance," *IEEE Transactions on Human-Machine Systems*, vol. 44, no. 6, pp. 779-787, Dec 2014.
- [7] K. Zheng, D. F. Glas, T. Kanda, H. Ishiguro, and N. Hagita, "Designing and implementing a human-robot team for social interactions," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 4, pp. 843-859, July 2013.

- [8] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 398-409, April 2015.
- [9] A. Kolling, P. Walker, N. Chakraborty, K. Sycara, and M. Lewis, "Human interaction with robot swarms: A survey," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 1, pp. 9-26, Feb 2016.
- [10] C. S. Karagöz, H. I. Bozma, and D. E. Koditschek, "Coordinated navigation of multiple independent disk-shaped robots," *IEEE Transactions on Robotics*, vol. 30, no. 6, pp. 1289-1304, Dec 2014.
- [11] H. Rezaee and F. Abdollahi, "A decentralized cooperative control scheme with obstacle avoidance for a team of mobile robots," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 1, pp. 347-354, Jan 2014.
- [12] F. Arrichiello, A. Marino, and F. Pierri, "Observer-based decentralized fault detection and isolation strategy for networked multirobot systems," *IEEE Transactions on Control Systems Technology*, vol. 23, no. 4, pp. 1465-1476, July 2015.
- [13] A. Yazici, G. Kirlik, O. Parlaktuna, and A. Sipahioglu, "A dynamic path planning approach for multirobot sensor-based coverage considering energy constraints," *Cybernetics, IEEE Transactions on*, vol. 44, no. 3, pp. 305-314, March 2014.
- [14] S. Han, M. sook Kim, and H. S. Park, "Open software platform for robotic services," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 3, pp. 467-481, July 2012.
- [15] W. Woodall, M. Liebhardt, D. Stonier, and J. Binney, "Ros topics: Capabilities [ros topics]," *IEEE Robotics Automation Magazine*, vol. 21, no. 4, pp. 14-15, Dec 2014.
- [16] L. Riazuelo, M. Tenorth, D. D. Marco, M. Salas, D. Gálvez-López, L. Mösenlechner, L. Kunze, M. Beetz, J. D. Tardós, L. Montano, and J. M. M. Montiel, "Roboearth semantic mapping: A cloud enabled knowledge-based approach," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 432-443, April 2015.
- [17] Java, "Java software - oracle," 2016. [Online]. Available: <https://www.oracle.com/java/index.html>.
- [18] JPL, "JPL a interface to prolog," 2016. [Online]. Available: http://www.swi-prolog.org/packages/jpl/java_api/
- [19] D. Crockford, "The application/json media type for javascript object notation (json)," IETF, RFC 4627, 7 2006.

Paulo Henrique Lopes Silva possui graduação em Ciência da Computação pela Universidade do Estado do Rio Grande do Norte (2007) e mestrado em Ciência da Computação pela Universidade Federal do Ceará (2009). Atualmente é doutorando no Programa de Pós-Graduação em Engenharia Elétrica e de Computação (PPGEEC) da Universidade Federal do Rio Grande do Norte (UFRN) e é Professor Adjunto da Universidade Rural do Semiárido (UFERSA) desde 2010. Tem experiência em Computação, com ênfase em modelos, programação e aplicações para sistemas de computação paralela e distribuída.

Aquiles Medeiros Filgueira Burlamaqui possui graduação em Ciências da Computação pela Universidade Federal do Rio Grande do Norte (2003) e mestrado em Sistemas e Computação pela Universidade Federal do Rio Grande do Norte (2004). Doutor pela Universidade Federal do Rio Grande do Norte, professor da Universidade do Estado do Rio Grande do Norte e pesquisador do Laboratório Natalnet-DCA, atuando principalmente nos seguintes temas: Ambientes Virtuais Colaborativos Massivos, TV Digital Interativa, Sistemas Web, Aplicações de Robótica, Realidade Virtual e Engenharia de Software.