PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Programação de robôs distribuídos em Lua

Igor Babatunde Pinheiro Abiola

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Engenharia da Computação

Rio de Janeiro, novembro de 2010

Sumário

1 – Introdução

Nos últimos anos vem se difundindo o conceito de robôs distribuídos, enxames de robôs e WSN, com o propósito de diminuir o custo por robô, aumentar a redundância, robustez e capacidade de processamento dos sistemas. Em paralelo houve um aumento na disponibilidade de kits de robótica programáveis “off-the-shelf” de uso comercial e doméstico. Este fato possibilita que sistemas de robôs distribuídos possam ser estudados e construídos com esses kits diminuindo o tempo e custo de desenvolvimento.

Um grande problema na utilização de kits de robótica é o ambiente de desenvolvimento, que em sua maioria são proprietários e voltados para o publico iniciante, impossibilitando o seu uso para aplicações mais complexas. Linguagens como Lua, Perl e Python podem servir como ponte entre a facilidade de uso e aprendizado e a implementação de aplicações de alta complexidade, dada a sua facilidade de integração a programas e bibliotecas escritas em outras linguagens. Lua com sua fácil comunicação com C é ideal para este trabalho, pois a integração com dispositivos, como GPS, é geralmente feita em C. Alem disso a linguagem tem uma sintaxe simples e uma baixa curva de aprendizado.

Para um desses kits - o Nxt Mindstorms[LEGO00] - existem dois ambientes de desenvolvimento usando a linguagem Lua: o pbLua[HEM00] que é um firmware para o Nxt e o LuaNxt[FUSCO00] que possibilita o controle remoto do Nxt através da interface USB ou Bluetooth.

Foi escolhido como base para este trabalho o LuaNxt, pois além de ter sido produzido na própria Puc-Rio não requer nenhuma modificação no kit, podendo ser utilizado em qualquer Nxt Mindstorms com o firmware padrão instalado. Para a parte distribuída estou utilizando a biblioteca ALua[LUA01] em conjunto com a biblioteca DAlua[LUA02].

O sistema proposto aqui visa facilitar o estudo de aplicações de robótica distribuída fornecendo uma forma rápida de construir um sistema distribuído integrado ao kit Nxt, para que o maior tempo de desenvolvimento seja gasto no estudo dos algoritmos estudados e não na implementação do sistema.

Figura 1: Peças do Kit Nxt Mindstorns

Figura 2: Kit Montado.

2 – Estado da arte

Apesar do kit Nxt poder ser programado em uma variedade de linguagens dentre elas, Lua, Ruby, Java e outras, não existe até o momento uma biblioteca que facilite o desenvolvimento de sistemas com múltiplos robôs e sensores distribuídos. A seguir descreverei as tecnologias estudadas para o desenvolvimento desta biblioteca.

Nxt Mindstorms - O kit Nxt Mindstorms da Lego é composto por um micro controlador ARM7 de 32-bits com 256 Kb de Flash e 64Kb de Ram, possui quatro portas de entrada de sensores e três portas de saída, e interface USB e Bluetooth[LEGO00]. Pode ser facilmente montado em diferentes configurações, e programado usando várias linguagens e controlado remotamente, tornando-o ideal para construção de protótipos.

PbLua - Desenvolvido por Ralph Hempel[HEM00], o PbLua é um firmware que porta o interpretador de Lua para o microprocessador ARM7 do Nxt, possibilitando rodar programas escritos em Lua diretamente do robô. A principal vantagem dessa abordagem é a velocidade, pois não há latência de comunicação entre o robô e o software de controle, o que acontece quando o software é executado fora do robô.

LuaNxt - Desenvolvido por Victor S. F. Fusco como projeto final de graduação em 2007 na PUC – Rio[FUSCO00] o LuaNxt permite o controle do Nxt remotamente através de uma api em Lua, utilizando as interfaces USB ou Bluetooth do robô. O uso do software de controle fora do robô permite o desenvolvimento de programas mais complexo, pois não possui as limitações de armazenamento do Nxt.

Not eXactly C – ou nxc[NXC00] é uma linguagem de alto nível similar a C e é compilada para código de maquina para ser executado no kit Nxt. Ela permite o desenvolvimento de aplicações mais poderosas que o ambiente de desenvolvimento padrão, porem para alguém com pouca experiência em programação a linguagem apresenta uma curva de aprendizado acentuada.

3 – Objetivos do trabalho

A proposta do presente trabalho é a implementação de uma biblioteca na linguagem Lua pra facilitar o desenvolvimento de sistemas robóticos distribuídos baseados no kit Nxt Mindstorms, na biblioteca alua e na biblioteca LuaNxt. Esta biblioteca deveria ser simples e extensível de forma a no futuro poder ser utilizada com outros kits robóticos.

A biblioteca permite o processamento distribuído de informações vindas de vários robôs Nxt e de agentes externos, e a execução de comandos cadastrados nos servidores nxt , originados das unidades de processamento locais ou remotas, tais comandos poderão ser executados de forma síncrona ou assíncrona e podendo usar áreas criticas de exclusão mutua.

Cada servidor nxt modela um robô especifico e tem seu próprio conjunto de comandos, assim é possível ter um conjunto heterogêneo de robôs. Esses comandos devem ser cadastrados no inicio da execução do servidor, porem poderão também ser cadastrados e redefinidos durante a execução.

O publico alvo desta biblioteca são os estudantes de engenharia e programação e entusiastas de robótica, com alguma experiência em programação e que procuram uma forma simples de testar e implementar aplicações e algoritmos de complexidade média e de forma distribuída.

4 – Atividades realizadas

Estudo Conceituais e de Tecnologia:

As principais tecnologias necessárias para o desenvolvimento deste projeto são: a linguagem Lua, e as bibliotecas LuaNxt, Alua e DAlua.

A biblioteca LuaNxt pode ser facilmente integrada a outras bibliotecas e tem uma curva de aprendizado média, mas necessita um conhecimento prévio de programação com o o kit Nxt, as configurações de motores e sensores, assim como a conexão com o robô se mostraram como a maior dificuldade encontrada. Ao longo do desenvolvimento do projeto foram encontrados alguns erros que dificultaram o progresso do projeto. Estas correções estão descritas em anexo.

Processo de projeto e desenvolvimento:

O processo de desenvolvimento foi orientado a duas aplicações de exemplo, de forma que o projeto foi sendo implementado e modificado até que a aplicação estivesse totalmente funciona. As aplicações foram concebidas visando utilizar todos os recursos desejáveis na biblioteca.

A primeira aplicação tinha como objetivo a comunicação dos nós da aplicação e a execução de comandos originados em nós remotos, no robô.

A segunda aplicação visava a leitura e processamento de dados dos sensores do robô pelos nós da aplicação e a geração de novos comandos a partir deste processamento.

Cronograma planejado para Projeto I (incluído na proposta) :

Atividades / Mês	Set/09	Out/09	Nov/09
Atualização na linguagem Lua	X		
Estudo do pbLua/LuaNxt	X	X	
Estudo de bibliotecas de Sistemas Distribuídos	X	X	
Especificação do Sistema			X

Cronograma planejado para Projeto I (o que foi realizado de fato) :

Atividades / Mês	Set/09	Out/09	Nov/09
Atualização na linguagem Lua	X		
Estudo do pbLua/LuaNxt		X	X
Estudo de bibliotecas de Sistemas Distribuídos		X	X
Especificação do Sistema			X

Cronograma planejado para Projeto II :

Atividades	Fevereiro	Março	Abril	Maio	Junho
Implementação parcial	X	X			
1º aplicação de teste		X			
Implementação total e refatoração			X		
2º aplicação de teste				X	
Ajustes na documentação					X
Relatório Final					X

Cronograma real para Projeto II :

Devido a problemas pessoais, o projeto foi abandonado em abril e somente retomado no mês de agosto.

Atividades	Fevereiro	Março	Agosto	Setembro	Outubro	Novembro
Implementação parcial	X	X				
1º aplicação de teste		X	X			
Implementação total e refatoração				X	X	
2º aplicação de teste				X	X	
Ajustes na documentação						X
Relatório Final						X

5 – Projeto e especificação do sistema

O Sistema desenvolvido apresenta uma arquitetura cliente – servidor, podendo haver múltiplos clientes e múltiplos servidores. Os servidores são nós com conexão direta com o kit Nxt e são responsáveis por repassar os comandos emitidos por outros nós. Os clientes processam dados vindos dos servidores e outros clientes e enviam comandos aos servidores. A principal vantagem desta arquitetura é a simplificação do cliente, enquanto o servidor precisa de uma interface bluetooth ou usb alem de um interface de rede, o cliente só necessita de acesso a rede pois depende apenas de código Alua. Isso facilita a implementação de clientes em plataformas de baixo poder de processamento como dispositivos móveis.

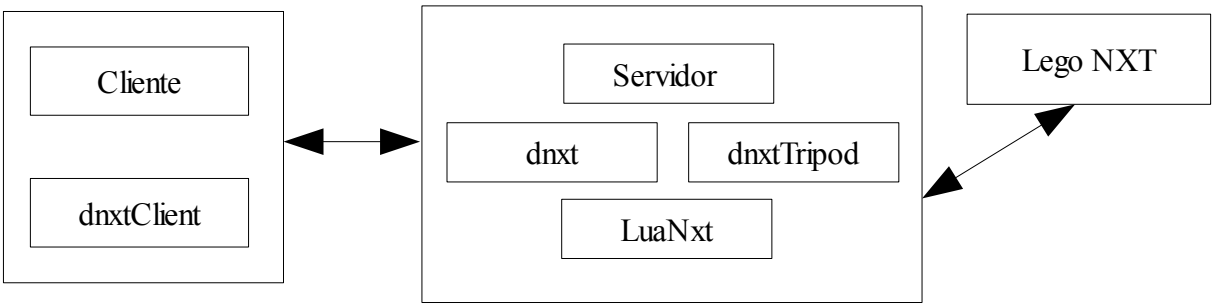


Figura 3: Diagrama dos módulos.

Além dos módulos cliente e servidor foi implementado um módulo auxiliar com funções de controle do kit, tomando como base a construção tribot[LEGO01] do kit.

Figura 4: Kit Nxt Mindstorms montado como tribot.

A seguir cada módulo é descrito de forma detalhada.

5.1 – Dnxt

Módulo servidor responsável pela conexão com o kit Nxt e pela criação da aplicação distribuída. Em sua função de inicialização recebe o endereço ip e porta na qual o servidor deve ser estabelecido e opcionalmente um endereço e porta de uma aplicação dnxt já existente, procura pela aplicação DAlua de nome “nxt” criando-a caso não exista.

A comunicação com o kit é feita pela biblioteca LuaNxt. O usuário escolhe qual a interface de conexão deve ser usada, usb ou bluetooth. Também deverá ser escolhido um identificador para o robô, que será usado para referenciar o robô em toda a aplicação.

O usuário poderá cadastrar funções para serem executados sobre um determinado robô. Essas funções são chamadas comandos, e para cada comando é associado um apelido. Comandos são funções Lua e recebem como parâmetro o objeto LuaNxt que representa o robô. O servidor ao receber um comando resolve com a ajuda do identificador em qual dos robôs o comando deve ser executado.

A interface do módulo e suas funções estão descritas no apêndice 9.2. Segue abaixo um exemplo simples da utilização do Dnxt.

```
require "dnxt"

mode = arg[1] or  "usb"

function printHello( robotID, robot, mutexList )
    print( “hello robot:”, robotID )
end

dnxt.nxtConnect(mode, "robot1", "00:16:53:07:58:F1")
dnxt.nxtSerCommand(“hello”, printHello)
dnxt.initServer(“127.0.0.1”, 4321)
```

5.2 – DnxtTripod

Com a possibilidade de construir o kit Nxt com formas diversas fica impossibilitada a criação de um módulo com comandos genéricos aplicáveis a qualquer construção. Optamos por implementar o modulo para uma forma padrão, desta forma facilitando o uso por um usuário que deseja apenas testar a biblioteca e também servir de exemplo para implementação de comandos por usuários com uma forma customizada do robô. Abaixo segue um exemplo de comando implementado por este módulo.

```
function forward(robotID, robot )
        robot:SetOutputState( "PORT_B",  80 , "MOTORON_REGULATED",
"MOTOR_SYNC" , 0 , "RUNNING", 50 )
        robot:SetOutputState( "PORT_C",  80 , "MOTORON_REGULATED",
"MOTOR_SYNC" , 0 , "RUNNING", 50 )
end
```

Este comando liga os motores conectados nas portas B e C, com oitenta porcento de potencia por cinquenta rotações, isso faz com que o robô ande em uma linha reta.

5.3 – DnxtClient

Este módulo tem a função de administrar a comunicação do código cliente com os nós servidores. A principal forma de comunicação dos nós clientes com os servidores é através do pedido de execução de comandos a serem executados em um robô. Estes pedidos são transmitidos a todos os servidores conectados a aplicação, mas somente o servidor que estiver fisicamente conectado ao robô destinatário do pedido executará o comando, desta forma não é necessário saber a qual servidor um determinado kit está conectado. Comandos podem ser executados em uma ou mais regiões de exclusão mutua, assim garantindo que nenhum outro comando execute concorrentemente com um comando dentro da região. No ato de conexão com o kit, é criada uma região critica com o mesmo identificados do kit.

É de responsabilidade do usuário chamar a função de saída da região critica como mostra o exemplo abaixo.

```
dnxtClient.createMutex(“mutex”)
function mutexRead()
    --pede a execução do comando read no robô “robot1” na região critica “mutex”
    --o comando read recebe como parâmetro o nome de uma callback
        dnxtClient.executeCommand(“robot1”,"read",{"mutex"}, "process" );
end

function process( msg, mutexList )
       --processa a o dado retornado
       print(msg)
       --Libera a região critica
       dnxtClient.exitCR(mutexList)
end
```

A função mutexRead executa o comando read sobre o robô robot1 dentro a região de exclusão mutuá mutex. O comando read read retorna o valor lido com uma chamada a função process. Não importa quantas chamadas simultâneas sejam feitas a mutexRead, elas serão executadas uma por vez. Ao final do processamento a função process chama dnxtClient.extCR

para liberar as regiões criticas.

Alem de executar comandos é possível também ler dados dos sensores do robô, de forma pontual ou agendando a leitura automática após um intervalo de tempo. O pedido de leitura é passado ao servidor que retorna o valor lida na callback especificada.

Antes de iniciar o cliente o usuário deve especificar uma função de loop, esta função será chamada após a inicialização do cliente e subsequentemente após o intervalo especificado. Abaixo está um exemplo de inicialização simples.

```
require "dnxtClient"
require "dalua"

--função de loop
function mainLoop(...)
        --move o robô para frente
        dnxtClient.executeCommand(“robot1”,"forward",”nil”);
end
--registra a função de loop para ser chamada 10 vezes  a cada ½ segundo
dnxtClient.registerCallback("mainLoop", 1/2, 10)
--inicializa o cliente na porta 4322 com o servidor na porta 4321
dnxtClient.initClient("127.0.0.1", 4322, "127.0.0.1", 4321)
```

O exemplo acima inicia um cliente no endereço 127.0.0.1:4322 conectado no servidor 127.0.0.1:4321, e a cada 0,5 segundos executa o comando forward no robô robot1.

5.4 – Comunicação com programas nativos nxt

A comunicação com programas rodando no nxt é feita por um sistema de mailbox. O nxt possui 20 mailbox cada uma com 5 posições em forma de fila. As primeiras 10 mailbox devem ser usadas para enviar mensagens ao nxt e as demais para o robô enviar mensagens. Caso uma nova mensagem chegue e a fila esteja cheia a mensagem mais velha é descartada.

A principal vantagem em usar programas nativos é o ganho de performance, que é resultante da diminuição da comunicação entre o servidor e o kit. Este ganho fica claro se considerarmos operações complexas como o movimento sincronizado de vários motores ou a leitura do sensor de ultrassom que requer vários comandos de inicialização a cada leitura. A interface bluetooth do nxt demora 50ms para trocar a direção da comunicação, assim ler uma dado no nxt demora 50ms alem do tempo de processamento e transmissão.

5.5 – Exemplos Desenvolvidos

Ao longo do desenvolvimento da biblioteca foram desenvolvidos dois exemplos básicos que serviram como parâmetro guia do processo e demonstração do projeto.

O primeiro exemplo demonstra o controle do kit por um nó remoto, que coletava dados de um joystick usb, processava estes dados e os transformava em comandos para o servidor. O objetivo deste exemplo era testar a comunicação entre os nós da aplicação pelo sistema de comandos. O trecho de código a seguir ilustra o processamento dos dados do joystick.

```
--joystick_control.lua
require "dnxtClient"
--inicialização do joystick
function initJoystick()
  ...
end

function  procJoystick()
        s, err = dhandle:interrupt_read(0x81, 21,0)
        if err == nil then
                a, b, c, xi, yi, zi = string.byte(s , 1 , #s)
                dnxtClient.executeCommand(“robot1”,"move",”nil”, xi, yi);
        else
                print("err =",err)
        end
end
--função de loop
function mainLoop(...)
        procJoystick()
end
--registra a função de loop para ser chamada a cada 1/5 segundo  indefinidamente
dnxtClient.registerCallback("mainLoop", 1/5, 0)
--inicializa o cliente na porta 4322 com o servidor na porta 4321
dnxtClient.initClient("127.0.0.1", 4322, "127.0.0.1", 4321)
```

```
--server.lua
require "dnxt"

mode = arg[1] or  "usb"

function move( robotID, robot, mutexList,  x, y )
        if( x == 128 ) then
                if( y == 128 ) then
```

```
                        robot:SetOutputState( "PORT_B", 0 , "BRAKE", "IDLE" , 100 , "IDLE",
0 )
                robot:SetOutputState( "PORT_C", 0 , "BRAKE", "IDLE" , 100 , "IDLE", 0 )
                end
        else
                robot:SetOutputState(  "PORT_B",  -((  (  y-128  )/128  )*100  )  -  (    ((x-
128)/128)*100 )  , "MOTORON", "MOTOR_SPEED" , 100 , "RUNNING", 0 )
                robot:SetOutputState(   "PORT_C",    -(((y-128)/128)*100)   +   (       ((x-
128)/128)*100 ) , "MOTORON", "MOTOR_SPEED" , 100 , "RUNNING", 0 )
        end
end

dnxt.nxtSerCommand(“move”, move)
dnxt.nxtConnect(mode, "robot1", "00:16:53:07:58:F1")
dnxt.initServer()
```

O segundo exemplo demonstra a interação com sensores e regiões de exclusão mutua. Nele o nó remoto com os dados obtidos dos sensores faz com que o kit siga um circuito delimitado por uma linha preta. A região critica foi usada para garantir que as leituras do sensor focem feitas com o robô sempre parado.

Neste exemplo foi usado um programa nativo nxt rodando no robô. Este programa lê os dados do sensor e o envia para a maquina a qual esta conectado, como descrito anteriormente, e espera mensagens do servidor para saber para qual direção andar. O uso desta forma de comunicação com o robô neste exemplo foi escolhido para demonstrar o conceito e melhorar o desempenho do exemplo. No desenvolvimento deste exemplo foi usada a linguagem Not eXactly C para escrever o programa main.rxe, este se encontra em anexo. Abaixo um trecho do exemplo.

```
require "dnxtClient"
require "dalua"

local on = nil
function init(...)
        if not on then
                --inicia a aplicação nativa main.rxe
                dnxtClient.executeCommand(dnxtClient.allRobots(),"startProgram","nil",
"main.rxe", true )
                on = true
        end
        mutexRead()
end
```

```
function mutexRead()
	--lê dados da mailbox 12, onde o main.rxe coloca os dados do sensor, os dados são processados pela função  processLight
	dnxtClient.executeCommand(dnxtClient.allRobots(),"readMsg",{"robot1"},	12	, "processLight" );
end

local degree_cunt = 0
function processLight( inbox, msg, mutexList )
	...
	if  light < 40 then
		--manda uma mensagem para o main.rxe, informando para andar para frente
		dnxtClient.executeCommand(dnxtClient.allRobots(),"sendMsg","nil", 0, 0 )
		degree_cunt = 0
	else
		local inc = 15;
		if degree_cunt > 180 then
			cw = not cw
			degree_cunt = -degree_cunt
		end
		if cw then
		--manda uma mensagem para o main.rxe, informando para rotacionar para a esquerda
			dnxtClient.executeCommand(dnxtClient.allRobots(),"sendMsg","nil", 0, -90 );
		else
		--manda uma mensagem para o main.rxe, informando para rotacionar para a direita
			dnxtClient.executeCommand(dnxtClient.allRobots(),"sendMsg","nil", 0, 90 );
		end
		degree_cunt = degree_cunt + inc
	end
	-- verifica se o movimento acabou e libera a região critica
	checkMovimentstatus(mutexList)
end
```

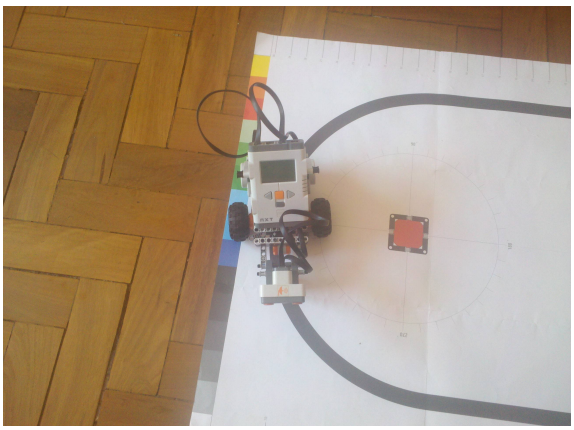


Figura 5: Teste do 2º exemplo.

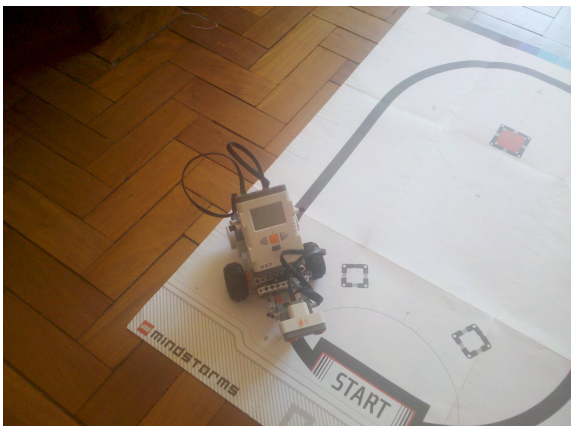


Figura 6: Teste do 2º exemplo

6 – Implementação e Avaliação

6.1 – Planejamento e execução de testes funcionais

Foram planejadas duas etapas de desenvolvimento e em cada uma delas foi produzido um exemplo funcional que testa e demostra as funcionalidades da biblioteca. Apesar dos exemplos terem sido planejados logo no inicio do projeto, suas forma e funcionalidades finais foram implementadas e modificadas durante o processo de desenvolvimento, mas sempre mantendo os objetivos traçados no planejamento.

6.2 - Comentários sobre a implementação

Ao longo do desenvolvimento foram encontradas falhas na biblioteca LuaNXT que dificultaram o andamento do projeto. Destas falhas duas tiveram um maior impacto. A primeira um erro de implementação na função de escrita na lowspeed. Este erro tornava inviável o uso do sensor de ultrassom, algumas semanas foram perdidas tentando determinar se o sensor não funcionava por mau uso da api ou por falha da biblioteca até que o erro foi encontrado. A segunda falha era na comunicação bluetooth com o nxt,a LuaNXT sempre levava dois segundos para efetuar qualquer leitura via bluetooth, devido a um erro no usa da biblioteca de sockets. Esta falar provocou a a sensação de que a comunicação bluetooth era quase inviável e a procura de formas de diminuir a troca de mensagens entre o servidor e o kit, levando ao desenvolvimento de um exemplo usando um programa nativo nxt.

Outro problema encontrado foi falta de experiência com o kit nxt. Apesar do estudo

realizado a falta de experiência com o kit levou a um rendimento menor que o esperado e a simplicidade dos exemplos apresentados.

7 – Considerações finais

O presente projeto gerou a biblioteca Dnxt, que pode servir de plataforma para estudos sistemas distribuídos aplicados a robótica, sem que o estudante perca tempo construindo o ambiente de desenvolvimento podendo focar nos algoritmos e tecnologias especificas do seu estudo, mas sem perda de funcionalidade.

Durante o processo de desenvolvimento foi adquirido conhecimento e experiência em áreas como robótica, comunicação bluetooth, sistemas distribuídos, sockets, alem da experiência de desenvolver um projeto do inicio ao fim.

Ao longo do projeto foram identificados alguns temas para trabalhos futuros, como o adaptação do Dnxt para plataformas moveis que já tem suporte a Lua, como android , iphone e outros. A integração com o fimware pblua, esta integração possibilitaria o desenvolvimento de todas as partes do sistema em Lua, assim tornando mais simples o uso da biblioteca.

Bibliografia

LEGO00: Lego, Nxt Mindstorms, http://mindstorms.lego.com/
HEM00: Ralph Hempel, pbLua, http://www.hempeldesigngroup.com/lego/pbLua/
FUSCO00: Victor S. F. Fusco, Ambiente de desemvolvimento para o kit de robótica Nxt Mindstorms, 2007
LUA01: Various, ALua: Programação Paralela e Distribuída em Lua, , http://alua.inf.puc-rio.br/
LUA02: Various, DALua, http://alua.inf.puc-rio.br/dalua/
NXC00: Various, Next Byte Codes & Not eXactly C, http://bricxcc.sourceforge.net/nbc/
LEGO01: Lego, http://mindstorms.lego.com/en-us/support/buildinginstructions/8527-/Tribot.aspx
LUA00: Various, Lua , http://www.lua.org/
MAKE00: Various, http://en.wikipedia.org/wiki/Make_(software)
GCC00: Various, http://en.wikipedia.org/wiki/GNU_Compiler_Collection

9 – Apêndices

9.1 – Manual do Usuário

Requisitos:

1. Lua-5.1[LUA00] ;
2. Uma ferramenta GNU make[MAKE00] ;
3. Um copilador C compatível com o GCC[GCC00];

Instalação(no ambiente linux):

O primeiro passo é editar o arquivo config na raiz da pasta da biblioteca, devemos editar as seguintes entradas:

1. A variável LUABASE deve conter o caminho para o diretório da instalação da linguagem Lua. EX:

   LUABASE=/home/igor/lua

2. A variável CC deve conter o caminho para o compilador C. EX:

   CC=gcc

   Agora basta executar o makefile passando a opção de sistema operacional usado, linux ou windows. EX:

   make linux

9.2 – Manual de referência – dnxt

Abaixo segue uma breve descrição das funções de cada modulo. A documentação completa encontra-se em formato digital e acompanha o código fonte do projeto.

initServer (addr, port)	Inicia o loop do servidor.
nxtConnect (mode, name, mac)	Conecta ao kit nxt.
nxtSetCommand (name, command)	Cadastra função para ser chamada remotamente.

9.2.1 – Comandos básicos

sendMsg(robotID, robot, mutexList, mailbox_index ,msg)	Envia uma mensagem para o programa em execução no nxt.
readMsg(robotID, robot, mutexList, inbox , callback, ...)	Lê a próxima mensagem da fila.
startProgram(robotID, robot, program)	Inicia a execução de um programa nativo.
stopProgram(robotID, robot)	Para a execução de um programa nativo.

9.3 - Manual de referência – dnxtClient

executeCommand (robotID, cmd, mutexList ,...)	Executa um comando em um robô.
getSensorData (robotID, port, callback)	pede o valor de um sensor.
initClient(addr, port, serverAddrParam, serverPortParam)	inicia loop do cliente.
registerCallback (callback, frequency, times , ...)	Registra função para ser chamada no cliente em um tempo determinado.
registerSensorCallback(robotID, port, frequency, callback, lowspeed)	registra nos servidores o cliente para receber o valor de um sensor.
removeSensorCallback (robotID, port)	Remove o registro do cliente de receber o valor de um sensor.
mutexfunction (mutex, func, ...)	Chama função dentro de uma Região critica.

9.4 - Manual de referência(comandos) – dnxtTripod

forward(robotID, robot)	Move o robô para frente.
back(robotID,robot)	Move o robô para trás.
stop(robotID,robot)	Para o robô.
rotateRight(robotID, robot , degrees)	Rotaciona o robô para a direita.
rotateLeft(robotID, robot, degrees)	Rotaciona o robô para a esquerda.

9.5 - Tutorial de uso da biblioteca Dnxt

Tutorial de uso da biblioteca Dnxt.

Igor Babatunde Pinheiro Abiola

Sumário

Introdução:

Este tutorial tem como objetivo esclarecer o uso da biblioteca Dnxt, através do estudo de uma aplicação simples. A aplicação deverá, usando o sensor de luminosidade e os motores, conduzir o robô por um circuito delimitado por uma tarja preta no chão.

Começaremos pela instalação da biblioteca, depois uma breve descrição da arquitetura e então passaremos para a análise dos arquivos que compõem o exemplo.

Instalação:

Requisitos:

1. Lua-5.1 [LUA00] ;
2. Uma ferramenta GNU make [MAKE00] ;
3. Um copilador C compatível com o GCC [GCC00];

Instalação(no ambiente linux):

O primeiro passo é editar o arquivo config na raiz da pasta da biblioteca, devemos editar as seguintes entradas:

1. A variável LUABASE deve conter o caminho para o diretório da instalação da linguagem Lua. EX:

   LUABASE=/home/igor/lua

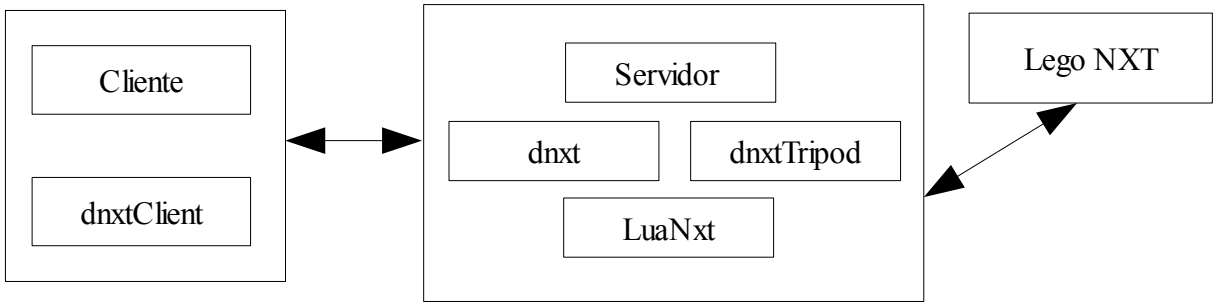
2. A variável CC deve conter o caminho para o compilador C. EX:

   CC=gcc

   Agora basta executar o makefile passando a opção de sistema operacional usado, linux ou windows. EX:

   make linux

Arquitetura:



A biblioteca Dnxt é formada por três módulos: módulo dnxt, dnxtClient e dnxtTripod.

O módulo dnxt pode ser considerado um servidor, pois ele esta encarregado de repassar os pedidos dos outros módulos ao robô, e para isto deve ser executado em uma máquina com acesso físico ao nxt. Pode se ter várias instâncias do módulo dnxt e cada instância pode se conectar a vários robôs, mas para cada robô deve ser atribuído um identificador único.

O módulo dnxtTripod é auxiliar ao dnxt e contém funções que representam comandos ao robô em uma construção tripod [NXT00].

Figura 1: Kit Nxt Mindstorms montado como tribot.

O módulo dnxtClient tem a lógica da aplicação e se comunica com o dnxt para comandar e ler dados do robô.

Aplicação de Exemplo:

O exemplo é dividido em três arquivos, server.lua, folowLine.lua, main.nxc.

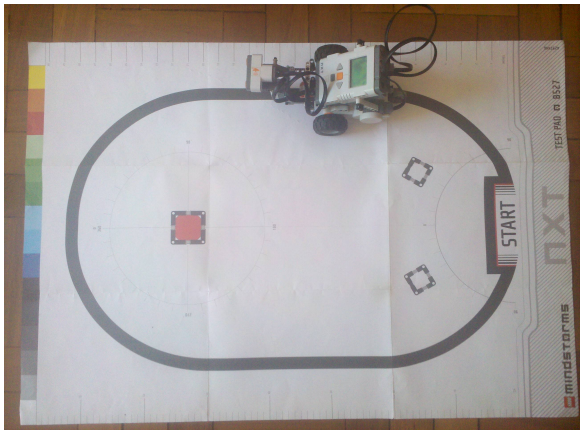


Figura 2: Teste da aplicação de exemplo.

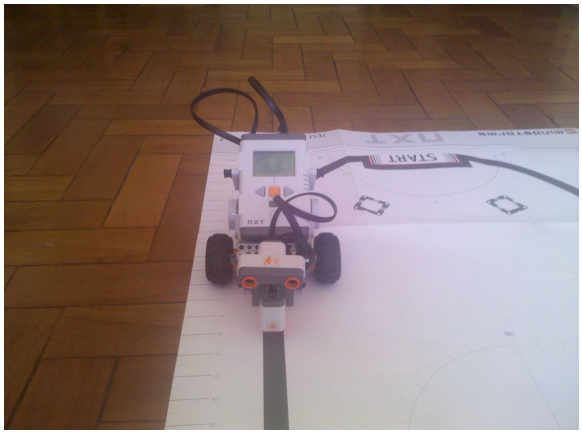


Figura 3: Teste da aplicação de exemplo.

server.lua:

O arquivo server.lua implementa a camada servidor da aplicação e deve ser executado em uma maquina com acesso físico ao robô lego. A camada servidor deve conectar com o lego nxt via usb ou bluetooth e cadastrar os comandos que podem ser executados no servidor.

O arquivo deve fica assim:

Inclui a biblioteca nxt:

```
require “dnxt”
```

Pega o modo de conexão com o nxt dos parâmetros de execução:

```
local mode = arg[1] or “usb”
```

Conecta com o nxt, e atribui um nome a ele:

```
dnxt.nxtConnect(mode, “robot1”, “00:16:53:08:58:F1”)
```

Inicializa o servidor no endereço e porta especificados:

```
dnxt.initServer(“127.0.0.1”, “4321”)
```

folowLine.lua:

Este arquivo contem a logica da aplicação e pode ser executado de qualquer máquina com acesso via rede ao servidor.

Inclui as bibliotecas necessárias:

```
require “dnxt“
require “dalua”
```

Função de inicialização e loop:

```
loca on = nil
function init()
    if not on then
        dnxrClient.executeCommand(“robot1”,”startProgram”, nil, “main.rxe”, true)
        on = true
    end
    mutexRead()
end
```

Em sua primeira execução a função inicia o programa main.rxe no robô, nas demais execuções faz uma leitura do sensor de luminosidade.

A função mutexRead solicita o valor do sensor ao programa em execução no robô. O resultado desta solicitação sera passado para a função processLight.

```
function mutexRead()
    dnxtClient.executeCommand(“robot1”,"readMsg",{"robot1"}, 12 , "processLight" );
end
```

O comando readMsg lê uma mensagem de uma das filas de mensagens, no caso da fila 12 onde o programa main.rxe coloca o valor do sensor.

A função processLight recebe do valor do sensor e decide se o robô deve andar para frente fazer uma rotação:

```
local degree_count = 0
function processLight(inbox, msg, mutexList)
```

```
        local light = 0
        if msg then
            light = string.byte(msg, 1, #msg)
        eles return end
        if light < 40 then
        -- anda pra frente
            dnxrClient.executeCommand(“robot1”, “sendMsg”, “nil”, 0,”frente”)
            degree_count = 0
        else
            local inc = 15;
            if degree_count > 180 then
                cw = not cw
                degree_count = - degree_count
            end
            if cw then
                -- rotação para esquerda
                dnxrClient.executeCommand(“robot1”, “sendMsg”, “nil”, 0,”esquerda”)
            else
                -- rotação para direita
                dnxrClient.executeCommand(“robot1”, “sendMsg”, “nil”, 0,”direita”)
            end
            degree_count = degree_count + inc
        end
        checkMovimentStatus(motexList)
end
```

A função checkMovimentStatus solicita o status do movimento dos motores ao robô:

```
            function  checkMovimentStatus(mutexList)
                dnxtClient.executeCommand(“robot1”, “readMsg”, “nil”, 10, “endMoviment”,
                mutexList)
            end
```

A função endMoviment recebe o status dos motores. Se o ultimo movimento acabou ela libera o acesso para o próximo movimento, caso contrario efetua uma nova verificação em 20 ms.

```
function endMoviment(inbox, msg, mutexList, mutexList2)
        print("endMovment", inbox, msg, mutexList, mutexList2)
        if msg then
                msg = msg:sub(1, #msg-1)
                if msg == "ok" then
                        print ("endMovment ok!", mutexList, mutexList2)
                        dnxtClient.exitCR(mutexList2)
                        return
                end
        end
        dalua.timer.add(dalua.self(), 1/500 , 1, "checkMovimentStatus", mutexList2)
end
```

Configuração da função de inicialização como a função init e que ela sera chamada a cada 200 ms:

```
        dnxtClient.registerCallback(“init”, 1/5, 0)
```

Por ultimo inicia o cliente especificando o seu endereço e o do servidor:

```
        dnxtClient.initClient(“127.0.0.1”, 4322, “127.0.0.1”, 4321)
```

main.nxc:

Para obter uma melhora de desempenho e diminuir a comunicação entre o servidor e o robô é recomentado o uso de programas em código nativo nxt. Estes programas são copiados para o robô e executados conforme necessário. Qualquer linguagem que compile programas para o firmware padrão do nxt pode ser usada com o dnxt. O arquivo main.nxc especifica um programa na linguagem Not eXactly C[NXC00].

A comunicação entre o servidor dnxt e o programa nativo é feita pelo sistema de mailbox do nxt. Existem 10 filas de mensagens e 10 filas de resposta cada uma com 5 posições. Uma vez colocada na fila, a mensagem será retirada no ato da leitura. Se a fila estiver cheia, a nova mensagem substitui a mensagem mais antiga na fila. As filas são numeradas de 0 a 10 para mensagem e 11 a 20 para resposta .

A task move verifica a fila de mensagens de numero 0 (o nxc oferece as constante MAILBOX1 a MAILBOX10 para especificar o numero da fila), a mensagem indica se o robô deve andar para frente ou efetuar uma rotação:

```
task move(){
	string message;
	while( true ) {
		if( ReceiveMessage(MAILBOX1, true, message)== NO_ERR ){
			if ( message == “frente” ){
				OnFwdSync(OUT_BC, 60, 0);
				Wait(300);
			}else{
				if ( message ==”esquerda” ){
					OnFwdSync(OUT_BC, 60, -90);
				}else{ OnFwdSync(OUT_BC, 60, 90); }
			}
		}
		Off(OUT_BC);
		SendResponseString(MAILBOX1, “ok”);
	}
	Wait(50);
}
```

A task lightSensor fica enviando os dados do sensor de luminosidade para o servidor dnxt:

```
task lightSensor(){
	SetSensorLight(S3);
	while( true ){
		byte data = SensorScaled(S3);
		SendResponseNumber(MAILBOX3, data);
		Wait(30);
	}
}
```

A task main inicializa as outras duas tasks:

```
task main(){
	Proccedes(lightSensor, move);
}
```

Bibliografia

LUA00: Various, Lua , http://www.lua.org/
LEGO01: Lego, http://mindstorms.lego.com/en-us/support/buildinginstructions/8527-/Tribot.aspx
NXC00: Various, Next Byte Codes & Not eXactly C, http://bricxcc.sourceforge.net/nbc/
MAKE00: Various, http://en.wikipedia.org/wiki/Make_(software)
GCC00: Various, http://en.wikipedia.org/wiki/GNU_Compiler_Collection