Tutorial de Uso para o Simulador Kilombo

Fundamentos da Estrutura da Informação – DAS 5102

8 de novembro de 2018

1 Introdução ao Kilobot

Os Kilobots[1] são robôs de baixo custo projetados na Universidade de Harvard para estudo de algoritmos e estratégias de coordenação a serem aplicados em enxames de robôs, isto é, em centenas de robôs que trabalham em conjunto para realizar tarefas que não poderiam realizar individualmente. Cada Kilobot tem habilidades básicas típicas de robôs que operam coletivamente, tais como: controlador programável, capacidade de locomoção e capacidade de comunicação com outros Kilobots próximos.

A Figura 1 apresenta um Kilobot e seus principais componentes. Basicamente os robôs são dotados de dois tipos de atuadores: motores de vibração e leds RGB. Para comunicação entre eles utiliza-se emissores e sensores de infravermelho que permitem aos Kilobots trocarem informação com vizinhos próximos e determinarem a distância entre eles. Há também um sensor de iluminação ambiente, que pode ser usado nos algoritmos para guiar os Kilobots em função da luz.

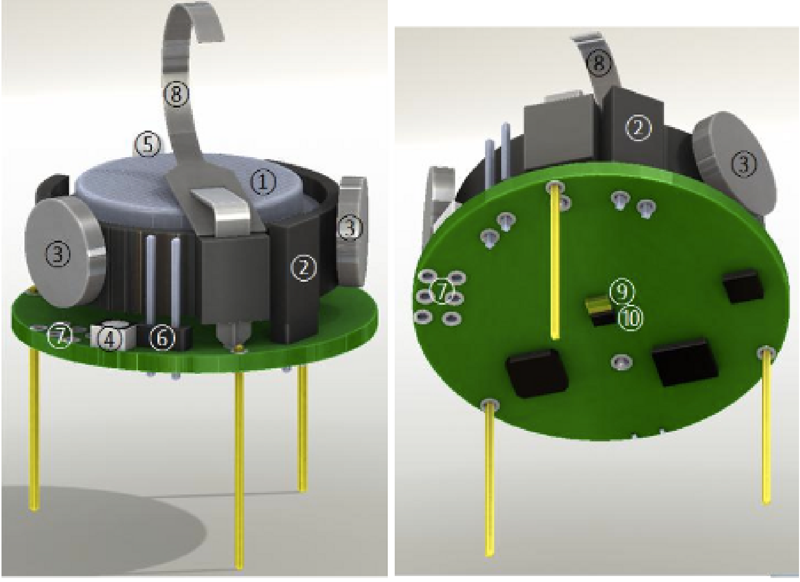

Figura 1: Kilobot e seus principais componentes: 1) bateria recarregável de 3.7 volts; 2) power jumper; 3) motores de vibração; 4) led RGB; 5) sensor de luz ambiente; 6) porta de comunicação serial; 7) socket para programação; 8) haste de recarregamento; 9) transmissor infravermelho; 10) receptor infravermelho. *(Fonte:)*

A programação dos robôs é realizada em linguagem C AVR, a mesma utilizada na programação da plataforma arduino, com funções definidas pela *Kilolib API*[2], que provê acesso aos recursos fornecidos pelos Kilobots. A gravação dos programas nos Kilobots é feita através do *Overhead Controler* (OHC) que é um dispositivo utilizado para acessar e controlar algumas funcionalidades dos robôs como, por exemplo, iniciar/pausar a execução de um algoritmo, gravar neles o código compilado, calibrar motores e sensores, entre outras. O OHC utiliza emissores de infravermelho para controlar todos os Kilobots simultaneamente, e para tanto, precisa ser elevado a uma certa altura acima deles. A Figura 2 mostra como o OHC deve ser instalado em relação aos Kilobots.

[1] https://ssr.seas.harvard.edu/kilobots
[2] https://www.kilobotics.com/docs/index.html

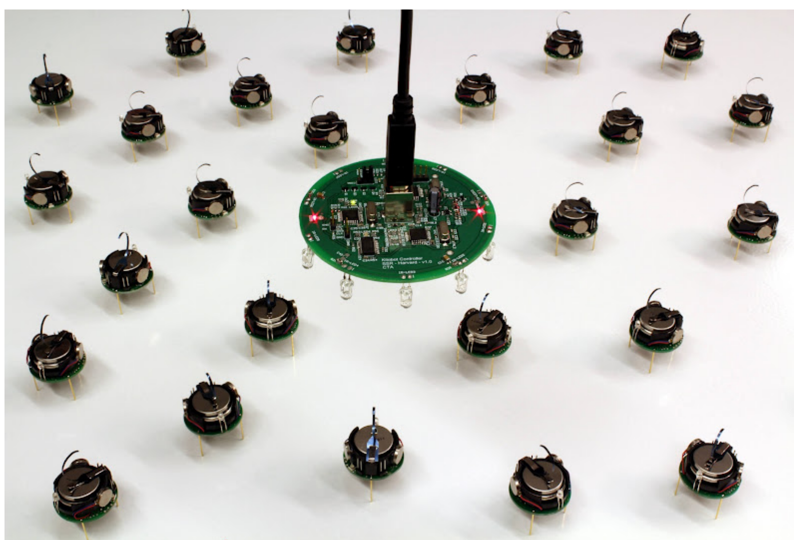

Figura 2: Kilobots e OHC. *(Fonte:* ↗*)*

2 Introdução ao Simulador

O Kilombo é um simulador para os robôs Kilobots baseado em linguagem C ANSI. Seu desenvolvimento está associado ao projeto *Swarm Organ*[3] que pesquisa abordagens na área de auto-organização em enxames utilizando estratégias de controle cooperativo e distribuído. A grande vantagem deste simulador em relação a outros é que embora seja baseado em liguagem C ANSI, o código fonte escrito para para programar os robôs no simulador pode ser compilado diretamente para o Kilobot sem necessidade de compatibilizá-lo com o C AVR. Isso possibilita o desenvolvimento de algoritmos complexos utilizando diretamente o simulador, uma vez que os mesmos terão comportamento muito próximo quando carregados nos Kilobots[4].

2.1 Instruções de Instalação

O projeto original do simulador está disponível em http://jic-csb.github.io/kilombo/, inclusive com as instruções de instalação, dependências e pré-requisitos. Entretanto, neste curso, por questões de comodidade preferimos fazer uso de uma versão auto-contida do Kilombo, de forma que se elimina a necessidade de instalação do simulador e das bibliotecas auxiliares exigidas.

Os pré-requisitos básicos para se executar os exemplos e resolver os exercícios propostos são:

- sistema operacional Linux (Ubuntu ou derivados são recomendados)
- git
- cmake, make
- avr-gcc (compilador C para processadores AVR)

Os demais pré-requisitos foram eliminados na versão auto-contida do simulador.

Todos os exercícios apresentados neste documento podem ser desenvolvidos na versão auto-contida do simulador. Basicamente, para se obter acesso a esta versão, deve-se baixar os arquivos do repositório https://github.com/sidneyrdc/kilombo_example. Para isso, abra um terminal linux e execute os seguintes comandos:

[3] http://www.swarm-organ.eu/

[4] Para mais informações acerca dos Kilobots desenvolvidos na UFSC, consultar http://kilobots.paginas.ufsc.br

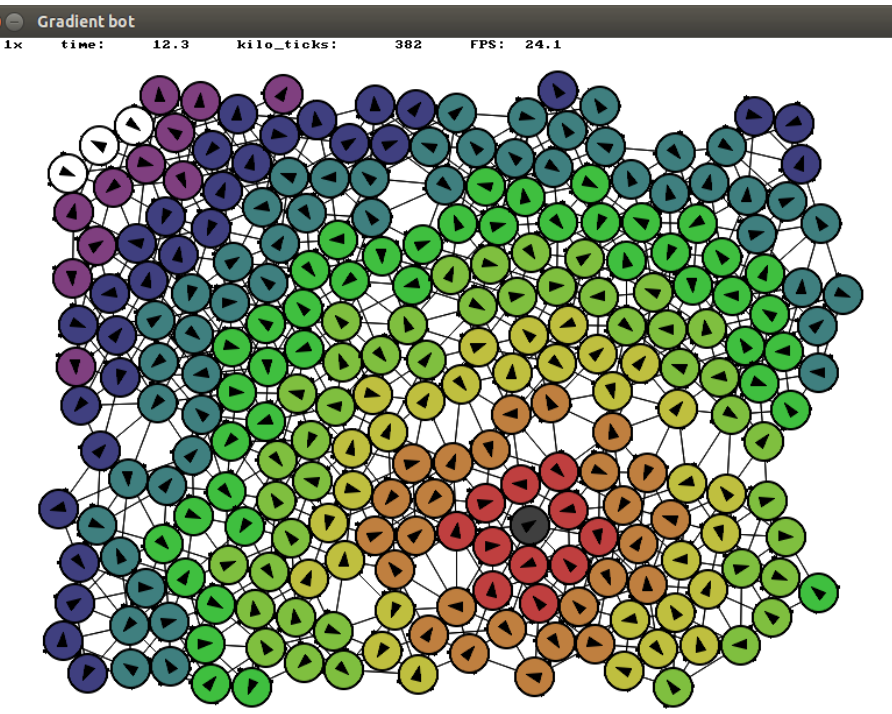

Figura 3: Kilobots simulados no Kilombo.

```
$ cd Desktop/
$ git clone https://github.com/sidneyrdc/kilombo_example
```

Estes comandos baixam os arquivos no Desktop do seu computador na pasta `kilombo_example`. Uma vez que os arquivos estiverem baixados, o usuário tem acesso aos exemplos de uso do simulador, bem como a um arquivo principal que deverá ser utilizado nas atividades propostas. Basicamente este diretório é composto por cinco pastas:

- `include` que é a pasta onde há os *headers* das bibliotecas usadas;
- `lib` que é a pasta onde se encontram os binários das bibliotecas utilizadas;
- `LICENSE` que é um arquivo de licença, `MakeFile` que é um arquivo de configuração do `make` (utilizado para compilar os códigos fontes);
- `src` que é pasta onde há os códigos fontes dos exemplos e o arquivo principal a ser utilizado nos exercícios.

Efetivamente, neste curso apenas os arquivos encontrados na pasta `src` serão alterados.

2.2 Como Compilar e Executar Programas

Os arquivos fornecidos já contam com um arquivo `Makefile` pré configurado, logo é necessário apenas se executar os comandos derivados do `make` para se compilar e gerar os arquivos binários (executáveis) para os exemplos e/ou exercícios propostos.

Para se compilar o arquivo `main.c`, disponibilizado como arquivo de base para os exercícios, deve-se executar os seguintes comandos:

```
$ cd kilombo_example/src
$ make
```

Isso resultará em um arquivo binário `main` que pode ser executado através da seleção com o mouse ou através da linha de comando:

```
$ ./main
```

Note que o arquivo binário que foi gerado, é o próprio simulador com os Kilobots se comportando conforme programa no arquivo `*.c`.

Para se compilar os exemplos dentro da pasta `kilombo_example/src/examples` o processo é um pouco diferente. Deve-se especificar explicitamente o nome do exemplo, que é o nome do arquivo `*.c`, a ser compilado durante a chamada do comando `make`:

```
$ make TARGET=gotolight
```

Esse comando gera um arquivo binário com o nome do exemplo, nesse caso `gotolight`, na própria pasta onde o comando foi executado, `kilombo_example/src/examples`.

2.3 Como Compilar Programas para os Kilobots Reais

Uma vez que o simulador Kilombo é compatível com a *Kilolib API* utilizada na programação dos Kilobots, um código escrito neste simulador pode ser compilado diretamente para os Kilobots. Para isso, bastando adicionar o termo `hex` no final do comando `make`:

```
$ make hex
```

que gera o arquivo `main.hex`, ou

```
$ make TARGET=gotolight hex
```

que gera o arquivo `gotolight.hex`. Cada um dos arquivos `*.hex` pode ser enviado diretamente para os Kilobots utilizando a interface disponibilizada para este fim.

2.4 Principais Atalhos

Durante a simulação, estão disponíveis algumas teclas de atalho para as principais funções do simulador descritas na Tabela 1.

Além do teclado, o mouse também pode ser utilizado para acessar a interface do simulador. Basicamente o mouse pode ser utilizado para mover e girar os robôs no cenário.

3 Programação

O simulador é compatível com as definições originais do código dos Kilobots, apresentada pela *Kilolib API*. Os processos de inicialização, troca de mensagens, movimentação e geração de números aleatórios funcionam através das mesmas chamadas de funções no simulador e nos Kilobots reais. Entretanto, há três tópicos que exigem atenção especial na programação para o simulador: **tamanhos dos tipos de dados**, **variáveis globais** e **temporização**.

3.1 Tipos de Dados

A diferença entre o compilador C AVR utilizado para os Kilobots reais e o compilador C ANSI utilizado quando o código é compilado para o simulador é o tamanho dos tipos de dados. Por exemplo, em C AVR o tipo `int` tem

Tecla	Ação
`ESC`	terminar a simulação
`+`	aumentar o zoom
`-`	diminuir o zoom
`*` ou `F4`	aumentar a velocidade da simulação
`/` ou `F3`	diminuir a velocidade da simulação
`arrow-keys`	mover a tela
`space`	pausar/continuar simulação
`mouse-left`	mover o robô de um ponto a outro
`mouse-right`	girar o robô (mudar a orientação)
`s`	salvar um print da tela
`v`	salvar vários prints da tela para criação de vídeos, usando o parâmetro `imageName`
`F1`	dispersar os robôs
`F2`	aproximar os robôs
`F5`	chamar a função de recarga de parâmetros (se implementada)
`F6`	chamar a função de reset em cada robô (se implementada)
`F11`	ativar máxima velocidade de simulação (sem delay entre os frames)
`F12`	ativar máxima velocidade de comunicação (mensagens enviadas a cada tick de relógio)

Tabela 1: Principais teclas de atalho para o simulador.

16 bits, enquanto que em C ANSI tem 32 ou 64 bits, dependendo do sistema operacional/hardware utilizados no computador. Normalmente o tamanho dos tipos de dados não deveria ser um problema, a não ser em casos em que ocorra o uso intencional de *overflow* sobre algumas variáveis. Entretanto, o uso não explícito dos tamanhos dos tipos de dados pode resultar em um código funcional no simulador que causa *overflow* nos Kilobots e que, portanto, não funciona corretamente na prática. Obviamente isso só é relevante, se deseja-se em algum momento usar o código em Kilobots reais.

Para lidar com essa situação a melhor solução é trabalhar apenas com tipos de dados com tamanho explícito, isto é, especificar o tamanho do tipo de dado na declaração das variáveis. Por exemplo, ao invés de se declarar `int i` deve-se declarar `uint8_t i` para um inteiro de 8 bits. Os tipos de dados com tamanho explícito são os mesmos definidos pela biblioteca `stdint.h`[5] utilizada em C ANSI.

3.2 Estrutura Básica de um Programa para Kilobots/Kilombo

Um programa para o simulador Kilombo deve começar com a inclusão do arquivo `kilombo.h`, que é distribuído com o simulador e pode ser encontrado na pasta `kilombo_example/include`:

```
#include <kilombo.h>
```

A sintaxe original para os Kilobots, baseada na *Kilolib*, normalmente faz uso de variáveis estáticas (para permitir que estas persistam entre as diversas chamadas de função providas pelo usuário) ou variáveis globais. Essas variáveis necessitam de um tratamento especial quando implementadas no simulador. Como o simulador lida com todos os robôs em um único programa, as variáveis globais se tornam comuns a todos os robôs. Isso é indesejado, uma vez que na aplicação real isso não ocorre. Para lidar com essa situação, o simulador exige que todas as variáveis globais sejam declaradas dentro de uma única `struct`. Além disso, essa `struct` deve ser instanciada para o simulador por meio da macro `REGISTER_DATA`, como segue:

[5] https://en.wikibooks.org/wiki/C_Programming/stdint.h

```
typedef struct {
    uint8_t n_neighbors;
    ...
} USERDATA;

REGISTER_USERDATA(USERDATA);
```

As variáveis declaradas dentro da `struct` podem ser acessadas usando o operador `->` da notação de ponteiros, por exemplo, `mydata->n_neighbors`. O simulador garante que `mydata` aponta para os dados do robô que está sendo executado. Note-se que variáveis locais, isto é, variáveis regulares declaradas dentro de uma função, podem ser utilizadas da forma usual.

O código desenvolvido no simulador contém, como é típico em C, uma chamada à função `main` que deve ter basicamente a seguinte estrutura de comandos:

```
// main function
int main(void) {
    // initialize the hardware of the kilobots
    kilo_init();

#ifdef DEBUG
    // setup debugging, i.e. printf to serial port, in real Kilobot
    debug_init();
#endif

    // register a callback function to return a string describing the internal
    // state of the current bot, used for the simulator status bar
    SET_CALLBACK(botinfo, botinfo);

    // register message callback
    kilo_message_rx = message_rx;

    // register message transmission callback
    kilo_message_tx = message_tx;

    // start kilobot event loop
    kilo_start(setup, loop);

    return 0;
}
```

Cada comando tem a seguinte finalidade:

- `kilo_init()`: inicializa o *hardware* do Kilobot. Isso inclui calibração do oscilador de *hardware*, a definição dos *timers*, a configuração das portas, entre outras funcionalidade. Deve ser a primeira função a ser chamada na função `main` do seu programa;

- `debug_init()`: inicializa a comunicação serial do Kilobot, de forma que pode-se utilizar a instrução `printf` padrão da linguagem C para se escrever dados na porta serial, o que por sua vez possibilita posterior leitura no computador.

- `SET_CALLBACK(ID, function)`: é uma macro que permite registrar funções de *callback*, isto é, que são executadas a partir de interrupções disparadas por eventos específicos. Desta forma, pode-se simular obstáculos, pontos de luz, recarregar a configuração dos robôs simulados sem reiniciar a simulação, entre outras coisas. Todas as funções de *callback* são opcionais e não atuam nos Kilobots reais. A Tabela 2 apresenta todas as possíveis funções de *callback* disponibilizadas pelo simulador.

- `kilo_message_rx = rx_message`: é uma variável global que aponta para uma função de *callback* que processa a recepção de mensagens. É chamada sempre que uma nova mensagem é decodificada com sucesso. A função `rx_message`, cujo ponteiro é associado a esta variável deve conter o código de tratamento das mensagens recebidas, i.e., armazenamento das mensagens em um buffer para posterior extração de dados. Note-se que todas as funções de *callback* devem ser registradas antes da chamada da função `kilo_start`.

- `kilo_message_tx = tx_message`: é uma variável global que aponta para uma função de *callback* que processa o envio de mensagens. É chamada sempre que uma nova mensagem é escalonada para transmissão (aproximadamente duas vezes por segundo). A função `tx_message`, cujo ponteiro é associado a esta variável deve conter o código de tratamento das mensagens a serem enviadas, i.e., preenchimento dos dados a serem enviados, etc.

- `kilo_start(void(*)(void) setup, void(*)(void) loop)`: inicializa o laço de eventos para os Kilobots, i.e., os códigos de inicialização e laços de execução programados pelo usuário. O primeiro parâmetro (`setup`) é um ponteiro para uma função que será chamada uma vez para executar qualquer tarefa de inicialização requerida pelo usuário. O segundo parâmetro (`loop`) é um ponteiro para uma função que será chamada repetidamente para executar qualquer computação requerida pelo usuário (é onde se coloca o código principal que será repetido).

ID	Exemplo de Chamada da Função	Uso
`params`	`void callback_F5(void)`	Recarrega parâmetros de configuração a partir de um arquivo. Chamada uma vez quando F5 é pressionado.
`reset`	`void callback_F6(void)`	Reinicia os robôs. Chamada para todos os robôs quando F6 é pressionada.
`botinfo`	`char *botinfo()`	Retorna uma string descrevendo o estado interno do robô selecionado com o mouse, é utilizada para mostrar dados na barra de status.
`json_state`	`json_t *json_state(void)`	Retorna um objeto json descrevendo o estado interno de cada robô, é utilizada para salvar *snapshots* da simulação.
`global_setup`	`void callback_global_setup(void)`	Executa uma configuração global através da leitura de parâmetros específicos. Chamada uma vez, após o arquivo contendo os parâmetros ser lido, mas antes da configuração dos robôs.
`lighting`	`int16_t callback_lighting(double, double)`	Define os níveis de luz no ambiente simulado.
`obstacles`	`int callback_obstacles(double, double, double*, double*)`	Define obstáculos a serem simulados.

Tabela 2: Funções de *callback* disponibilizadas pelo simulador.

3.3 Temporização e Delays

O simulador não implementa a função `delay()`, como ocorre na *Kilolib* original. A função de fato existe, mas retorna imediatamente, não tendo efeito algum sobre a execução do código. A razão para esse comportamento é simples: o simulador executa a mesma função `main` para todos os robôs, desta forma uma função similar a `delay` causaria um comportamento temporal errôneo, uma vez que todos os robôs seriam afetados.

Para lidar com questões temporais deve-se optar pela utilização da variável global `kilo_ticks` que é incrementada pelos Kilobots/simulador exatamente 31 vezes por segundo, oferecendo assim acesso às informações temporais precisas sem comprometer o funcionamento do simulador. Como um exemplo de uso, pode-se substituir a seguinte chamada de função:

```
...
delay(2000); // waits 2 seconds to continue
...
```

por uma avaliação da variável `kilo_ticks`:

```
uint32_t saved_time = kilo_ticks;            // stores the current time for posterior use
...
if(kilo_ticks - saved_time >= 2*31) { ... }  // compares current time with the stored one
...
```

3.4 Comunicação

Como descrito anteriormente, a associação de funções personalizadas a serem executadas quando eventos de comunicação ocorrem é feita por meio das variáveis globais `kilo_message_rx` e `kilo_message_tx`. Essas funções personalizadas devem possuir uma declaração em particular, geralmente relacionada a sua utilidade quando mensagens são recebidas ou transmitidas.

No geral uma função associada à variável de *callback* `kilo_message_rx` (i.e., chamada sempre que uma nova mensagem é recebida) deve ser da forma:

```
// receive message callback
void rx_message(message_t *msg, distance_measurement_t *d) {
    mydata->new_message = 1;                // sets the indicator for new messages as true
    mydata->received_msg = *msg;            // stores the received message
    mydata->dist = estimate_distance(d);    // estimates the distance to the sender
}
```

em que os parâmetros `*msg` e `*d` são ponteiros para a mensagem recebida e para a medida de distância obtida a partir da intensidade do sinal entre os robôs, respectivamente.

A medida de distância `*d` é um struct definido pela *Kilolib*, formado pelos seguintes campos:

```
typedef struct {
    int16_t low_gain;
    int16_t high_gain;
} distance_measurement_t;
```

onde `low_gain` e `high_gain` são valores de 10 bits (0–1023) que representam o nível de sinal coletado pelo sensor infravermelho do robô receptor da mensagem. Note-se que para extrair a distância em valores inteiros a partir de `*d` é preciso utilizar a função `estimate_distance(const distance_measurement_t *d)` que retorna um valor do tipo `uint8_t` correspondendo à distância em milímetros entre o robô emissor e o robô receptor da mensagem.

Um ponto importante a respeito da função `rx_message` é que ela não deve possuir chamadas de código complexas, nem laços, uma vez que sua chamada ocorre por meio de interrupções assíncronas a cada recebimento de uma nova mensagem. Logo é preciso processar a mensagem recebida em uma outra função para extrair dela a informação transmitida, da seguinte forma:

```
// process the received message
void process_message() {
    // check if there are new messages
    if(new_message == 0) return;

    // auxiliary index
    int8_t i;

    // get the received data from the message
    uint8_t *data = mydata->received_msg.data;

    // get sender id
    uint16_t id = data[0] | (data[1] << 8);

    ...

    // nullify the new message indicator
    mydata->new_message = 0;
}
```

Quando há mais de um vizinho para cada robô, deve-se utilizar um buffer para armazenar todas as mensagens recebidas, uma vez que a frequência de execução da função `rx_message` é diferente da frequência de execução da função `process_message`, que em geral é determinada pelo parâmetro `timeStep` (uma vez que esta deve ser chamada na função principal `loop`). No arquivo base `main.c` as funções `rx_message` e `process_message` estão implementadas considerando essa questão.

Já a função associada à variável de *callback* `kilo_message_tx` deve ser da forma:

```
// message transmission callback
message_t *tx_message() {
    return &mydata->transmit_msg;
}
```

ou seja, deve retornar um ponteiro para a mensagem a ser enviada. Note-se mais uma vez que a mensagem a ser enviada não é processada dentro da função `tx_message`, pois assim como no caso da função `rx_message`, esta não deve conter instruções complexas ou laços.

Para tanto, usualmente se opta por escrever uma outra função que deve construir a mensagem (isto é, preencher a mensagem com *payload*, definir o tipo de mensagem, etc.), cuja chamada deve ser feita dentro da função `loop`, resultando em:

```
// fill the message with the bot information
void setup_message() {
    // sets the message type
    mydata->transmit_msg.type = NORMAL;

    // fill up the data to be sent
    mydata->transmit_msg.data[0] = kilo_uid & 0xff;             // 0 low  id
    mydata->transmit_msg.data[1] = kilo_uid >> 8;               // 1 high id

    // compute the CRC of the message struct
    mydata->transmit_msg.crc = message_crc(&mydata->transmit_msg);
}
```

Cada mensagem a ser enviada é na verdade uma `struct` definida na *Kilolib API* com três campos bem definidos:

```
// message structure
typedef struct {
    uint8_t data[9]; // message payload
    uint8_t type;    // message type
    uint16_t crc;    // message crc
} message_t;
```

em que `data[9]` é um vetor de 9 posições do tipo `uint8_t` (um array de bytes) que deve conter o *payload* da mensagem (ou seja, sua carga útil). Os campos `type` e `crc` são para controle interno do simulador/robô e verificação da integridade das mensagens, respectivamente.

3.5 Atuação

Como recursos de atuação, os Kilobots possuem motores que provocam a vibração que lhes permite movimentar-se, e LEDs RGB que podem ser usados para indicar o estado interno dos robôs, bem como as interações entre eles. Ambas as funções são ativadas da mesma forma tanto no simulador quanto nos robôs reais.

Para mover os robôs, a *Kilolib* implementa a função `set_motors(uint8_t left, uint8_t right)` que define o nível do duty-cycle do sinal de PWM (0 até 255) em cada um dos motores. Em outras palavras, definir um motor com 0% de duty-cycle equivale a desligar o motor, enquanto que definir com 100% de duty-cycle equivale a ligar o motor em sua potência máxima. Como os Kilobots reais devem ser calibrados para que os melhores valores de potência sejam definidos individualmente, deve-se fazer uso apenas das seguintes constantes, que buscam esses valores a partir da calibração: `kilo_turn_left`, `kilo_turn_right`, `kilo_straight_left`, `kilo_straight_right`. Desta forma, um exemplo de uso pode ser:

```
// turn motors off
set_motors(0, 0);

// spin up left motor
set_motors(255, 0);

// turn the kilobot left
set_motors(kilo_turn_left, 0);
```

```
// turn the kilobot right
set_motors(0, kilo_turn_right);

// go straight
set_motors(kilo_turn_left, kilo_turn_right);
```

Os LEDs podem ser acionados pela função `set_color(uint8_t color)` que recebe um inteiro de 8-bits sem sinal para determinar as cores do LED RGB montado no Kilobot. Cada cor tem 2-bits de resolução, o que permite que cada canal de cor seja definido independente entre desligado (0) até máximo brilho (3).

Por questões de conveniência a macro `RGB` é definida pela *Kilolib* para ligar os canais de cores de maneira individual. Por exemplo, `RGB(0,0,0)` desliga todas cores, e portanto o LED permanece desligado. Enquanto que `RGB(0,3,0)` liga o canal verde em intensidade máxima e desliga todas as demais cores, o que resulta no LED mostrando a cor verde.

Um exemplo de funcionamento da função `set_color` e da macro `RGB`:

```
// set led color as red
set_color(RGB(1,0,0));

// set led color as green
set_color(RGB(0,3,0));

// set led color as blue
set_color(RGB(0,0,3));
```

3.6 Sensoreamento

Entre os sensores disponíveis para os Kilobots estão o sensor de incidência luminosa, o sensor de temperatura, e o sensor de nível da bateria. Destes sensores, apenas o primeiro está disponível no simulador.

Para acessar o sensor de luz deve-se utilizar a função `get_ambientlight` que retorna um valor de 10 bits (0 até 1023) que representa o nível de luz ambiente detectado pelo fotodiodo utilizado como sensor. Quando esse nível não pode ser medido (por exemplo, se o conversor analógico-digital estiver indisponível), esta função retorna `-1`.

Exemplo de uso:

```
// stores the light level
uint16_t light_level = get_ambientlight();
```

3.7 Parâmetros de Configuração

O simulador suporta a passagem de diversos parâmetros de configuração da simulação. A maioria se dá pela utilização de um arquivo `JSON` que contém as configurações iniciais da simulação. Este arquivo deve estar no mesmo diretório do binário a ser executado e deve ser nomeado como `kilombo.json`. Este arquivo deve conter alguns dos parâmetros listados na Tabela 3. Entre os arquivos baixados há uma versão desse arquivo já configurada.

Segue um exemplo desse arquivo inicialmente configurado com as opções mais significantes para os exercícios propostos neste curso:

```
{
    "botName": "test",
    "randSeed": 1,
    "nBots": 5,
    "formation": "line",
    "timeStep": 0.0416666,
    "simulationTime": 0,
    "commsRadius": 70,
    "showCommsRadius": 1,
```

Parâmetro	Tipo	Padrão	Uso
botName	string	"default"	nome da simulação (a ser exibido como título da janela)
randSeed	int	0	semente aleatória, para reproduzir simulações iguais
simulationTime	float	0	quanto tempo executar a simulação
timeStep	float	0.02	intervalo de tempo entre cada ciclo de simulação
nBots	int	1	número de robôs a serem simulados
formation	string	"random"	formação inicial dos robôs: "random", "line", "pile", "circle", "ellipse"
distributePercent	float	0.2	distribuir os robôs sobre essa porcentagem da tela
commsRadius	int	70	raio de comunicação dos robôs em mm
msgSuccessRate	float	1.0	probabilidade de sucesso na transmissão de mensagens
distanceNoise	float	0	estocasticidade das medidas de distância (desvio padrão)
distanceCoefficient	float	1	inclinação na função de distância entre os robôs
speed	float	7	velocidade de movimentação dos robôs em mm/s
turnRate	float	13	velocidade de giro dos robôs em deg/s
pushDisplacement	float	1.0	deslocamento de robôs estacionários ao serem empurrados
displayWidthPercent	float	0.9	proporção do comprimento da largura da janela
displayHeightPercent	float	0.9	proporção da altura da janela
displayWidth	int	1	valor absoluto do comprimento da janela em pixels
displayHeight	int	1	valor absoluto da altura da janela em pixels
displayScale	float	1.0	zoom inicial
displayX	float	0.0	coordenadas iniciais do centro da janela
displayY	float	0.0	coordenadas iniciais do centro da janela
showHist	int	0	mostrar os caminhos percorridos pelos robôs
histLength	int	2000	tamanho do histórico de caminhos em número de ciclos de simulação
showComms	int	1	mostrar uma linha representando a comunicação entre os robôs
showCommsRadius	int	1	mostrar um círculo em volta dos robôs representando seu raio de comunicação
stepsPerFrame	int	1	número de ciclos de simulação entre quadros
GUI	int	1	mostrar (1) ou não (0) interface gráfica
colorscheme	string	"dark"	esquemas diferentes de cores: "dark" ou "bright"
saveVideo	int	0	salvar (1) ou não (0) prints da tela para uso em vídeos
saveVideoN	int	1	salvar prints da tela a cada N iterações
imageName	string	""	nome das imagens salvas durante a simulação, e.g., "movie/f%04d.bmp"
finalImage	string	""	nome da imagem do estado final da simulação, para desativar use "null"
stateFileName	string	""	nome do arquivo JSON para salvar o estado da simulação
stateFileSteps	int	100	frequência de salvação do estado da simulação (use 0 para desativar)
useGrid	int	1	usar cache de grade para encontrar os vizinhos (rápido para grandes enxames)

Tabela 3: Parâmetros a serem passados através do arquivo JSON.

```
    "showComms": 1,
    "distributePercent": 0.9,
    "displayWidth": 800,
    "displayHeight": 550,
    "displayScale": 0.75,
    "displayX": 1,
    "displayY": 1,
    "showHist": 1,
    "histLength": 10500,
    "storeHistory": 1,
    "imageName": "./movie/f%04d.bmp",
    "saveVideo":  0,
    "saveVideoN": 1,
    "stepsPerFrame": 1,
    "finalImage": null,
    "stateFileName": "simstates.json",
    "stateFileSteps": 0,
    "colorscheme": "bright",
    "GUI": 1,
    "msgSuccessRate": 0.8,
    "distanceNoise": 2
}
```

Além dos parâmetros passados através dos arquivos de configuração, o Kilombo também aceita a passagem direta de parâmetros pela linha de comando usando nomes personalizados para o arquivo JSON, por exemplo:

```
$ ./main -p parameterfile.json  # allows the use of a personal name to the configuration file
```

```
$ ./main -b bots.json            # starting positions for the robots
```

No final e, opcionalmente, durante a simulação o simulador salva o estado dos robôs como um arquivo JSON. O arquivo `endstate.json` contém o estado final da simulação, enquanto que os arquivos que são salvos durante a simulação (com uma frequência especificada por `stateFileSteps`) são nomeados de acordo com o parâmetro `stateFileName`. Ambos podem ser utilizados como o estado inicial em um outra simulação, bastando para isso apenas que se mude o seu nome (no caso do `endstate.json`) e se passe esse nome juntamente o parâmetro `-b` na chamada do binário da simulação, na linha de comando.

O arquivo JSON gerado contém um vetor nomeado `bot_states`. Cada elemento deste vetor contém os dados para um robô descritos através das chaves especificadas na Tabela 4.

Chave	Valor
`ID`	identificador único de cada robô
`direction`	ângulo de apontamento do robô (yaw) em radianos
`x_position`	coordenada no eixo x, em milímetros
`y_position`	coordenada no eixo y, em milímetros
`state`	objeto JSON descrevendo o estado interno do robô, fornecido pela função de *callback* `json_state`

Tabela 4: Conjunto de chaves e valores disponibilizados nos arquivos JSON de estado.

4 Exercícios

Nesta seção são apresentados alguns exercícios práticos de programação distribuída usando os Kilobots e o simulador Kilombo. Para tanto, deve-se utilizar o arquivo base `main.c` contido na pasta `kilombo_example/src` que já possui as funções de comunicação e outras utilidades pré-configuradas. O arquivo de configuração `kilombo.json` contém as definições padrões para se executar um programa no Kilombo e pode ser alterado pelo usuário.

4.1 Formação de Gradiente

- **Objetivo:** Neste exercício cada robô deve computar sua distância (medida em número de saltos de informação – *hops*) até um robô "fonte".
- **Recursos envolvidos:** comunicação e LEDs.

Este comportamento é inspirado pela ideia de gradientes morfogênicos e formação de padrões no desenvolvimento de organismos multicelulares: imagine uma célula particular sendo a fonte de uma substância química morfogênica, que se difunde e degrada à medida que passa sobre uma superfície 2D de células. As células que não são fontes podem registrar ou limitar o valor do gradiente morfogênico e, em seguida, usá-lo para decidir qual parte do padrão se tornar. Um exemplo clássico é o embrião da mosca da fruta, e o morfógeno Bicoid, que permite que o embrião se divida amplamente nas regiões da cabeça, do tórax e do abdômen. Em termos gerais, um gradiente define uma noção de distância de uma fonte porque seu valor diminui à medida que este se afasta da fonte.

Neste exercício, deve-se criar um gradiente que aumenta em valor conforme se afasta da fonte. O robô fonte emitirá um gradiente com valor 0, e então os robôs vizinhos vão armazenar o menor valor que eles receberem, em seguida incrementar esse valor em 1, e então enviá-los para todos os seus vizinhos. Desta forma, a difusão do gradiente seguirá o fluxo de informação a partir do robô fonte. Para que o valor do gradiente em cada robô seja conhecido, deve-se mudar a cor dos LEDs RGB de acordo com este valor.

Primeiro deve-se criar uma constante que define quem é o robô fonte. Essa constante conterá o valor do ID do robô fonte:

```
...
#define SEED_ID 0  // the ID of the source bot
...
```

Cada robô deverá armazenar o valor do seu gradiente e do gradiente dos seus vizinhos, para todos os seus vizinhos. O arquivo `main.c` a ser utilizado já contém a definição dos duas `structs` necessárias para se fazer isso:

```
...

// neighbor datatype
typedef struct {
    uint16_t id;
    uint8_t dist;
    uint8_t n_neighbors;
    uint32_t timestamp;

    // insert the gradient value for the neighbors here
    uint16_t gradient;
} neighbor_t;

// declare global variables
typedef struct {
    neighbor_t neighbors[MAXN];
    uint8_t n_neighbors;

    char new_message;
    char message_lock;
    message_t transmit_msg;
    received_message_t received_msg[MAXMSG];
    uint16_t n_messages;

    // insert the gradient value for the robot here
    uint16_t gradient;
} USERDATA;

...
```

Cada robô deverá transmitir o seu valor de gradiente e extrair das mensagens recebidas o valor de gradiente dos seus vizinhos. Para isso deve-se alterar as funções `setup_message` e `process_message`, inserindo as linhas:

```
// -----------------------------------------------------------------------
// put the data to be send below (limited to 9 bytes)
// -----------------------------------------------------------------------
...
mydata->transmit_msg.data[3] = mydata->gradient & 0xFF;   // 4 low  byte of gradient value
mydata->transmit_msg.data[4] = mydata->gradient >> 8;   // 5 high byte of gradient value
// -----------------------------------------------------------------------
```

na função `setup_message`, antes da geração do *checksum* da mensagem (no trecho indicado de código), e as linhas:

```
// -------------------------------------------------------------------
// extract the received data below (limited to 9 bytes)
// -------------------------------------------------------------------
...
mydata->neighbors[i].gradient = data[3] | data[4] << 8;
// -------------------------------------------------------------------
```

na função `process_message`, no trecho indicado pelos comentários.

Uma vez que os robôs são capazes de enviar e receber o valor do gradiente, pode-se implementar a função principal. O aluno deve, portanto, programar a função `compute_gradient`, pré-definida como:

```
// primitive behavior gradient formation
void compute_gradient() {
    uint8_t i;
    uint16_t min = UINT16_MAX-1;

    ...
```

```
}
```

de forma que o robô cujo ID é `SEED_ID` deve possuir o gradiente sempre igual a 0 e os demais devem incrementar 1 ao menor valor de gradiente recebido dos vizinhos. Para se obter o ID de um robô, faz-se uso da variável global `kilo_uid`.

Uma vez a função `compute_gradient` pronta, pode-se chamá-la na função `loop`, que será executada várias vezes.

```
// put your main code here, will be run repeatedly
void loop() {
    // build message
    setup_message();

    // process received message
    process_message();

    // compute the gradient level for each robot
    compute_gradient();
}
```

Para mostrar o valor do gradiente com os LEDs RGB é preciso criar um vetor de cores utilizando a macro `RGB`, citada anteriormente:

```
// colors array
const uint8_t colors[] = {
    RGB(0, 0, 0),  //0 - off
    RGB(2, 0, 0),  //1 - red
    RGB(2, 1, 0),  //2 - orange
    RGB(2, 2, 0),  //3 - yellow
    RGB(1, 2, 0),  //4 - yellowish green
    RGB(0, 2, 0),  //5 - green
    RGB(0, 1, 1),  //6 - cyan
    RGB(0, 2, 2),  //7
    RGB(0, 1, 2),  //8
    RGB(0, 0, 1),  //9 - blue
    RGB(1, 0, 1),  //10 - purple
    RGB(2, 0, 1),  //11
    RGB(3, 3, 3)   //12  - bright white
};
```

que deve ser utilizado para definir a cor de cada robô de acordo com o seu nível de gradiente:

```
// put your main code here, will be run repeatedly
void loop() {
    ...

    // set the current color according to the gradient level
    set_color(colors[mydata->gradient%13]);
}
```

Finalmente, para compilar o código abra um terminal na pasta `kilombo_example/src` e digite `make` no terminal, se tudo estiver certo com o código, o compilador vai gerar um arquivo executável `main` que poderá ser executado pelo comando `./main` inserido no terminal ou pela seleção do mouse.

Note-se que ao se tentar executar o exemplo, uma janela se abrirá com 5 robôs em linha, e poder-se-á visualizar o gradiente se formando a partir do robô cujo ID é zero. Tente mover esse robô utilizando o mouse para ver como o gradiente se adapta. Isso só ocorre porque foi implementada uma função que faz o robô "esquecer" vizinhos mais antigos (i.e., que não enviam mais mensagens), em outro caso ao se mover o robô fonte, perceberia-se apenas a adaptação do gradiente da nova vizinhança e não da que fora deixada.

Agora edite o arquivo `kilombo.json`, alterando os parâmetros `nBots` e `formation`, da seguinte forma:

```
{
    ...
    "nBots": 250,
    "formation": "random",
```

```
    ...
}
```

isso adicionará 250 robôs ao simulador em formação aleatória. Perceba como a disseminação do gradiente ocorre entre robôs que compartilham vizinhos de mesmo *hop*.

4.2 Robôs em Órbita

- **Objetivo:** Usar a medida de distância entre os robôs para manter um deles estacionário enquanto o outro deve orbitá-lo a uma distância fixa.
- **Recursos envolvidos:** comunicação, movimentação e leds.

Em enxames de robôs e outros sistemas altamente densos, a ação orbitar é uma primitiva básica para comportamentos emergentes. Basicamente essa ação permite que um robô (o planeta) mantenha uma distância fixa em relação a outro robô (a estrela) durante sua movimentação. O mesmo comportamento pode ser estendido para o seguidor de bordas (*edge-following*) se houver um grupo de robôs estrelas, desta forma os planetas podem orbitar esse grupo. Em robótica clássica, a ação orbitar está relacionada a outras algoritmos como seguidor de linha, Phototaxis, etc.

Neste exercício, o robô definido como planeta deve orbitar o robô definido como estrela a uma distância fixa de 60 mm para manter um bom compromisso entre manter-se afastado sem perder a comunicação e não colidir com os outros robôs, uma vez que a distância de comunicação é de aproximadamente 70 mm e a distância mínima, em que há contato entre os robôs, é de 30 mm.

A ideia básica para resolver esse exercício é analisar a distância entre os robôs, se esta estiver abaixo de 40 mm, então o robô planeta está muito próximo do robô estrela. Neste caso, o robô planeta deve se mover para frente até que a distância entre eles seja maior do que 40 mm. Isso permite que o robô planeta chegue rapidamente a uma distância razoável da estrela e então inicie a órbita, no caso dele começar em uma posição que não deixa muito espaço para manobras. Uma vez que o robô planeta já não está mais tão próximo do robô estrela, ele deve apenas alternar entre virar à esquerda, quando a distância entre eles é menor do que a desejada, e virar à direita quando essa distância é maior do que a desejada. Esta lógica faz com que o robô planeta faça pequenos ajustes de posição em torno da distância desejada de 60 mm, enquanto se move para frente durante todo o tempo. Note-se que este comportamento faz com que o robô planeta orbite o robô estrela no sentido horário, pode-se trocar a ordem das ações de ajuste de movimentação para se obter uma órbita no sentido contrário.

O arquivo `main.c` já contém definições básicas para transmissão e recepção de mensagens e armazenamento de informações dos vizinhos, tais como distância. Portanto apenas as funções `loop` e `setup` serão alteradas. Além delas, uma nova função chamada `compute_orbit` deve ser usada para calcular o movimento de cada robô planeta.

Primeiramente deve-se definir as constantes para distância mínima, distância desejada e para o ID do robô estrela:

```
...

#define TOO_CLOSE 40            // minimum distance between the star and the planet
#define DESIRED_DISTANCE 60     // desired distance between the star and the planet
#define STAR_ID 0               // star robot's ID

...
```

Com as constantes definidas, deve-se programar a função `compute_orbit` da seguinte maneira:

```
// compute orbit according to the distance between the planet and the star
void compute_orbit() {
    uint8_t i;

    // if the robot is a planet, compute its movement
    if(kilo_uid != STAR_ID) {

        // search for the star ID among the neighbors
        ...
```

```
        // evaluate the distance to the star: go forward if its value is lower than TOO_CLOSE
        ...

        // go to left if the distance is lower than DESIRED_DISTANCE
        ...

        // go to right if the distance is equal or greater than DESIRED_DISTANCE
        ...

    // if the robot is the star, set its movement as stopped
    } else {
        ...
    }
}
```

Para melhor visualizar a tomada de decisão do robô planeta, pode-se definir a cor do LED RGB em cada uma das ações executadas, por exemplo, definir a cor como VERMELHO quando a distância é menor do que `TOO_CLOSE`, definir a cor como VERDE quando a distância é menor do que `DESIRED_DISTANCE`, definir a cor como AZUL quando a distância é maior do que `DESIRED_DISTANCE`.

Uma vez a função `compute_orbit` finalizada, pode-se chamá-la na função `loop` para esta ser executada repetidamente:

```
// put your main code here, will be run repeatedly
void loop() {
    // build message
    setup_message();

    // process received message
    process_message();

    // compute orbit according with to the robot type
    compute_orbit();
}
```

Finalmente, para compilar o código abra um terminal na pasta `kilombo_example/src` e digite `make` no terminal, se tudo estiver certo com o código, o compilador vai gerar um arquivo executável `main` que poderá ser executado pelo comando `./main` inserido no terminal ou pela seleção do mouse.

Antes de executar o binário gerado, deve-se alterar o arquivo `kilombo.json` de forma que este suporte o posicionamento dos robôs de maneira adequada para inicialização das posições:

```
{
    ...
    "nBots": 2,
    "formation": "random",
    ...
}
```

Além deste arquivo, deve-se criar um arquivo JSON chamado `start_pose.json` com a posições iniciais de cada robô:

```
{
    "bot_states": [
        {
            "ID": 0,
            "direction": 5.7269688373804088,
            "state": {},
            "x_position": -20.900863330484285,
            "y_position": 7.4895537075976648
        },
        {
            "ID": 1,
            "direction": -132.14180267595034,
            "state": {},
            "x_position": 36.053018924909047,
```

```
            "y_position": 25.928440210936703
        }
    ],
    "ticks": 7292
}
```

desta forma os robôs já inicializam dentro do raio de comunicação um do outro. Claro que há também a opção de utilizar o mouse para aproximar os robôs, no entanto, a utilização de um script com as posições iniciais permite um maior controle sobre a simulação.

Uma vez que os arquivos forem alterados, para se executar o arquivo binário deve-se abrir um terminal em `kilombo_example/src` e inserir a seguinte linha de comando:

```
$ ./main -b start_pose.json
```

que irá abrir a janela do simulador com as posições iniciais definidas pelo arquivo `start_pose.json`.

Agora teste seu código! Tente perturbar o planeta e veja como ele tenta se recuperar desta perturbação. Mova o robô estrela, e perceba como o robô planeta tenta acompanhar a sua variação de posição. Finalmente, tente adicionar mais uma ou duas estrelas à simulação, tomando cuidado de sempre verificar a distância entre o planeta e a estrela mais próxima ao invés das duas distâncias simultâneas. O robô planeta deverá orbitar as duas estrelas ou quantas mais forem adicionadas, de acordo com as posições definidas no arquivo `start_pose.json`.

Referências

[1] M. Rubenstein, C. Ahler e R. Nagpal. “Kilobot: A Low Cost Scalable Robot System for Collective Behaviors”. Em: *2012 IEEE International Conference on Robotics and Automation*. Saint Paul, MN, USA: IEEE, 2012, pp. 3293–3298. DOI: 10.1109/ICRA.2012.6224638.

[2] M. Rubenstein, A. Cornejo e R. Nagpal. “Programmable Self-assembly in a Thousand-Robot Swarm”. Em: *Science* 345.6198 (2014), pp. 795–799. DOI: 10.1126/science.1254295.

[3] M. Gauci, R. Nagpal e M. Rubenstein. “Programmable Self-Disassembly for Shape Formation in Large-Scale Robot Collectives”. Em: *13th International Symposium on Distributed Autonomous Robotic Systems*. London, UK, 2016, pp. 1–14. DOI: 10.1007/978-3-319-73008-0_40.

[4] F. Jansson et al. *Kilombo: a Kilobot Simulator to Enable Effective Research in Swarm Robotics*. arXiv. 2016. arXiv: 1511.04285.