Programando Robôs Móveis Utilizando a API ARIA

Luan Rios Campos[1], **Matheus Giovanni Pires**[1]

[1]Departamento de Ciências Exatas – Universidade Estadual de Feira de Santana (UEFS)
Caixa Postal 44.036-970 – Feira de Santana – BA – Brasil

luan.rios.campos@gmail.com, mgpires@ecomp.uefs.br

Abstract. *This paper shows the code development process for autonomous mobile robots using the API ARIA. ARIA is really useful, since it allows the developer to utilize many resources from the robot, like: speed control, positioning, usage of range devices (sonars and lasers). Besides, it is possible to implement own actions (threads) for the robot or make it exchange data with other applications using TCP socket without regard for low level details.*

Resumo. *Este artigo tem como objetivo mostrar o processo de desenvolvimento de códigos para robôs móveis autônomos utilizando a API ARIA. A ARIA é bastante versátil, permitindo que o desenvolvedor utilize vários recursos do robô, como controle de velocidade, posicionamento, utilização de dispositivos de alcance (sonares e lasers). Além disso, é possível implementar ações (threads) próprias para o robô ou fazê-lo trocar dados com outra aplicação através de soquetes TCP sem que exista a preocupação com detalhes em baixo nível.*

1. Introdução

Sistemas capazes de automatizar processos com o objetivo de preservar a integridade humana apresentam um apelo muito interessante do ponto de vista científico. Existem várias tarefas do mundo real que oferecem perigo à integridade humana, tais como, inspeção interna de dutos de transporte de petróleo, limpeza de sistemas de ar-condicionado, a pulverização de inseticidas em estufas [MANDOW et al. 1996], detecção e salvamento de pessoas soterradas em escombros de estruturas danificadas [MURPHY 2000], etc. Neste contexto, os robôs móveis autônomos podem realizar tarefas que seriam impossíveis e/ou perigosas para os humanos [BAY 1995].

Em relação à navegação de robôs móveis, várias tarefas específicas são essenciais. O robô deve ser capaz de sensorear seu ambiente e construir uma representação local que seja suficiente em detalhes e precisão [TRONCO and PORTO 2005] para a sua locomoção, de forma precisa e suave, ao mesmo tempo em que permite reações rápidas às mudanças do ambiente para evitar colisões, garantindo que o destino seja atingido de forma satisfatória [FRACASSO and COSTA 2005]. O robô deve também ser capaz de se localizar, ou seja, de determinar sua posição e orientação no ambiente e mapear esta informação em sua representação local e de sua vizinhança [TRONCO and PORTO 2005].

Na realização de tais tarefas são necessárias formas de interação e programação entre o robô e o ser humano, as quais são discutidas em diversas pesquisas que evidenciam a necessidade de um ambiente de desenvolvimento de aplicações para robôs com

alto nível de abstração e que possua uma interface amigável [SCATENA 2008]. Dessa forma, a API[1] ARIA, fornece uma interface para controlar e receber dados de todas as plataformas de robôs da MobileRobots, assim como de vários dispositivos acessórios [MobileRobots 2013b], provendo um ambiente que acelera a programação de robôs móveis, já que não é necessária a preocupação com detalhes de baixo nível. Dessa forma, este trabalho tem por objetivo apresentar as funcionalidades básicas da API ARIA para a programação de robôs móveis.

Este artigo está organizado da seguinte forma: na seção 2, será mostrado a arquitetura e o ambiente de desenvolvimento para robôs móveis, como ferramentas e métodos de desenvolvimento utilizando a API. Na seção 3, será explicada algumas funcionalidades da API e como ela provê comunicação entre o robô e o simulador. Na seção 4, além de apresentar sobre o uso de dispositivos no robô através da API, suas funcionalidades serão explicadas de forma mais detalhada. Na seção 5, será explicado como realizar conexão entre uma aplicação cliente e uma aplicação servidor e como enviar e receber dados por essa conexão.

2. Passos Básicos para Programação de Robôs Móveis

Esta seção está dividida em subseções que irão explicar o que é necessário para iniciar a programação dos robôs móveis da MobileRobots, tais como, a arquitetura de um sistema que utiliza a API ARIA e o ambiente de simulação necessário para testar a aplicação desenvolvida.

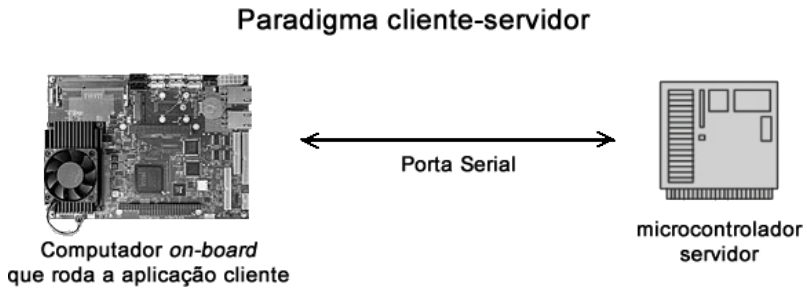
2.1. Arquitetura do Sistema

A API ARIA é baseada na arquitetura cliente-servidor, onde os detalhes de baixo nível dos robôs móveis são administrados por servidores encorpados no sistema operacional do micro-controlador presente no robô [ActivMedia 2005]. O cliente é responsável pelo controle em alto-nível, deve ser executado em um computador conectado ao micro-controlador. Esse computador pode ser tanto *on-board* (Figura 1(a)), cuja comunicação é feita diretamente com o robô via cabo serial, ou em um computador *off-board* (Figura 1(b)), cuja comunicação é feita remotamente [WHITBROOK 2010]. Essa comunicação entre o servidor e o cliente é gerenciado pela classe ArRobot, que é responsável pela construção, envio e decoficação dos pacotes de informações entre eles.

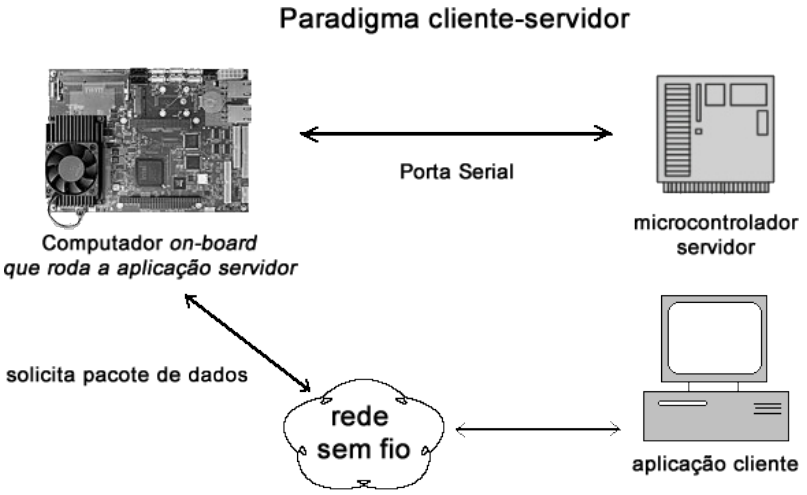
É possível programar robôs móveis com a API ARIA utilizando três diferentes linguagens: C++, linguagem ”nativa” da API, Java e Python, sendo que para as duas últimas linguagens é necessária a instalação de seus respectivos *wrappers*[2]. É possível realizar diversas ações utilizando a ARIA, tais como, controlar e recuperar informações sobre o posicionamento, velocidade e dispostivos instalados no robô (sonares, infravermelhos), configurar o uso de *joysticks* e outros acessórios (câmeras, braços), etc. A API, ainda, inclui muitas utilidades multi-plataformas para redes soquete, ferramentas para *threads*, comunicação através de portas seriais, execução de sons, conversão de coordenadas GPS, entre outras funções [MobileRobots 2013b].

[1]API, de Application Programming Interface, ou em Português, Interface de Programação de Aplicativos.

[2]Padrão de projeto que permite utilizar códigos em uma linguagem através de outra.



(a) Paradigma através de um computador *on-board*



(b) Paradigma através de um computador *off-board*

Figura 1. Arquitetura cliente-servidor usando a API ARIA.

2.2. Simulador

A MobileRobots fornece um simulador (MobileSim) que permite *debug* e experimentação dos códigos feitos utilizando a ARIA antes de serem utilizados em um robô real.

O MobileSim provê a simulação de um controle de conexão que é acessível via porta TCP[3], de forma similar à porta serial presente no robô. Dessa forma, a API é capaz de, automaticamente, conectar-se à porta TCP ao invés da porta serial, o que torna prático usar o mesmo programa, sem necessitar de alterações, tanto no simulador, quanto no robô real [MobileRobots 2013b]. Ele também consegue simular com exatidão os sonares ou infravermelhos instalados no robô e permite que o desenvolvedor acompanhe o caminho percorrido pelo robô no mapa.

2.3. Construção de Mapas

A navegação do robô no simulador é realizada em mapas que podem ser construídos pelo próprio usuário. O mapa pode ser construído tanto através da edição de arquivos de texto que possuem as coordenadas dos obstáculos e cuja extensão é .map, quanto através do uso do *software* Mapper3Basic, o qual é fornecido pela própria MobileRobots.

O Mapper3Basic, por sua vez, permite adicionar ou editar obstáculos visíveis aos sensores do robôs, como, por exemplo, muros. Além disso, é possível determinar o ponto

[3]Transmission Control Protocol

de partida e chegada do robô [MobileRobots 2013b].

3. Estabelecendo Conexão entre Cliente e Servidor

Conforme foi dito na seção 2, a ARIA fornece um conjunto de funcionalidades para o desenvolvimento de aplicações de controle e monitoramento do robô que permite ao programador uma abstração em alto nível do robô físico, cabendo ao desenvolvedor implementar os comportamentos desejados, como por exemplo, andar, virar à esquerda, aumentar a velocidade, parar, entre outras funções. Na Figura 2 há um código básico para conectar um robô ao simulador.

```
1 int main(int argc, char** argv) {
2
3     Aria::init();
4     ArArgumentParser parser(&argc, argv);
5     parser.loadDefaultArguments();
6     ArRobot robot;
7
8     ArRobotConnector robotConnector(&parser, &robot);
9
10    if (!robotConnector.connectRobot()) {
11        ArLog::log(ArLog::Terse,
12                "actionExample: Could not connect to the robot.");
13        if (parser.checkHelpAndWarnUnparsed()) {
14            Aria::logOptions();
15            Aria::exit(1);
16        }
17    }
18
19    if (!Aria::parseArgs() || !parser.checkHelpAndWarnUnparsed()) {
20        Aria::logOptions();
21        Aria::exit(1);
22    }
23
24}
```

Figura 2. Código inicial para conexão do robô

É de fundamental importância para o funcionamento de todos os métodos das outras classes da ARIA, como conexão do robô, habilitação de motores ou conexão de dispositivos, que a API seja inicializada através do método *init()* da classe **Aria** antes de qualquer código que a referencia e que esteja dentro da função de execução da aplicação (Figura 2, linha 3). Isto se deve ao fato de que esse método inicia a estrutura global de dados da API, além de realizar uma inicialização específica do sistema operacional onde o código está sendo desenvolvido, como adiocionar *handlers* de sinais no Linux ou carregar bibliotecas de soquete no Windows [MobileRobots 2013a].

A classe **ArArgumentParser** (linha 4) age como um *parser* dos parâmetros recebidos via linha de comando pelo método *main()* enquanto que a classe **ArRobot** (linha 6) é a responsável por instanciar o objeto que fará a comunicação com o robô através de envio de comandos ou recuperação de dados (incluindo odometria das rodas, entradas digitais ou analógicas, dados do sonar, entre outros). Essa classe também é usada para prover acesso ao objetos que controlam os acessórios instalados [MobileRobots 2013a].

A conexão do robô é feita através da classe **ArRobotConnector**. Conforme pode ser visto na figura 2, primeiro os objetos do *parser* e do robô são instanciados (linhas 4

e 6) e após isso instancia-se o objeto responsável pela conexão (robotConnector) (linha 8). A conexão propriamente dita é feita, entretanto, através do método *connectRobot()* (linha 10), o qual utiliza o robô que foi determinado durante a criação do conector. A conexão é realizada tanto através de uma porta TCP (para o simulador ou computador *offboard*), quanto diretamente através da porta serial do robô. Normalmente, a primeira tentativa de conexão é com o simulador através da porta TCP, caso o simulador não esteja em execução, tenta-se conectar utilizando a porta serial.

4. Movimentos do Robô e Utilização de Dispositivos de Alcance

Os pacotes de dados utilizados na comunicação entre servidor e clientes são caracterizados em pacotes de informação do servidor (PIS) que são enviados do servidor ao cliente com informações sobre o robô e seus acessórios. Os pacotes de comando que são enviados do cliente para o servidor através da conexão que há com o robô, utilizando funções simples de movimento ou ações (*actions*), e servem para o cliente determinar a ação que o robô deve executar.

Os robôs funcionam a partir de um ciclo síncrono de tarefas determinado pela classe **ArRobot**. O PIS padrão é enviado para esse ciclo constante e a recepção desses pacotes dispara uma nova interação no ciclo síncrono de processamento de tarefas da ArRobot. O ciclo em questão consiste em uma série de tarefas, incluindo pacotes para manuseio do PIS, invocação de tarefas para interpretação dos sensores, manuseadores e resolvedores de *actions*, entre outros, que são acionadas em uma ordem previsível por conta de o ciclo (normalmente) ser ativado a cada PIS recebido [MobileRobots 2013a].

Existem dois métodos da API para iniciar o ciclo de comandos em um robô: *runAsync()* e *run()*. O *runAsync* inicia o ciclo em uma nova *thread* de plano de fundo e é utilizado para definir movimentos simples ao robô. Para que os dados do robô não sejam modificadas por outras *threads*, é preciso que utilizar os métodos *lock()* e *unlock()*, para, respectivamente, travar o robô antes da chamada de algum método ou alteração de dados, e destravá-lo para executar os comandos que foram determinados (ver Figura 3).

```
1 robot->runAsync(true);
2
3 ArUtil::sleep(500);
4
5 robot->lock();
6 robot->setVel(500);
7 robot->move(1000);
8 robot->enableMotors();
9 robot->unlock();
```

Figura 3. Código de movimento simples utilizando o método *runAsync()*

Utilizando o *runAsync*, a não ser que os comandos estejam em uma estrutura de repetição, o robô executará o que foi determinado apenas uma vez. Isso acontece porque, como pode ser visto na Figura 3 nas linhas 6 e 7, comandos como o *move()* ou o *setVel()* são explícitos, indicando diretamente o que o robô deve fazer sem se preocupar com outros detalhes, como análise do ambiente para determinar o quanto a velocidade deve ser alterada ou o quanto o robô deve andar sem se preocupar em atingir um obstáculo, por exemplo.

O método *run* (Figura 4, linha 1), por sua vez, não inicia um novo ciclo, ele entra de forma síncrona no ciclo atual e é utilizado para tornar possível definir comportamentos mais complexos e independentes ao robô, permitindo usar um sistema baseado em *actions*. As *actions* são objetos que solicitam um determinado movimento ao robô. A avaliação dessas solicitações ocorre em ordem decrescente de prioridade (primeiro os de alta prioridade) e são combinadas a cada ciclo para produzir um conjunto final de movimentos [MobileRobots 2013a]. A taxa de prioridade avaliada na lista de *actions* é determinada quando se adiciona uma ação ao ciclo do robô através do método *addAction()* (ver Figura 4).

```
1 robot.run(true);
2 robot.enableMotors();
3
4 ArActionStallRecover recover;
5 ArActionBumpers bumpers;
6 ArActionAvoidFront avoidFrontNear("Avoid Front Near", 225, 0);
7 ArActionAvoidFront avoidFrontFar;
8 ArActionConstantVelocity constantVelocity("Constant Velocity", 400);
9
10 robot.addAction(&recover, 100);
11 robot.addAction(&bumpers, 75);
12 robot.addAction(&avoidFrontNear, 50);
13 robot.addAction(&avoidFrontFar, 49);
14 robot.addAction(&constantVelocity, 25);
```

Figura 4. Código de movimento através de *actions* fornecidas pela API

Como pode ser visto na Figura 4, as ações são objetos, mas cada ação pertence a uma classe distinta. As *actions* são determinadas, de forma geral, pela superclasse **ArAction**, essa classe, porém, não pode ser instanciada. Sendo assim, cada *action* existente na figura, para se comportar como ação, herda da classe **ArAction**. Assim, é possível sobrecarregar (*overload*) o método *fire()* e determinar o comportamento que a *action* deve possuir (ver Figura 5).

O ”desejo” que o robô tem ao realizar determinada ação é controlado pela classe **ArActionDesired**. Os objetos dessa classe são usados para passar os valores dos canais de ação desejadas e sua respectiva força para o *resolver*. Esses canais podem ser usados para alterar a velocidade de movimento do robô, por exemplo.

```
1 ArActionDesired *ActionMove::fire(ArActionDesired currentDesire) {
2
3     robotDesire.reset();
4     robotDesire.setVel(500);
5     return &robotDesire;
6 }
```

Figura 5. Método *fire* para alterar a velocidade do robô

O movimento independente do robô requer alguns dispositivos de leitura do ambiente para determinar seu comportamento. A depender da distância que o robô esteja de um determinado objeto, ele pode movimentar-se em velocidade máxima ou dosar sua velocidade de forma a evitar que atinja o obstáculo, por exemplo.

Os dispositivos de alcance são conectados a uma instância específica da **ArRobot** para obter posicionamento e outras informações do robô quando as leituras dos dispositivos são recebidas e armazenadas, além de ser uma forma de o robô encontrar todas os dispositivos que estão instalados no robô. A API conta com a classe **ArRangeDevice**, a qual é uma abstração dos sensores que detectam a presença de obstáculos no espaço ao redor do robô, provendo uma série de leituras espaciais. Dentre as subclasses da **ArRangeDevice**, duas serão explicadas: a **ArSonarDevice** e **ArLaser**.

ArSonarDevice é a classe que representa o sonar do robô e armazena o histórico de leituras desse dispositivo. O sonar é usado para detectar obstáculos no caminho do robô através da leitura do ambiente a partir da superfície do disco transdutor do sonar. A conexão entre o sonar e o robô pode ser feita de duas formas: a partir do **ArRobot** ao adicionar um dispositivo de alcance através do método *addRangeDevice()*, ou através da **ArSonarDevice** ao definir qual a instância da **ArRobot** o dispositivo pertence pelo método *setRobot()*.

Em *actions* que necessitam a utilização de um sonar, costuma-se usar um objeto do tipo **ArRangeDevice**. Na *thread* principal da aplicação (método *main*) é instanciado e adicionado um **ArSonarDevice** ao robô através do **ArRobot**. Dentro da classe responsável pela ação deve existir um ponteiro **ArRangeDevice**. Ao definir o robô que realizará a *action*, é feita uma busca pelo dispositivo desejado, nesse caso, o sonar (Figura 6, linha 4). Com o dispositivo definido, é possível recuperar os valores de leitura que ele possui através do método *currentReadingPolar()* e, a partir daí, determinar como o robô se comportará. Esse método de recuperação de leitura pertence à **ArRangeDevice** e recupera as informações referentes a uma determinada região polar, a partir de valores inicial e final de ângulos (Figura 6, linha 6).

```
1 void ActionMove::setRobot(ArRobot *robot) {
2
3     ArAction::setRobot(robot);
4     robotSonar = robot->findRangeDevice("sonar");
5 }
6 range = robotSonar->currentReadingPolar(-170, 170)
7         - myRobot->getRobotRadius();
```

Figura 6. Detecção do sonar e uso no método *fire*

ArLaser é a classe responsável por recuperar as leituras referentes aos *lasers* do robô. Diferentemente do sonar, onde a conexão do dispositivo é feita instanciando um objeto **ArSonarDevice** e adicionando-o ao robô, o *laser* utiliza a classe **ArLaserConnector**, e não a **ArLaser**, para criar seus objetos e o método *connectLasers()* para conectar o dispositivo no robô (Figura 7, linha 6). Após conectar o(s) *laser(s)*, ele(s) pode(m) ser buscado(s) através dos métodos da **ArRobot**: *findLaser()*, o qual pegará um *laser* específico que esteja conectado, ou através do *getLaserMap()*, que recupera todos os *lasers* conectados ao robô.

Em uma ação, pode-se tanto usar apenas um *laser*, quanto todos que estiverem conectados. Isso dependerá do comportamento da ação. Na ação, define-se um ponteiro **ArLaser**. Ao definir o robô que realizará a ação, é feita a busca pelo dispositivo através dos métodos citados anteriormente (Figura 7, linha 4). A partir daí, para recuperar as

leituras feitas pelo dispositivo, assim como é feito com o sonar, utiliza-se o método *currentReadingPolar*.

```
1  void ActionMove::setRobot(ArRobot *robot) {
2
3      ArAction::setRobot(robot);
4      robotLaser = robot->findLaser(1);
5  }
6  ArLaserConnector laserConnector(&parser, &robot, &robotConnector);
7  if (!laserConnector.connectLasers()) {
8      ArLog::log(ArLog::Terse,
9              "Could not connect to configured lasers. Exiting.");
10     Aria::exit(3);
11     return 3;
12 }
```

Figura 7. Conexão do *laser* e uso no método *fire*

5. Conexão Socket: Comunicação Entre Aplicações

A aplicação principal da API, além de se comunicar com o robô, pode também se comunicar com outra aplicação que utilize a ARIA. Nessa comunicação, determina-se que uma aplicação se comportará como servidor, enquanto a outra se comportará como o cliente. A classe **ArSocket** se comporta como uma camada que permite a utilização dos soquetes de rede de forma independente ao sistema operacional [MobileRobots 2013a], contendo funções para receber um cliente, conectar-se a um servidor e transmitir dados entre eles.

```
1  ArSocket serverSocket, clientSocket;
2
3  Aria::init();
4
5  if (serverSocket.open(7777, ArSocket::TCP))
6      ArLog::log(ArLog::Normal, "Server: Opened the server port.");
7  else {
8      ArLog::log(ArLog::Normal, "Server: Failed to open the server port: %s.",
9              serverSocket.getErrorStr().c_str());
10     Aria::exit(-1);
11 }
12
13 if (serverSocket.accept(&clientSocket))
14     ArLog::log(ArLog::Normal, "Server: Client has connected.");
15 else
16     ArLog::log(ArLog::Terse, "Server: Error in accepting a connection from the client: %s."
17             serverSocket.getErrorStr().c_str());
```

Figura 8. Conexão do servidor na porta 7777 e no endereço *localhost*

Uma aplicação servidor pode ser vista na Figura 8 e deve possuir dois objetos do tipo **ArSocket**, os quais podem ser vistos na linha 1: um referente ao próprio servidor e outro referente ao cliente. Após estabelecida a conexão do servidor (linha 5), ele fica aguardando até que um cliente seja recebido (linha 13). Na aplicação cliente (figura 9), é necessário apenas o objeto referente a ele mesmo (linha 1), o qual irá se conectar ao servidor a partir da porta e endereço definidos pelo do servidor (linha 5).

```
ArSocket clientSocket;

ArLog::log(ArLog::Normal,
        "Client: Connecting to localhost TCP port 7777...");
if (clientSocket.connect("localhost", 7777, ArSocket::TCP))
    ArLog::log(ArLog::Normal,
            "Client: Connected to server at localhost TCP port 7777.");
else {
    ArLog::log(ArLog::Terse,
            "Client: Error connecting to server at localhost TCP port 7777: %s",
            clientSocket.getErrorStr().c_str());
    Aria::exit(-1);
}
```

Figura 9. Conexão do cliente no servidor

Basicamente, as duas aplicações irão realizar as mesmas funções de ler e enviar dados utilizando os métodos *read()* e *write()*, respectivamente, modificando apenas quais dados ou como isso será feito. Por exemplo, assumindo que o cliente é a aplicação do robô (ver Figura 10(a)), ele poderá colher as leituras do sonar, *laser* (ou outros dispositivos) (linhas 1 e 3) e enviá-los para o servidor (linhas 6 e 7). No servidor (ver Figura 10(b)), esses dados serão recebidos e processados de forma a calcular uma velocidade segura de movimento do robô a partir dos dados fornecidos pelo cliente (linhas 1 até 11). Após os cálculo da velocidade, o servidor envia o resultado de volta para o cliente (Figura 10(b), linha 13), onde o valor da velocidade de movimento será alterado (Figura 10(a), linha 12).

```
range = robotSonar->currentReadingPolar(-170, 170)
        - myRobot->getRobotRadius();
distance = robotLaser->currentReadingPolar(robotLaser->getStartDegrees(),
        robotLaser->getEndDegrees(), 0);

client.write(&range, sizeof(range));
client.write(&distance, sizeof(distance));

float speed;
client.read(&speed, sizeof(speed));

robotDesire.setVel(speed);
```

(a) Aplicação cliente

```
server.read(&range, sizeof(range));
server.read(&distance, sizeof(distance));

float speed;

if(range <= 500 && speed > 100)
    speed *= 0.3;
else if(range <= 100)
    speed = 0;
else
    speed = maxSpeed;

server.write(&speed, sizeof(speed));
```

(b) Aplicação servidor

Figura 10. Comunicação entre aplicações cliente e servidor

6. Conclusão

Este artigo teve por objetivo apresentar os passos básicos para a programação de robôs móveis usando a API ARIA. Programar robôs móveis não é uma tarefa tão simples quando o desenvolvimento deve-se preocupar com detalhes em baixo nível, como recuperação da odometria das rodas do robô, conexão e leitura dos dispositivos instalados, conexão entre o robô e computador, entre outros. Além disso, ainda há a implementação dos comportamentos que o robô deve executar em ambientes com obstáculos (na grande maioria dos casos) por parte do programador.

A ARIA provê muitas funcionalidades em alto nível para o desenvolvimento de aplicações com robôs móveis, proporcionado uma forma rápida e segura de testar os algoritmos que devem ser executados pelos robôs. Consequentemente, os pesquisadores economizam tempo e dinheiro. Portanto, a utilização da API ARIA juntamente com o simulador MobileSim torna-se uma boa alternativa para o desenvolvimento de pesquisas com robôs móveis.

Referencias

ActivMedia (2005). Activmedia robotics interface for application (aria). available with ARIA software download.

BAY, J. S. (1995). Design of the army-ant cooperative lifting robot. *IEEE Robotics and Automation Magazine*, 2(1):36–43.

FRACASSO, P. T. and COSTA, A. H. R. (2005). Navegação reativa de robôs móveis autônomos utilizando lógica nebulosa com regras ponderadas. *VII SBAI / II IEEE LARS*.

MANDOW, A., GOMES-DE-GABRIEL, J. M., MARTINEZ, J. L., MUÑOZ, V., OLLERO, A., and GARCÍA-CEREZO, A. (1996). The autonomous mobile robot aurora for greenhouse operation. *IEEE Robotics and Automation Magazine*, 3(4):18–28.

MobileRobots (2013a). Aria developer‘s reference manual. http://robots.mobilerobots.com/Aria/docs/main.html, Acesso em 05/03/2014.

MobileRobots (2013b). Pioneer robots software development kit. http://www.mobilerobots.com/Software.aspx, Acesso em 27/02/2014.

MURPHY, R. R. (2000). Marsupial and shape–shifting robots for urban search and rescue. *IEEE Intelligent Systems*, 15(2):14–19.

SCATENA, J. M. (2008). Ambiente de desenvolvimento de aplicações para robôs móveis. Tese (Doutorado), Escola de Engenharia de São Carlos, Universidade de São Paulo.

TRONCO, M. L. and PORTO, A. J. V. (2005). Módulo de visão omnidirecional com controlador fuzzy para navegação de rôbo móvel autônomo. *VII SBAI / II IEEE LARS*.

WHITBROOK, A. (2010). *Programming Mobile Robots with Aria and Player*. Springer, Nottingham – UK.