SIMULADOR DE COMUNICAÇÕES ENTRE REDES DE SENSORES E VEÍCULOS AUTÓNOMOS

Daniel Filipe Arada de Sousa

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2012

Relatório da Disciplina de Seminário/Estágio, do 3º ano, da Licenciatura em Engenharia Electrotécnica e de Computadores

Candidato: Daniel Filipe Arada de Sousa, Nº 1000146, 1000146@isep.ipp.pt

Orientação científica: Jorge Manuel Estrela da Silva, jes@isep.ipp.pt

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

04-novembro-2012

Agradecimentos

Agradeço o apoio inconstitucional da minha família e amigos que como sempre quebram a rocha que impede o procedimento da caminhada - Não funciona, mas porquê? É sempre bom ter resposta. E conseguir fazer mais do que apenas uma coisa ao mesmo tempo? Fantástico.

Não posso deixar este capitulo sem apresentar o meu muito obrigado por todo o apoio do Professor Jorge Estrela, responsável pela ideia e orientação.

A todos os que iniciaram agora a leitura, seja por obrigação, curiosidade ou desespero, o meu muito obrigado.

Escrevo com a esperança de conduzir os vossos olhos por um conjunto de palavras, durante uma caminhada alucinante, capaz de traduzir algum conhecimento e frescura.

Resumo

Enquadrado numa linha de investigação ativa, na qual se pretende estudar diferentes cenários de aquisição de informação/dados em localizações remotas com recurso a veículos autónomos com base em simulação computacional, tornou-se necessária a estruturação e desenvolvimento de código. Surge, então, o problema a resolver.

Pretende-se desenvolver uma biblioteca, que para simplificação de conceitos designamos de biblioteca de sensores, capaz de interagir com código de simulação existente, testado e desenvolvido em linguagem C++.

Tendo em conta que num conjunto de sensores surge a hipótese de mais do que um sensor estar habilitado a comunicar no mesmo instante, facilmente entramos numa situação em que se considera que simultaneamente temos mais do que um sensor a comunicar e/ou são ultrapassados os limites de débito permitidos pelo canal de comunicação. Logo, a referida biblioteca deverá gerir a informação a cada instante e aplicar o devido escalonamento da largura de banda do canal de comunicação.

Assim, foram desenvolvidos os conhecimentos, conceitos e métodos de programação em linguagem C++, foram estudadas e analisadas as diferentes hipóteses de desenvolvimento e definidos os objetivos chave a ter em consideração no desenvolvimento.

Com base na informação recolhida e de uma forma geral, foram definidos os seguintes objetivos chave: abstração na codificação, para permitir a reutilização de código e expansão futura; simplicidade de conceitos e políticas de escalonamento, para permitir a validação do correto funcionamento da biblioteca; e otimização de performance de execução, para permitir a utilização da biblioteca com simuladores extremamente “pesados” em termos de recursos computacionais.

Concretizando, o código é desenvolvido em linguagem C++, garantindo a continuidade do trabalho existente, nomeadamente do simulador, e são utilizadas apenas utilizadas as bibliotecas standard de C++, com exceção da importação de ficheiros associados à configuração – ficheiros XML – caso em que é usada a "libxml++", C++ *wapper* para a "libxml2".

A validação de funcionamento do código desenvolvido é realizada pela utilização da biblioteca em código de simulação existente e análise de resultados por comparação com especificação técnica contida/definida neste documento.

Palavras-Chave

AUV, AGV, sensores, *cluster*, escalonamento, C++, biblioteca, recolha de dados, simulação

Abstract

The abstract should summarize the report’s contents, in what concerns the problem identification and/or the formulated hypotheses. The solution, its validation and assessment should also be briefly focused. In all, it should be less than 2 pages in length.

Keywords

We would like to encourage you to list your keywords, key phrases and most relevant acronyms in this section.

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

AGV – Automated Guided Vehicle

AUV – Autonomous Unamed Vehicle

XML – eXtensible Markup Language

UNIX™ – UNiplexed Information and Computing System

STL – Standard Template Library

GDB – GNU Project Debugger

GCC – GNU Compiler Collection

[™] UNIX is a registered trademark of The Open Group

1. INTRODUÇÃO

Neste capítulo é apresentada a orientação, a contextualização e o guia do projeto e do relatório desenvolvido no âmbito da disciplina de Seminário/Estágio, do 3º ano, da Licenciatura em Engenharia Electrotécnica e de Computadores.

1.1. GENERALIDADES

A acelerada evolução tecnológica e constantes mudanças nos mercados económicos, obrigam ao desenvolvimento de protótipos de simulação computacional para recriar cenários reais, sistemas reais. Com os dados resultantes das simulações são desenvolvidos um conjunto de análises e consequente validação do modelo a seguir no desenvolvimento de um produto ou serviço.

De acordo com histórico conhecido da natureza, algo nasce e desenvolve-se, evolui. No campo da simulação o caminho é o mesmo. Na base da simulação complexa está um modelo simples com capacidade de evolução. A evolução aqui referida define-se como a capacidade de um conjunto de código atingir elevado grau de eficiência e maturidade, incluindo as funcionalidades associadas ao mesmo.

Porquê um modelo simples? Na realidade a validação das aplicações é sempre o objetivo mais complexo. Assim, para permitir para facilitar a validação de resultados regra geral é criado um modelo simples tendo como ponto de partida uma boa base de evolução.

1.2. CONTEXTUALIZAÇÃO

Este projeto designado de Simulador de comunicações entre redes de sensores e veículos autónomos aparece inserido numa linha de investigação e de desenvolvimento com o principal objetivo de simular diferentes modelos de partilha do meu de comunicação por diferentes objetos – concretizando no âmbito deste projeto, designados de sensores.

Assim, decide-se criar uma biblioteca de sensores e respetivos métodos de interação com a mesma, permitindo a utilização da referida biblioteca em modelos de simulação já desenvolvidos e testados no âmbito de outros projetos.

Partindo da base do código já existente e tendo em conta que todo este código faz uso da linguagem de programação C++, torna evidente e obrigatória a utilização do mesmo tipo de linguagem no desenvolvimento deste projeto, sendo mesmo um dos requisitos do projeto.

Sem se conhecer integralmente os modelos de simulação existentes e com os quais a biblioteca deverá interagir, todo o desenvolvimento leva em linha de conta a necessidade de constante adaptação para interação com código desenvolvido por terceiros.

Mas o que levar em consideração? O que é necessário para...? O desconhecimento dos simuladores é um facto que leva a uma profunda análise na definição do modelo a seguir no desenvolvimento. Questões relacionadas com a performance, tornam-se severas se considerarmos que não se pretende comprometer futuras aplicações, bem como a expansão do sistema desenvolvido no presente.

1.3. OBJETIVOS

De acordo com os pontos anteriores defini-se como principal objetivo a criação de uma biblioteca de sensores capaz de gerir a informação contida em cada sensor, bem como a gestão devida do largura de banda disponível no canal de comunicação e respetivos elementos de ligação com modelos de simulação.

Partindo do zero, e para viabilizar a realização deste projeto é necessária a estruturação e consequente divisão do objetivo principal num conjunto de tarefas e respetivos objetivos para viabilizar a realização. Desta forma, define-se um conjunto de pontos a ter em conta:

- Estudar a linguagem de programação C++, incluindo conceitos, modos de programação e de otimização; Objetivo: familiarização com a linguagem de programação e aquisição de conhecimentos suficientes ao desenvolvimento necessário;
- Desenvolver classes de objetos que constituem um objeto sensor; Objetivo: aplicar conhecimentos adquiridos em desenvolvimento concreto, habilitando a biblioteca de um elevado nível de simplicidade de diagnostico e compreensão, incluindo a possibilitar a reutilização de código;
- Desenvolver o objeto sensor, sendo uma classe de classes; Objetivo: introdução ao conceito de modelização, levando em linha de conta o objetivo definido no ponto anterior;
- Desenvolver o objeto *cluster* (conjunto de sensores); Objetivo: aplicação da biblioteca standard de contentores, garantindo os objetivos definidos nos pontos anteriores;
- Estudar e implementar um método de importação das configurações dos sensores; Objetivo: desenvolver os conhecimentos em XML e criar uma biblioteca dinâmica no que se refere a parâmetros de entrada;
- Analisar comportamentos e resultados obtidos recorrendo ao uso da biblioteca com simuladores de terceiros; Objetivo: validar o correto funcionamento da biblioteca.

Com a concretização das diferentes tarefas acima enumeradas é garantido o sucesso deste projeto e consequente realização do objetivo principal, bem como todos os princípios a ter em consideração num conceito dito de desenvolvimento estruturado e organizado, nomeadamente: modelização, abstração, simplicidade e performance.

1.4. CALENDARIZAÇÃO

Todo o trabalho desenvolvido no decorrer deste projeto foi realizado durante o 2º semestre do ano letivo de 2011/12.

Em termos gerais, consideraram-se 4 (quatro) grandes macros:

- Lançamento do projeto e definições técnicas;
- Desenvolvimento de código e reuniões conjuntas de controlo;
- Validação do funcionamento da biblioteca;
- Desenvolvimento do relatório.

Em termos gerais, foi dedicado cerca de um mês a cada ponto, sendo que a validação da biblioteca foi realizada em cerca de dois meses.

1.5. ORGANIZAÇÃO DO RELATÓRIO

Este relatório é estruturado em 5 (cinco) capítulos: Introdução, Programação em C++, Desenvolvimento, Utilização da aplicação e Conclusões.

O presente capitulo - Introdução - é uma breve introdução que orienta e enquadra o leitor neste projeto, apresentando uma completa identificação e estruturação sobre o que vai ler e sobre o trabalho desenvolvido, incluindo a identificação de elementos chave considerados e aplicados na concretização.

O próximo capitulo, designado de Programação em C++, desenvolve-se em torno da linguagem de programação C++. Neste capítulo são apresentados os principais conceitos e definições essenciais adquirir para a completa compreensão das decisões tomadas no desenvolvimento.

No 3º capítulo, intitulado de Desenvolvimento, é narrado o código desenvolvido, muitas vezes em modo de manual, mas sempre com a perspetiva de detalhar as funcionalidades e capacidades da biblioteca.

O 4º capítulo, Utilização da aplicação, é uma breve descrição do modo como utilizar a biblioteca como aplicação. Embora não seja objetivo, o código desenvolvido, permite a utilização na forma de aplicação.

Por último, temos o capítulo Conclusões, onde normalmente se escreve: os objetivos foram alcançados com sucesso. Pois embora de uma forma simplista é exatamente isso que lá está escrito, com a ressalva de um maior detalhe e um conjunto de informações mais concretas.

2. PROGRAMAÇÃO EM C++

Neste capítulo são introduzidos os conceitos considerados essenciais para a compreensão do código desenvolvido e necessários adquirir o desenvolvimento do mesmo, incluindo métodos de desenvolvimento e mecanismos de suporte ao programador.

2.1. LINGUAGEM DE PROGRAMAÇÃO

No controlo e comando de equipamentos, nomeadamente no controlo de um computador (no âmbito deste projeto), são necessários mecanismos de interação, em modelo ação/reação. Assim, para interagir com as maquinas é usada uma linguagem específica, designada de linguagem de programação.

Uma aplicação informática, muitas vezes identificada no dia à dia como programa, é um conjunto de texto, tal como este documento, que faz uso de um dialeto, definido como linguagem de programação, que, após compilação[1], habilita um sistema/maquina a reagir em diferentes cenários e de acordo com o definido. A linguagem de programação C++ é um dos possíveis dialetos disponíveis para utilizar.

[1] Designa-se por compilação ao processo que traduz a linguagem de programação, contida num ou mais ficheiros – ficheiros de código – em código máquina.

2.2. A LINGUAGEM DE PROGRAMAÇÃO C++

Como descrito no ponto anterior a linguagem de programação C++ é apenas um dos muitos dialetos disponíveis para programação. Embora considerada por muitos como desvantagem o facto da linguagem ser complexa, se devidamente enquadrada a linguagem C++ é extremamente poderosa.

A linguagem de programação C++ deriva da linguagem C, linguagem originalmente desenvolvida para programação em ambiente UNIX™. O C é considerado uma linguagem de baixo nível e bastante poderosa. Quando o C é comparado (inevitável) com outras linguagens são identificadas algumas lacunas no que diz respeito a conceitos de programação mais recentes. Assim, o C++, pode ser apresentado como: uma linguagem de programação atual e com as potencialidades do C, mas inclui uma série de novas e modernas funcionalidades, gerando a capacidade de produzir aplicações complexas de forma simples.

2.3. ESTRUTURA DE UM PROGRAMA

Em C++, o programa mais simples de realizar, embora sem qualquer utilidade prática é o apresentado abaixo:

```
int main ()
{
}
```

A 1ª linha de código

```
int main ()
```

diz ao compilador que existe uma função designada de main e que a mesma devolve/retorna um inteiro. Uma função é um conjunto de código que executa uma determinada tarefa, por exemplo, uma operação matemática. Uma função pode devolver um parâmetro e aceita parâmetros de entrada.

Em C++ uma função é definida da seguinte forma

```
tipo nome (parâmetro_1, parâmetro_2, …)
{
  corpo da função
}
```

™UNIX is a registered trademark of The Open Group

onde:

- tipo é o tipo de dados devolvido pela função;
- nome é a identificação pela qual é realizada a chamada da função;
- parâmetros (os necessários) são os dados de entrada: cada parâmetro é constituído por um tipo de dados seguido de um identificador;
- corpo da função é um conjunto de expressões/atribuições limitadas por chavetas {}, que representam, respetivamente, o inicio e fim do corpo da função.

A função main é o ponto de partida de todas as aplicações desenvolvidas em C++. Independentemente da localização da mesma é a sempre a 1ª função a ser executada. Por esta mesma razão, é essencial que todos os programas em C++ contenham a função main.

Mas para que o código apresentado realize algo de visível para o utilizador e seja passível de complicação tem de ser alterado de acordo com o seguinte:

```
#include <iostream>

using namespace std;

int main ()
{
  cout << “Hello World!\nThanks for the work...”
<< endl;

  return 0;
}
```

Agora a 1ª linha de código

```
#include <iostream>
```

inicia-se por cardinal (#). Todas as linhas iniciadas pelo símbolo # são diretivas do preprocessador. Os detalhes associados ao preprocessador não são apresentados neste relatório. Interessa reter que esta linha de código, diretiva, solicita ao preprocessador a introdução da biblioteca contida entre os símbolos <>, neste caso a biblioteca *iostream* que é a biblioteca padrão que gere os fluxos dados entrada e saída em C++.

Todos os elementos da biblioteca padrão do C++ estão agrupados numa declaração identificada por *std*. Para a utilização dos elementos da biblioteca é obrigatório a utilização de respetiva identificação, por exemplo:

```
std::cout << “Hello World!\nThanks for the
work...” << std::endl;
```

Para simplificar pode ser usada a linha de código:

```
using namespace std;
```

evitando a utilização repetida da identificação *std*.

De notar que esta linha de código é a 1ª a ser terminada com ponto e virgula. O ponto e virgula faz parte da sintaxe do C++. Este símbolo diz ao compilador que é o fim de uma expressão.

As restantes linhas código de carácter desconhecido

```
cout << “Hello World!\nThanks for the work...” <<
endl;
return 0;
```

no caso da expressão

```
cout << “Hello World!\nThanks for the work...” <<
endl;
```

faz uso das funcionalidades disponibilizadas pela biblioteca padrão iostream para imprimir a sequencia de caracteres contidos entre as aspas (“”), levando em linha de conta o formato.

Já a expressão

```
return 0;
```

causa a conclusão da execução da função *main* e devolve o valor 0 (zero). Regra geral, funções que devolvem valores positivos representa que a mesma foi executada com sucesso.

Apenas será interessante referir que é possível introduzir no código linhas de comentários que podem ser bastante úteis para o programador. Os referidos comentários podem ser introduzidos de acordo com a seguinte sintaxe:

```
/* Comentário
   multi
     linha */

// Comentário de linha
```

2.4. VARIÁVEIS

Para a aplicação dialogar com o utilizador é necessário que a mesma aceite parâmetros de entrada, informação proveniente do exterior da aplicação. Para realizar este dialogo é necessário manipular e reter um conjunto de dados. Em programação estes dados são guardados em variáveis. Há diferentes tipos de variáveis para o efeito, definidas por tipo de acordo com a informação que contem.

Antes de ser possível a utilização de uma variável é necessário proceder declaração da mesma. A declaração de uma variável em C++ e realizada de acordo com a seguinte sintaxe:

```
tipo nome;
```

onde o:

- tipo é a classificação dos dados contidos na variável;
- nome é a identificação da variável.

No âmbito do projeto encontramos muitas vezes variáveis do tipo double em variáveis que poderiam ser tipo int. A utilização deste tipo double advém do facto da biblioteca ser capaz de interagir com elementos de simulação muitas vezes baseados em integração.

Para declara uma variável considerando que a mesma é sem sinal dever ser utilizado o identificador *unsigned*.

A manipulação e operações lógicas com variáveis são realizadas em C++ por uma série de operadores, sendo os mais comuns mencionados na tabela Tabela 2: Operadores em C++. Mais à frente iremos ver como os referidos operadores podem também ser utilizados em classes.

A tabela abaixo apresenta um resumo dos principais tipos de dados em C++.

Tabela 1: Tipos de dados em C++

Tipo	Descrição	Tamanho[2]	Gama[2]
char	Carácter ou número inteiro	1 byte	[-128 127] [0 255]
short int (short)	Numero inteiro	2 bytes	[-32768 32767] [0 65535]
int	Numero inteiro	4 bytes	[-2147483648 2147483647] [0 to 4294967295]
bool	Booleano	1 byte	Verdadeiro ou falso
float	Numero real com ponto flutuante de precisão simples	4 bytes	±3.4e±38 (~7 dígitos)
double	Numero real com ponto flutuante de precisão dupla	8 bytes	±1.7e±38 (~7 dígitos)

Tabela 2: Operadores em C++

Operador	Função
=	Atribuição
+=	Atribuição por adição
-=	Atribuição por subtração
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto)
++	Incremento
--	Decremento
<	Resultado lógico da operação menor que
<=	Resultado lógico da operação menor ou igual que
>	Resultado lógico da operação maior que
>=	Resultado lógico da operação maior ou igual que
==	Resultado lógico da operação igual a
!=	Resultado lógico da operação diferente de
&&	E lógico
||	Ou lógico
!	Negação lógica

[2] Os valores das colunas dependem do sistema no qual o programa é compilado. Os valores apresentados são os geralmente constantes em sistemas a 32 bits.

No campo das variáveis, nomeadamente no que diz respeito à criação das mesmas, há ainda dois aspetos a ter em consideração:

- O C++ é sensível à utilização de maiúsculas e minúsculas;
- Após a declaração das variáveis as mesmas não são inicializadas, sendo o conteúdo da mesma desconhecido;

Uma variável pode ser declarada como "estática" fazendo uso do identificador *static*. Esta funcionalidade está associada à alocação de memória, sendo uma variável "estática" alocada no momento em que é inicializada e apenas é "desalocada" no fim do programa.

2.5. ESTRUTURAS DE CONTROLO

Na execução de um programa não é limitada a uma sequência linear de instruções. Durante o processo, podem ocorrer decisões e repetições. Neste sentido em C++ existem as designadas estruturas de controlo para desempenhar as referidas tarefas.

No âmbito deste projeto são tipicamente utilizadas duas estruturas de controlo: uma condicional e uma de repetição, *if* e *for* respetivamente.

A estrutura condicional *if* apresenta a seguinte sintaxe e comportamento:

```
if (<expressão verdadeira>)
  Executa esta linha de código
```

ou

```
if (<expressão verdadeira>)
{
  Executa este bloco de código
}
```

No caso de se pretender uma cadeia condicional, *if else*, a sintaxe e comportamento são:

```
if (<expressão verdadeira ou falsa>)
  Executa esta linha de código se verdadeira

else
  Executa esta linha de código se falsa
```

ou

```
if (<expressão verdadeira ou falsa>)
{
  Executa este bloco de código se verdadeira
}

else
{
  Executa este bloco de código se falsa
}
```

A estrutura de repetição utilizada foi o ciclo *for* que apresenta a seguinte sintaxe e comportamento:

```
for (<inicialização da variável de controlo>;
     <condição de controlo>;
     <atualização de variável>)
{
  Executa este bloco de código enquanto a
    condição de controlo é verdadeira
}
```

2.6. FUNÇÕES

Terminado o ponto anterior mencionando repetições, aproveito a oportunidade para mencionar que uma das grandes aplicações das funções em C++ é exatamente a repetição de código. Quando se torna necessário a utilização de um conjunto de instruções, repetidamente ao longo do código, este conjunto de instruções deve ser encapsulado numa função, sendo a mesma chamada sempre que necessário. Desta conclusão, podemos ainda referir que a utilização de funções cria um código modular e estruturado, simplificando a interpretação do mesmo e evitando um conjunto de erros associados à repetição de código – uma função quando testada assume sempre o mesmo comportamento.

Introduzido o conceito de função no inicio deste capítulo, resta apenas referir que em função, tal como as variáveis, apenas pode ser utilizada após declaração da mesma. Este facto leva à introdução do conceito de programa dividido em vários ficheiros de código. Um programa pode ser construido por mais do que um ficheiro de código, sendo a ligação dos mesmos realizada posteriormente, existindo mesmo mecanismos proprios para facilitar a tarefa como o GNU Make.

Aproveito o conceito de função para introduzir um novo mecanismo: recursividade. Uma função pode invocar-se a sim mesma, a este mecanismo dá-se o nome de recursividade.

2.7. APONTADORES

O conceito de apontadores é algo que historicamente ganhou a atribuição de complexo. Mas, na realidade, é uma das ferramentas disponíveis em C++ que potencializa a linguagem.

Um apontador permite ao programador obter acesso à posição de memória onde a variável guarda os dados. Isto habilita o programador a gerir a memória possibilitando uma elevada performance da aplicação e o evitar do *overhead*[3] de memória.

Passando à utilização dos apontadores, estes são utilizados em semelhança com as variáveis da seguinte forma:

```
// Declaração de um apontador
tipo *nome
```

Para aceder ao endereço de memória de uma variável é usado o operador &.

Tal como nas variáveis, embora de uma forma mais severa, tendo em conta que acedemos diretamente a posições de memoria, é necessária um atenção cuidada em termos de inicialização e alocação de memória. No momento da declaração de um apontador é desconhecido o conteúdo da memória por parte do programador, bem como a quantidade de memória necessária por parte do sistema.

2.8. CLASSES

A decisão de utilização da linguagem de programação C++ deve-se muitas vezes facto da existência de classes. Uma classe é um conjunto de código, um conjunto dados e de funções que alteram ou disponibilizam dados. Em termos concretos, os dados designamos de membros de dados e as funções membros funcionais ou métodos.

Em termos práticos de programação a estrutura de declaração de uma classe é:

```
class nome_da_class {
private:
  // Membros privadas

public:
  // Membros públicos
};
```

[3] *Overhead* é um termo disponível no campo da ciência de computação para representar o uso excessivo de recursos.

sendo todos os membros identificados como *private* apenas acessíveis pelos métodos implementados, e os membros identificados como *public* totalmente acessíveis, fora do contexto da classe.

A declaração de uma classe é realizada de acordo com a seguinte sintaxe:

```
class my_class {
private:
  char *my_name;

public:
  void  set_name (const char &name);
  char *get_name (void) const;
};

// Declaração da classe
my_class me;
```

A utilização do identificador *const* em

```
char *get_name (void) const;
```

significa que o método não altera os membros da classe.

Servindo-me do exemplo anterior para alterar os dados – membros de dados – da classe, são utilizados os métodos:

```
// Atribuir o valor
me.set_name (“Chris”);

// Obter o valor
char *actual_name = me.get_name ();
```

Na realidade para que este bloco de código realize algo é necessário definir os métodos da classe. Fazendo uso da seguinte sintaxe:

```
nome_da_classe::método
```

podemos então definir os métodos:

```
// Método set_name ()
void my_class::set_name (const &name)
{
  this->my_name = name;
}

// Método get_name ()
char *my_class::get_name (void) const
{
  return my_name;
}
```

A utilização do apontador *this* é um apontador para a própria classe que pode ser usado quer com os membros de dados quer com os métodos.

Em C++ é permitido sobrepor métodos, como por exemplo:

```
class my_class {
private:
  char *m_first_name;
  char *m_last_name;

public:
  void  set_name (const char &last_name);
  void  set_name (const char &first_name,
                  const char &last_name);
  char *get_name (void) const;
};
```

Embora o código apresentado se considere correto e capaz de realizar tarefas, uma classe deve conter 3 (três) operações básicas:

- Inicialização da classe, construtor da classe
- Destruição da classe, destrutor da classe
- Copia da classe, construtor cópia

Em termos de sintaxe temos:

```
class my_class {
private:
  int   m_age;
  char *m_name;

public:
  void  set_age  (int age);
  void  set_name (const char &name);
  int   get_age  (void) const;
  char *get_name (void) const;
};

// Inicialização da classe
// Construtor da classe
my_class::my_class () { }

// Sobreposição do construtor
my_class::my_class (int age) :
  m_age (age)
{
}
```

```
// Construtor cópia
my_class::my_class (const my_class& other):
  m_age (other.m_age),
  m_name (other.m_name)
{
}

// Destrutor da classe
my_class::~my_class () { }
```

Tal como os métodos os construtores podem ser sobrepostos, na medida em que, uma classe pode conter mais do que um construtor, como podemos ver no exemplo acima. Em C++ existem uma série de construtores definidos por defeito. Estes devem ser utilizados com cuidado para evitar comportamentos inesperados do código.

2.8.1. CLASSES DERIVADAS EM C++

Um mecanismo disponível e bastante útil em C++ é a possibilidade de criar classes derivadas. Uma classe que deriva de outra designa-se de subclasse, sendo a classe que dá origem designada de superclasse. Assim, a subclasse pode herdar os membros da superclasse. Podemos considerar que, para efeitos de compreensão, a subclasse é uma particularização.

Em termos de sintaxe e concretizando a explicação em código, temos:

```
class Polygon {
private:
  char *m_name;

protected:
  double m_width;
  double m_height;

public:
  char *get_name (void) const
    { return m_name; }
  void set_serial (const char &name)
    { m_name = name; }
  void set_width (double width)
    { m_width = width; }
  void set_height (double height)
    { m_height = height; }
  virtual set (double width, double height)
    { this->set_width (width);
      this->set_height (height); }
};
```

```
class Rectangle : public Polygon {
public:
  double area ()
    { return (width * height); }
};

class Triangle : public Polygon {
public:
  double area ()
    { return (width * height / 2); };
};

class Square : public Polygon {
public:
  set (double side)
    {  m_width = side;
       m_height = side; }
  area () { return (side * 2); }
};
```

As relações entre classes são geralmente representadas graficamente. Para o exemplo acima, a representação é:

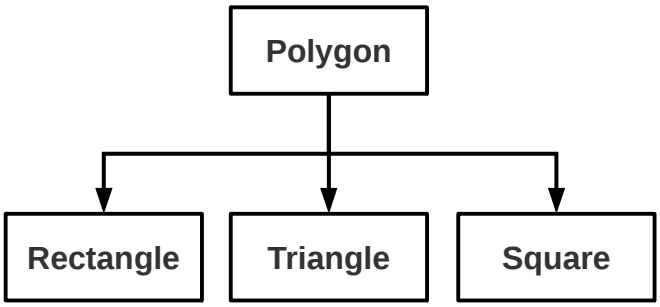

Figura 1: Classes derivadas

2.9. VETORES

O C++ disponibiliza um conjunto de estruturas de dados e respetivos operadores/funções. Todas estas estruturas estão disponíveis na STL. No âmbito deste projeto apenas vou detalhar a estrutura de vetores.

Por estrutura de vetores, entenda-se um contentor sequencial de objetos de determinado tipo e com acesso direto. A grande vantagem da utilização de vetores da STL é a alocação de memória automática, em oposição às cadeia conhecidas do C.

Tal como as classes tem de ser inicializado:

```
std::vector<tipo_de_dados> nome_do_vetor

// Inicialização de um vetor de inteiros
std::vector<int> serial_number
```

e permite manipulação:

```
// Adicionar elementos
serial_number.push_back (10);
serial_number.push_back (20);

// Numero de elementos
size_type nb_elements = serial_number.size ();

// Copia vetor
std::vector<int> serial_number_A
serial_number_A = serial_number
```

Como já foi mencionado, o diferentes elementos dos vetores são de acesso direto, isto é, é possível aceder a cada elemento do vetor através do índice (não aconselhado) ou de iteradores. Em termos de sintaxe:

```
// Inicialização de um vetor de inteiros
std::vector<int> serial_number

// Adicionar elementos
serial_number.push_back (10);
serial_number.push_back (20);
serial_number.push_back (30);
serial_number.push_back (40);

// Acesso direto por índice
int nb_1 = serial_number[0]
int nb_2 = serial_number[1]

// Acesso por iterador
std::vector<int>::iter;

int sum = 0;

iter = serial_number.begin ();
for (iter; iter != serial_number.end (); ++i)
  sum += *iter;
```

2.10. APLICAÇÕES DE SUPORTE AO DESENVOLVIMENTO

Todo este projeto foi desenvolvido em ambiente GNU/Linux, nomeadamente Arch Linux™, tendo sido utilizadas as seguintes ferramentas de suporte :

- Vim: editor de texto, desenvolvido para maximizar a eficiência da escrita;
- GCC: ferramentas de complicação do projeto GNU;
- GNU Make: aplicação que permite a criação de executáveis a partir de um conjunto de ficheiros de código fonte;
- GDB: ferramenta de monitorização e diagnostico do projeto GNU;
- Git: sistema de controlo de versões.

™The Arch Linux trademark policy is published under the CC-BY-SA license, courtesy of the Ubuntu project

3. DESENVOLVIMENTO

Neste capitulo são apresentado as principais funções desenvolvidas e toda a estrutura do código. Após um conjunto de considerações introdutórias, para simplicidade de enquadramento e apresentação, capitulo é desenvolvido em forma de manual.

3.1. DEPENDÊNCIAS

O código desenvolvido pode ser dividido em duas partes: biblioteca de sensores e interface com utilizador.

Assim, no âmbito da biblioteca de sensores, o código apresenta apenas uma dependência: libxml++. A libxml++ é uma biblioteca que disponibiliza um conjunto de ferramentas para trabalhar com xml. No contexto deste projeto é utilizada para permitir a importação de configuração de sensores.

No âmbito da interface com utilizador, para permitir a visualização das simulações e definição dos parâmetros de simulação é utilizado Gnuplot e Argp. O Gnuplot é uma aplicação de desenho gráfico utilizada para a visualização dos diferentes parâmetros manipulados pela aplicação desenvolvida e o Argp é uma ferramenta da biblioteca GNU C utilizada para o tratamento dos argumentos de entrada da aplicação.

3.2. ESTRUTURA

Em termos de estrutura de ficheiros a raiz da aplicação é intitulada de comlibsim. Na raiz estão todos os ficheiros código constituintes da biblioteca de sensores. Assim, a estrutura de ficheiros é a seguinte:

```
config/
gnuplot/
log/
sim/
README
accumulator.cpp
accumulator.hpp
cluster.cpp
cluster.hpp
comlibsim.cpp
comlibsim.hpp
comrate.cpp
comrate.hpp
makefile
object.hpp
parser.cpp
parser.hpp
position.cpp
position.hpp
sensor.cpp
sensor.hpp

config:
cluster_default_01.xml
cluster_default_02.xml
cluster_default_03.xml
cluster_default_04.xml
cluster_default_05.xml
cluster_default_06.xml

gnuplot:
map.p

log:
accumulator.log
rate.log
sensors.map
simulation.log

sim:
ode_solvers.cpp
ode_solvers.hpp
sim_main.cpp
sim_model.cpp
```

Os ficheiros da raiz são o desenvolvimento de um conjunto de classes que serão detalhadas posteriormente neste documento.

O diretório config agrupa um conjunto de ficheiros xml, ficheiros de configuração de sensores para utilizar nas simulações. O diretório gnuplot contem um script para criar um conjunto de representações gráficas para apresentação de resultados. O diretório log, agrupa os ficheiros de dados resultantes da simulação. O diretório sim, contem o código da ferramenta de simulação não objeto de detalhe neste documento uma vez que foi desenvolvida por terceiros.

Voltando ao ficheiros da raiz o ficheiro comlibsim.{cpp,hpp}[4] serve apenas de interface para o utilizador, incluindo a apresentação dos parâmetros carregados por defeito.

```
$ ./comlibsim --help
Usage: comlibsim [OPTION...] <filename>.xml
Communication Library Simulator

  -a, --accumulator=FILE     Accumulator log file
[log/accumulator.log]
  -e, --sensors=FILE         Sensors map file
[log/sensors.map]
  -l, --simulation=FILE      Simulation log file
[log/simulation.log]
  -p, --app=FILE             App to print sim
[gnuplot]
  -r, --rate=FILE            Rate log file
[log/rate.log]
  -s, --step=NUMBER          Simulation step size
[0.1]
  -t, --time=NUMBER          Simulation time
[500]
  -v, --verbose              Verbose output
  -?, --help                 Give this help list
      --usage                Give a short usage
message
  -V, --version              Print program
version

Mandatory or optional arguments to long options
are also mandatory or optional for any
corresponding short options.

Report bugs to <da.arada@gmail.com>.
```

[4] A nomenclatura comlibsim.{cpp,hpp} define dois ficheiros: comlibsim.cpp e comlibsim.hpp

O ficheiro README é um ficheiro de introdução à biblioteca incluindo a descrição da utilização, compilação e funcionalidades.

Resta apenas referir a ligação entre classes. A classe Accumulator, ComRate, Position e Sensor derivam da classe Object, sendo representado em termos gráficos da seguinte forma:

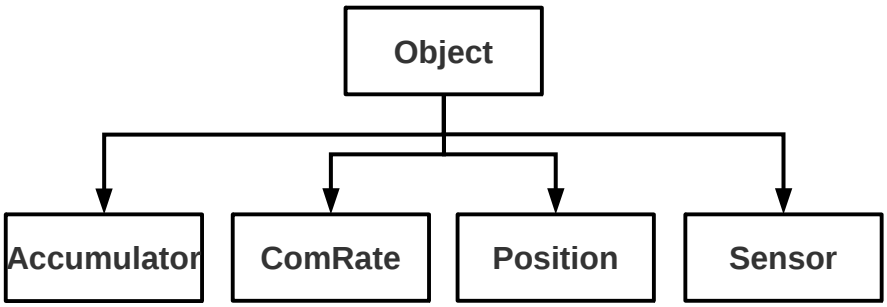

Figura 2: Ligação entre classes

3.3. ESPAÇO DE NOMES (NAMESPACE)

De acordo com o capítulo anterior, para o desenvolvimento da biblioteca, todos foi utilizado o seguinte espaço de nomes (namespace): ComLibSim. Assim, sempre que se pretende fazer uso das classes e membros das mesmas é obrigatória a utilização do espaço de nomes.

Tendo em conta que esta biblioteca será utilizada com simuladores desenvolvidos por terceiros, dos quais é desconhecido o código, o recurso à funcionalidade “espaço de nomes” faz todo o sentido uma vez que evita a sobreposição de código e respetivos comportamentos não esperados das ferramentas.

3.4. CLASSE POSITION

A classe Position define um ponto num plano xy. É utilizada na classe Sensor para definir a posição dos sensores num determinado plano xy, bem como para determinar a distancia de um ponto a outro, por exemplo determinar a distância de um sensor a outro.

Os elementos que constituem a classe são:

- double m_x – coordenanda x no plano xy;
- double m_y – coordenada y no plano xy.

Os construtores da classe são:

```
Position ();
Position (double x, double y);
Position (const Position& position);
```

sendo os parâmetros de entrada:

- double x – coordenada x no plano xy;
- double y – coordenada y no plano xy.

Quando omitidos os dois parâmetros é considerado o ponto (0,0).

Os membros de acesso aos elementos são:

```
virtual double get_x () const;
virtual double get_y () const;
virtual void get_xy (double *xy) const;
virtual double distance_to (const Position&
position) const;
```

sendo:

- double get_x – devolve a coordenada x;
- double get_y – devolve a coordenada y;
- void get_xy – escreve em xy[0] a coordenada x e em xy[1] a coordenada y;
- double distance_to – devolve a distancia entre o ponto da classe e o ponto de definido no parâmetro de entrada (position).

Os membros de registo dos elementos são:

```
virtual void write     (std::ostream& output =
std::cout) const;
virtual void write_ln  (std::ostream& output =
std::cout) const;
virtual void write_map (std::ostream& output =
std::cout) const;
friend std::ostream& operator << (std::ostream&
output, const Position& position);
```

sendo o parâmetro de entrada um ficheiro ou, por omissão o ficheiro standard de saída do sistema (consola).

Este conjunto de funções é utilizado para registar em ficheiro um conjunto de dados decorrentes da simulação. São apresentadas diferentes funções com a mesma funcionalidade variando apenas a forma como os dados são guardados. Quando se refere forma como os dados são guardados, entenda-se como a estrutura base que define o conteúdo do ficheiro.

Exemplo de utilização:

```
#include “position.h”

using namespace ComLibSim;

int main ()
{
  Position p0 (0.0, 10.0);
  Position p1 (10.0, 0.0);
  Position p2 (p1);

  std::cout << “Point 0: (“ << p0.get_x () <<
            “, “ << p0.get_y () << “)” <<
            std::endl;

  std::cout << “Point 1: (“ << p1.get_x () <<
            “, “ << p1.get_y () << “)” <<
            std::endl;

  std::cout << “Distance from p1 to p2: “ <<
            p1.distance_to (p2) << std::endl;

  return 0;
}
```

Saída:

```
Point 0: (0.0, 10.0)
Point 1: (10.0, 0.0)
Distance from p1 to p2: 0.0
```

3.5. CLASSE ACCUMULATOR

A classe Accumulator define um acumulador de dados. É utilizada na classe Sensor para definir a memória dos sensores em todos os instantes. Esta classe foi desenvolvida para utilização futura uma vez que disponibiliza um conjunto de funções associadas à comparação do volume de informação. De uma forma geral, pretende-se, como desenvolvimento futuro, gerir prioridades de acordo com o volume de informação contido em cada sensor. Por exemplo, atribuir uma periodicidade superior a sensores com maior volume de informação definindo a trajetória do veiculo de acordo com essas prioridades.

O elemento que constitui a classe é:

- double m_amount_data – quantidade de informação disponível no acumulador .

Esta variável é definida como double, tendo em conta que a mesma é manipulada por mecanismos de simulação, muitas vezes assentes em métodos de integração.

Os construtores da classe são:

```
Accumulator ();
Accumulator (double amount_data);
Accumulator (const Accumulator& accumulator);
```

sendo o parâmetro de entrada:

- double amount_data – quantidade de informação guardada no acumulador.

Quando o parâmetro é omitido considera-se zero.

Os membros de acesso aos elementos são:

```
virtual double get_amount_data () const;
virtual void set_amount_data (double
amount_data);
virtual bool is_empty () const;
```

sendo:

- double get_amount_data – devolve a quantidade de informação disponível;
- void set_amount_data – altera a quantidade de informação do acumulador;

- bool is_empty – testa se o acumulador não tem informação disponível.

Os membros de registo são:

```
virtual void write     (std::ostream& output =
std::cout) const;
virtual void write_ln  (std::ostream& output =
std::cout) const;
virtual void write_log (std::ostream& output =
std::cout) const;
friend std::ostream& operator << (std::ostream&
output, const Accumulator& accumulator);
```

Como já referido atrás, este conjunto de funções é utilizado para registar em ficheiro um conjunto de dados decorrentes da simulação.

Exemplo de utilização:

```
#include “accumulator.h”

using namespace ComLibSim;

int main ()
{
  Accumulator a0 (10.0);
  Accumulator a1;

  std::cout << “Accumulator 0: “ <<
            a0.get_amount_data () <<
            std::endl;

  a0.set_amount_data (20.0);

  std::cout << “Accumulator 0: “ <<
            a0.get_amount_data () <<
            std::endl;

  if (a1.is_empty ())
    std::cout << “Accumulator 1 is empty!” <<
              std:endl;

  return 0;
}
```

Saída:

```
Accumulator 0: 10.0
Accumulator 0: 20.0
Accumulator 1 is empty!
```

3.6. CLASSE COMRATE

Considerando um elemento comunicante, sabes-se que a taxa de transmissão varia em função da distância entre emissor recetor. Assim, a classe ComRate define o mapa de comunicação e respetiva taxa de transmissão. É utilizada na classe Sensor para determinar a taxa de transmissão nos diferentes pontos da simulação. Esta classe foi desenvolvida tendo em conta futuros implementações. Assim, caso se pretenda definir um qualquer outro mapa de comunicação, pode ser criada uma classe derivada.

A classe define um mapa de transmissão constituído por 3 (três) zonas circulares, centradas numa posição, sendo:

- Zona de não transmissão – zona no plano xy onde a taxa de transmissão é nula;
- Zona de transmissão variável – zona no plano xy onde a taxa de transmissão varia de forma exponencial, reduzindo de valor à medida que nos afastamos da posição central;
- Zona de transmissão máxima – zona no plano xy onde a taxa de transmissão é máxima.

Os elementos que constituem a classe são:

- const Position* m_reference – um apontador que define a posição central do mapa, corresponde à posição do sensor;
- double m_max_rate – taxa máxima de transmissão;
- double m_act_rate – taxa de transmissão activa;
- static const double m_radius_low – limite que define a zona de transmissão máxima e de valor 20,0;
- static const double m_radius_high – limite que define a zona de transmissão variável e de valor 60,0.

De acordo com a descrito, temos:

- para uma distância superior a 60,0, referenciada ao ponto central m_reference, considera-se a zona de não transmissão;
- para uma distância entre 20,0 e 60,0, referenciada ao ponto central m_reference, considera-se a zona de transmissão variável;
- para uma distância inferior a 20,0, referenciada ao ponto central m_reference, considera-se a zona de transmissão máxima.

Os construtores da classe são:

```
ComRate ();
ComRate (const Position& reference,
         double max_rate,
         double act_rate = 0);
ComRate (const ComRate& com_rate);
```

sendo os parâmetros de entrada:

- const Position& reference – classe Position que define o centro do mapa de transmissão;
- double max_rate – taxa máxima de transmissão para o mapa de comunicação;
- double act_rate – taxa de transmissão ativa; quando omitida é considerada 0 (zero).

Os membros de acesso aos elementos são:

```
virtual void set_act_rate (double act_rate);
virtual double get_max_rate () const;
virtual double get_act_rate () const;
virtual double rate_at (double distance) const;
virtual double rate_at (const Position& position)
const;
```

sendo:

- void set_act_rate – altera a taxa de transmissão ativa;
- double get_act_rate – devolve a taxa de transmissão ativa;
- double rate_at – devolve a taxa de transmissão a determinada posição ou distancia.

À semelhança das classes anteriores, nesta são também disponibilizados um conjunto de funções para registar em ficheiro um conjunto de dados decorrentes da simulação.

Exemplo de utilização:

```
#include “position.h”
#include “comrate.h”

using namespace ComLibSim;

int main ()
{
  Postion p0;
  ComRate cr0(p0, 10.0);

  std::cout << “Rate at p0: “ <<
            cr0.rate_at (0.0) <<
            std::endl;

  cr0.set_act_rate (2.0);
```

```
  std::cout << “Actual rate: “ <<
            a0.get_act_rate () <<
            std::endl;

  return 0;
}
```

Saída:

```
Rate at p0: 10.0
Actual rate: 2.0
```

3.7. CLASSE SENSOR

A classe sensor agrupa as 3 (trẽs) classes atrás mencionadas, nomeadamente a classe Position, a classe Accumulator e a classe ComRate. Esta classe representa um sensor, disponibilizando as funções descritas em cada uma das classes que o constituem, adicionando apenas funções funções para registo de dados.

O facto da classe Sensor ser constituída de um conjunto de classes, tendo cada classe funções especificas, possibilita a alteração do comportamento e/ou funcionalidade de uma das classes, sem que seja necessário alterar todo o código. Este tipo de programação é designado de programação modular e caracteriza-se por uma independência de funcionalidades. Cada classe é responsável pelo tratamento dos dados sendo estes disponibilizados por métodos próprios e independentes.

Os elementos que constituem a classe são:

- Accumulator m_accumulator – classe Accumulator, representa o acumulador de dados do sensor;
- Position m_position – classe Position, representa a posição do sensor num plano xy;
- ComRate m_com_rate – classe ComRate, representa o mapa de transmissão do sensor;
- std::string m_tag – identificação do sensor, utilizada para efeitos de apresentação gráfica e registo de resultados da simulação;
- double m_rate – taxa de transmissão da última descarga de dados, utilizada para efeitos de apresentação gráfica e registo de resultados da simulação.

Os construtores da classe são:

```
Sensor (const Position& position,
        double max_rate,
        double data,
        const std::string& tag = NULL);
Sensor (const Sensor& sensor);
```

sendo os parâmetros de entrada:

- const Position& position – classe Position que define o posicionamento do sensor num plano xy;
- double max_rate – taxa máxima de transmissão para o mapa de comunicação;
- double data – volume de informação disponível no sensor;
- const std::string& tag – identificação do sensor; quando omitida considera-se NULL.

Os principais membros de acesso aos elementos são os descritos em cada uma das classes que constituem o sensor.

À semelhança das classes anteriores, nesta são também disponibilizados um conjunto de funções para registar em ficheiro um conjunto de dados decorrentes da simulação.

Exemplo de utilização:

```
#include “sensor.h”

using namespace ComLibSim;

int main ()
{
  Postion p0;
  Sensor  s0 (p0, 10.0, 100.0, “Sensor_0”);

  std::cout << s0.write_tag () <<
            “ rate at p0: “ <<
            s0.rate_at (0.0) <<
            std::endl;

  return 0;
}
```

Saída:

```
Sensor_0 rate at p0: 10.0
```

3.8. CLASSE CLUSTER

No âmbito desta projeto define-se como *cluster* um conjunto de sensores, localizados num plano xy.

A classe Cluster é, na sua essência, um conjunto de sensores. Dado que se trata de um conjunto de sensores, é esta classe que gere a forma como os mesmo interagem com outro elemento, nomeadamente um veículo. Considerando um veiculo que se passeia pelo plano xy onde estão localizados os sensores, este pode interagir com o cluster para recolher informação dos referidos sensores. Mas nem todos, os sensores, tem a mesma quantidade de informação, nem todos podem comunicar à mesma taxa, nem todos estão à mesma distância do veiculo... É neste sentido que a classe Cluster existe, para dar sentido a todas estas questões, respondendo de forma concreta e simples.

De acordo com o descrito identifica-se como principais funções desta classe as seguintes:

- Identificar qual o sensor mais perto de determinado ponto, com o objetivo de orientar o veiculo;
- Selecionar quais os sensores com condições para transmitir em cada instante;
- E atribuir a cada sensor a taxa de transmissão para negociação com o veiculo.

No que diz respeito a gerir a taxa de transmissão, visando utilização da máxima taxa disponibilizada pelo o canal de transmissão em cada instante, a classe identifica, para determinada posição do veiculo, quais os sensores com condições para transmitir, atribuindo a cada sensor a taxa de transmissão. Mas o somatório da taxa transmissão atribuída aos sensores pode ser superior à taxa de transmissão do veiculo, superior à largura de banda disponível. Neste caso, a classe Cluster distribui a largura de banda disponível equitativamente por todos os sensores atribuindo taxas relativas de acordo com a taxa máxima de cada sensor.

Concretizando, considerando um veículo com taxa de transmissão de 100, em determinado instante, e um *cluster* constituído por 3 (três) sensores, cada sensor com taxa de transmissão máxima de 100 e todos com informação para transmitir, dada a posição do veículo, verifica-se que apenas 2 (dois) sensores tem condições para transmitir à taxa

máxima. Tendo em conta que a taxa de transmissão disponível é de 100, disponível pelo veículo, é atribuída, pela classe Cluster, a taxa de 50 a cada sensor.

Mantendo as condições anteriores, 2 (dois) sensores com condições para transmitir, alterando apenas a taxa de transmissão máxima possível no referido instante, de cada um dos dois sensores, de 100 para 55 e 60, mais uma vez, a soma da taxa de transmissão dos sensores é superior à taxa disponível. Neste cenário a classe Cluster reduz a taxa de 55 para 48 e de 60 para 52, em números redondos, perfazendo a taxa disponível e maximizando a taxa de cada sensor.

Mais uma vez, considerando que o sensor tem informação (caso contrário, o sensor não é considerado) em termos matemáticos, temos:

$$\begin{aligned} t_i(d) &= t_{i_{max}} && , 0 < d < 20 \\ t_i(d) &= t_{i_{max}} \, e^{1-\frac{d}{20}} && , 20 \leq d < 60 \\ t_i(d) &= 0 && , d \geq 60 \end{aligned}$$

sendo:

- t_i a taxa de transmissão instantânea do sensor i ;
- $t_{i_{max}}$ a taxa máxima de transmissão do sensor i ;
- d a distância entre o emissor e recetor, sensor e veículo.

Resultante da atribuição da taxa de transmissão acima mencionada é calculada a largura de banda necessária, que resulta de:

$$LB_t = \sum_{i=1}^{n} t_i$$

sendo:

- LB_t a largura de banda necessária, sem escalonamento;
- n o número total de sensores com condições para transmitir.

De acordo com a largura disponível é aplicado o escalonamento ou não. Sempre que se verifique que a largura de banda disponível é inferior à largura de banda necessária é realizado novo calculo:

$$t'_i(d) = \frac{t_i(d) LB_d}{LB_t}$$

sendo:

- t'_i a taxa de transmissão instantâneo do sensor após escalonamento;
- LB_d a largura de banda disponível.

A esta gestão de taxas de transmissão, incluindo a decisão de transmitir ou não, é a referida política de escalonamento mencionada no decorrer deste documento.

A classe Cluster permite a utilização de simuladores discretos ou contínuos. Entenda-se por:

- Simuladores discretos, os simuladores em que o programador é capaz de gerir cada um dos diferentes passos da simulação;
- Simuladores contínuos, os simuladores em que após o lançamento de um rotina o programador não tem condições para determinar qual o estado de todos os passos da simulação.

Desta forma, a classe Cluster apresenta um conjunto de métodos com o sufixo "_int" destinados a simuladores contínuos. A grande diferença entre os simuladores discretos e contínuos advém da liberdade do simulador executar um conjunto de tarefas de experimentação para aumentar a eficiência do algoritmo.

Durante as referidas experiências é necessário realizar o calculo da taxa de transmissão de cada sensor e manipular os dados da classe Cluster. Como já mencionado, a classe disponibiliza as ferramentas para este tipo de simuladores, simuladores contínuos.

O principal elemento que constituem a classe é:

- std::vector<Sensor> m_sensors – vetor de classe Sensor, representa os sensores que constituem o *cluster*.

A classe Cluster disponibiliza um construtor que apenas inicializa a classe. Pode ser utilizado como parâmetro de entrada o número total de sensores que serão guardados pela mesma. No entanto, como veremos mais à frente definições dos elementos que constituem a classe poderão ser importados de um ficheiro xml, o que torna o processo bastante mais simples.

Os principais membros da classe são:

```
virtual void add (const Sensor& sensor);
virtual Sensor& closest
                  (const Position& position);

// Para simuladores discretos
virtual void set_data     (double *data,
                           double delta_time);
virtual ComMap map (const Position& position,
                    double agv_bandwidth);

// Para simuladores contínuos
virtual void init_int ();
virtual void get_rate_int (double *rate,
                     const double *position,
                           double agv_bandwidth);
```

sendo:

- void add – para adicionar sensores ao cluster;
- Sensor& closest – devolve o sensor, com dados, mais próximo de determinada posição (position) para orientação do veiculo;
- void set_data – carrega o volume de informação de cada sensor de acordo com o parâmetro de entrada (data);
- ComMap map – carrega os sensores com as taxas de transmissão, isto é, após selecionar os sensores do cluster que tem condições para transmitir em determinada posição, define para cada sensor a taxa de transmissão, incluindo escalonamento, se necessário;
- void init_int – inicializa a classe para simulação contínua;
- get_rate_int – devolve a taxa de transmissão (com escalonamento, se necessário) dos sensores em determinado instante, tendo em conta a largura de banda disponível (avg_bandwidth) e a posição do veículo (position).

À semelhança das classes anteriores, nesta são também disponibilizados um conjunto de funções para registar em ficheiro um conjunto de dados decorrentes da simulação.

Exemplo de aplicação:

```
#include "cluster.hpp"

void main ()
{
  Cluster cluster;

  Sensor sensor_00 (Position (0.0, 0.0),
                    50.0, 500.0);
  Sensor sensor_01 (Position (100.0, 0.0),
                    50.0, 500.0);
  Sensor sensor_02 (Position (0.0, 100.0),
                    50.0, 500.0);
  Sensor sensor_03 (Position (100.0, 100.0),
                    50.0, 500.0);
  Sensor sensor_04 (Position (10.0, 10.0),
                    50.0, 500.0);

  cluster.add (sensor_00);
  cluster.add (sensor_01);
  cluster.add (sensor_02);
  cluster.add (sensor_03);
  cluster.add (sensor_04);

  cluster.map (Position (x, y), AGV_BANDWIDTH);

  double s0_rate = sensor_00.rate ();
  double s1_rate = sensor_01.rate ();
  double s2_rate = sensor_02.rate ();
  double s3_rate = sensor_03.rate ();
  double s4_rate = sensor_04.rate ();
}
```

3.9. CLASSE PARSER

Torna-se bastante penoso a criação do cluster como podemos observar no ponto anterior. Desta forma, foi desenvolvida esta classe, designada de Parser e baseada em libxml++, para habilitar o utilizador a realizar a importação de ficheiros xml com a configuração do cluster e respetivos sensores.

À semelhança das classes anteriores, nesta são também disponibilizados um conjunto de funções para registar em ficheiro um conjunto de dados decorrentes da simulação.

Exemplo de aplicação:

```
// Exemplo de ficheiro xml
/*
<?xml version="1.0" encoding="UTF-8"?>
<!-- XML Example file of cluster configuration
-->
<Cluster description="Example Cluster">
  <Sensor tag="Sensor 01" x="0.0" y="0.0"
max_rate="50.0" data="100.0"/>
  <Sensor tag="Sensor 02" x="10.0" y="10.0"
max_rate="50.0" data="100.0"/>
  <Sensor tag="Sensor 03" x="20.0" y="20.0"
max_rate="50.0" data="200.0"/>
  <Sensor tag="Sensor 04" x="30.0" y="30.0"
max_rate="25.0" data="10.0"/>
</Cluster>
*/

#include "parser.hpp"

void main ()
{
  Cluster cluster;

  Parser parser ("anydir/anyname.xml");

  parser.to_cluster (cluster)
}
```

4. UTILIZAÇÃO DA APLICAÇÃO

Embora fora do âmbito do projeto, para testar e verificar o comportamento da biblioteca de sensores, foi desenvolvido um conjunto de código que habilita a biblioteca a assumir o comportamento de aplicação.

Neste capitulo, embora muito sucintamente, apresenta-se a forma como utilizar a ferramenta, nomeadamente compilar e simular.

4.1. COMPILAR A APLICAÇÃO

No conjunto dos ficheiros que fazem parte da biblioteca está integrado um ficheiro makefile para utilizar a ferramenta GNU Make.

Para além das ferramentas necessárias para o ambiente de desenvolvimento em C++, como já mencionado, existem algumas dependências a ter em conta e que instaladar antes de compilar o código, nomeadamente libxml++ (biblioteca de desenvolvimento) e gnuplot (aplicação).

De acordo com o sistema operativo as referidas dependências poderão ser instaladas recorrendo aos seguintes comandos:

- Ubuntu: sudo apt-get install libxml++2.6-dev gnuplot
- Fedora: su -c 'yum install libxml++-devel gnuplot'
- ArchLinux: su -c 'pacman -S libxml++ gnuplot'

A biblioteca libxml++ é utilizada para a classe Parser para importação de ficheiros xml. A aplicação gnuplot é utilizada para representação gráfica dos resultados.

Após a instalação das dependências resta apenas compilar o código desenvolvido. Para tal basta correr em linha de comandos:

```
$ make
```

4.2. EXECUTAR A APLICAÇÃO

Para executar a aplicação basta apenas correr em linha de comandos:

```
$ ./comlibsim
Usage: comlibsim [OPTION...] <filename>.xml
Try `comlibsim --help' or `comlibsim --usage' for
more information.
```

Como se pode verificar do *output* da aplicação, é necessário especificar o ficheiro xml com a configuração do cluster. De base, na estrutura de ficheiros, estão disponíveis um conjunto de ficheiro xml na pasta config para simulações.

Assim, executando a aplicação:

```
$ ./comlibsim -t 250 -s 0.05
config/cluster_default_02_v1.xml
Communication Library Simulator
ComLibSim v0.0
Imported 2 sensors from file
"config/cluster_default_02_v1.xml"
n_dim=5; horizon_MR=1
```

São apresentados 3 (três) gráficos semelhantes às figuras abaixo.

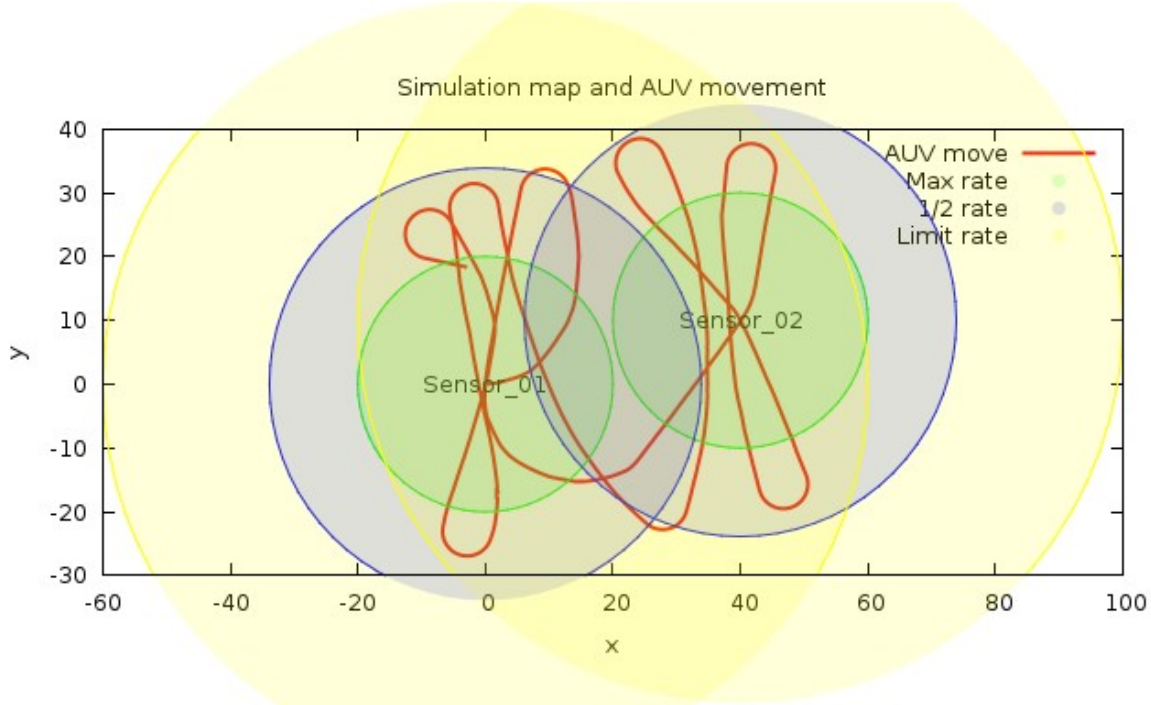

Figura 3: Mapa da simulação com dois sensores

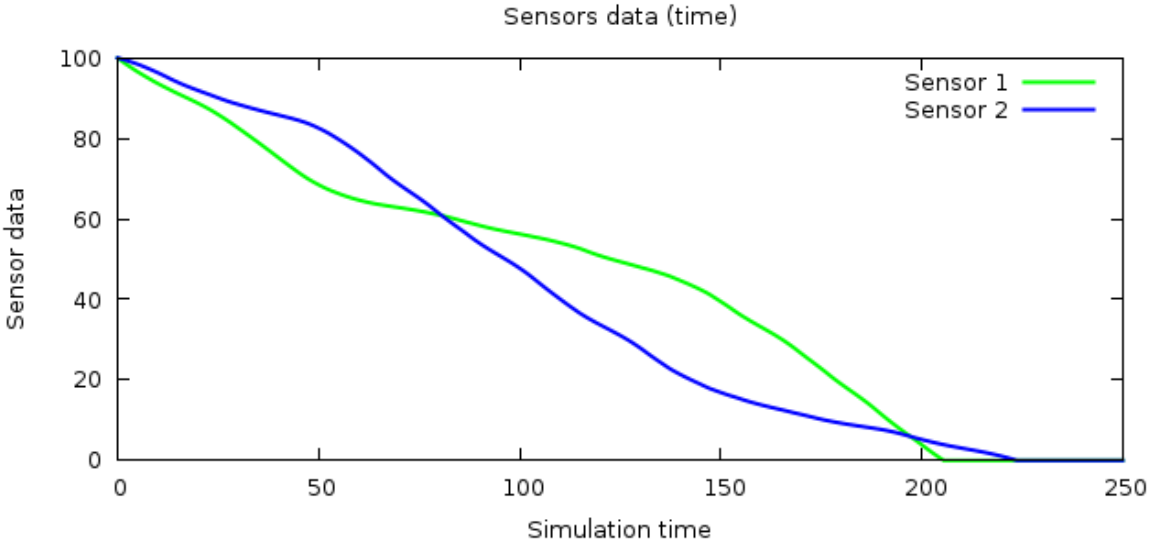

Figura 4: Quantidade de informação disponível, simulação com dois sensores

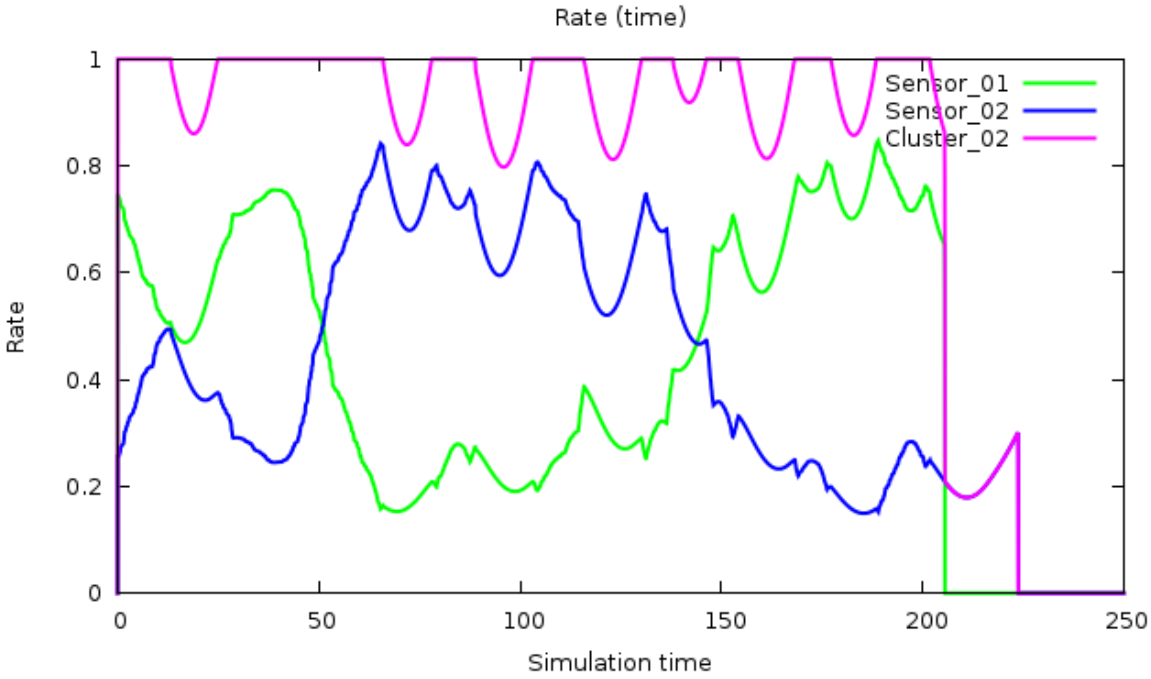

Figura 5: Taxas de transmissão, simulação com dois sensores

Embora seja detalhado no próximo capitulo, podemos verificar nos gráficos acima, que o cluster é constituído por 2 (dois) sensores, um posicionado em (0, 0) e outro em (40, 20). As taxas de transmissão máximas de todos os elementos, sensores e veículo, são de 1 (um). O veículo parte da posição (0, 0) e só para quando atinge os dois sensores já não tem dados, isto é, durante a aquisição de dados o veículo nunca para.

Ao longo do tempo podemos observar o decréscimo de informação dos sensores e a ocupação de largura de banda junto do limite máximo, durante grande parte do período da simulação, embora a taxa individual de cada sensor seja sempre inferior, maior parte do tempo inferior a 80% (valor retirado do gráfico acima) da taxa nominal – taxa máxima – causa da aplicação do escalonamento.

5. CONCLUSÕES

Ao longo dos capítulos foram apresentadas as principias ferramentas desenvolvidas bom como todas as decisões tomadas para atingir, com sucesso, os objetivos propostos.

Podemos facilmente verificar que os resultados apresentados no capitulo anterior traduzem a conclusão de que as decisões tomadas e as definições levadas em linha de conta no desenvolvimento foram corretas, nomeadamente pelo facto da taxa de ocupação da largura de banda e correto escalonamento. A ausência de monopólio, isto é, a ocupação de grande parte da largura de banda disponível pelos diferentes sensores é prova de que a política de escalonamento é eficaz. Verifica-se ainda que o decaimento do volume de dados dos sensores é semelhante em ambos.

Mais uma prova do descrito acima está no resultado obtido com uma simulação com 3 (três) sensores, novamente sem que o veículo pare. Executando a aplicação:

```
$ ./comlibsim -t 50 -s 0.0025
config/cluster_default_03_v1.xml
Communication Library Simulator
ComLibSim v0.0
Imported 3 sensors from file
"config/cluster_default_03_v1.xml"
n_dim=6; horizon_MR=1
```

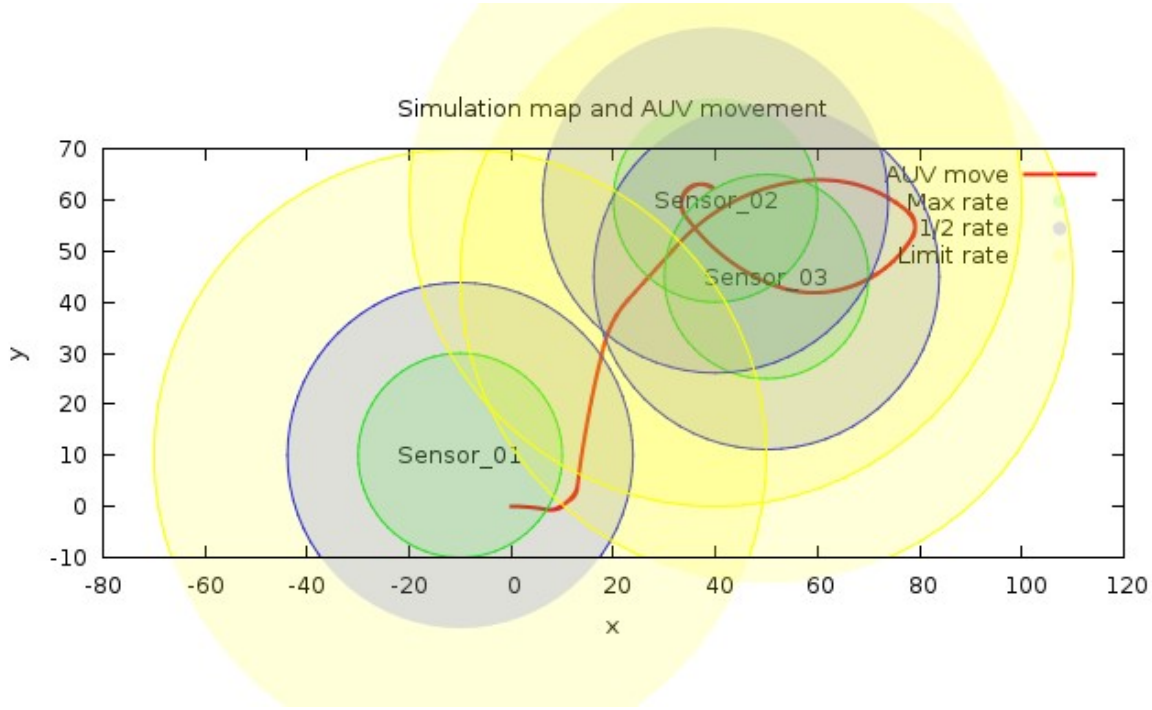

Figura 6: Mapa da simulação com três sensores

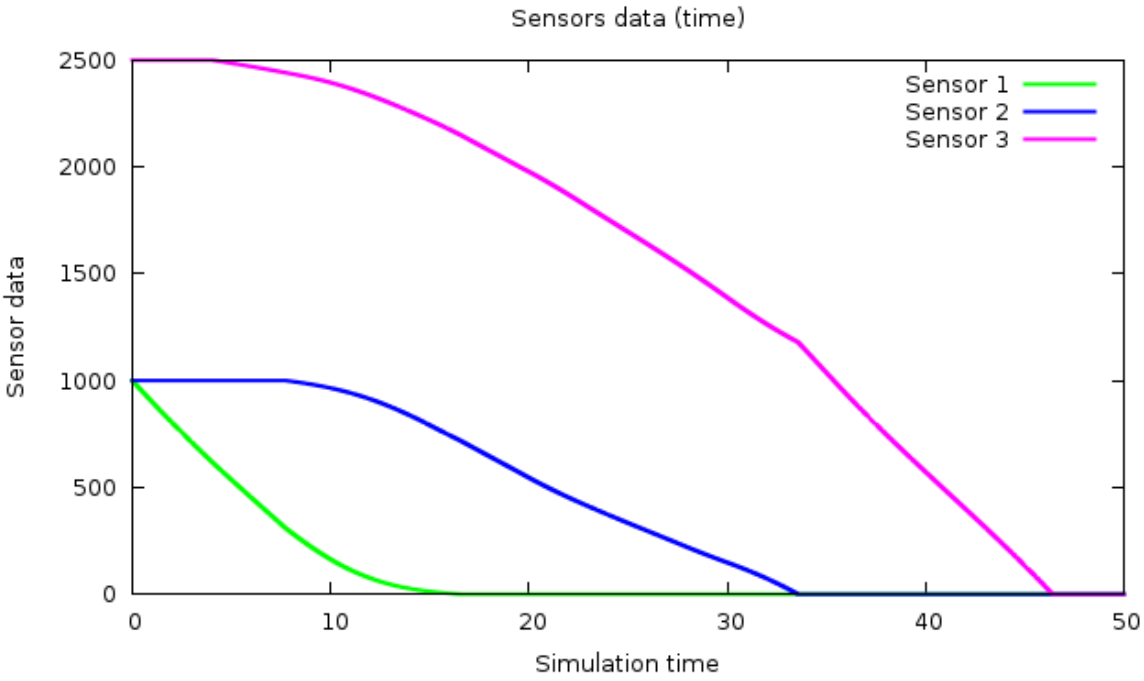

Figura 7: Quantidade de informação disponível, simulação com três sensores

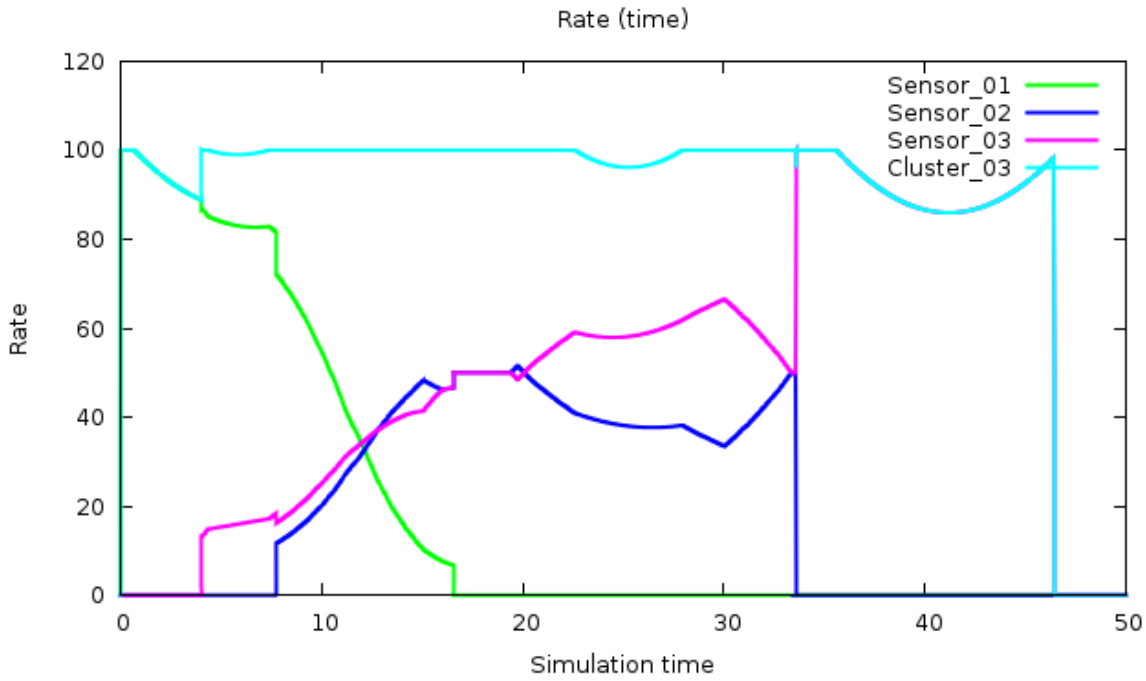

Figura 8: Taxas de transmissão, simulação com três sensores

Novamente, verifica-se que a ocupação da largura de banda disponível é praticamente máxima, durante o tempo da simulação.

Em ambas as simulações, volto a referir, é considerado que o veículo nunca para. Este facto traduz-se numa constante variação de taxas de transmissão e aumento de erros associados aos algoritmos.

Com base nos gráficos apresentados até agora depreende-se a dificuldade associada à verificação e validação de resultados. De todo o desenvolvimento executado salienta-se a dificuldade de verificar a validade dos dados resultantes da simulação. De uma forma geral, para verificar o correto funcionamento das classes desenvolvidas foram usados métodos discretos com chamadas aos diferentes membros das classes e análise de resultados. Após a verificação de todas as classes por métodos discretos, passou-se a usar a ferramenta de simulação. Como já descrito o simulador usado foi desenvolvido por terceiros. Assim, a verificação de dados foi realizada pela análise gráfica e análise de ficheiros com os resultados das simulações.

Resumindo, as simulações apresentadas até ao momento, são complexas e consequentemente os resultados são complexos tornando a realização de um texto de análise extenso. Passo a apresentar uma simulação com dois sensores e considerando que o veículo para sempre que a largura de banda disponível é totalmente ocupada.

Primeiro o mapa da simulação:

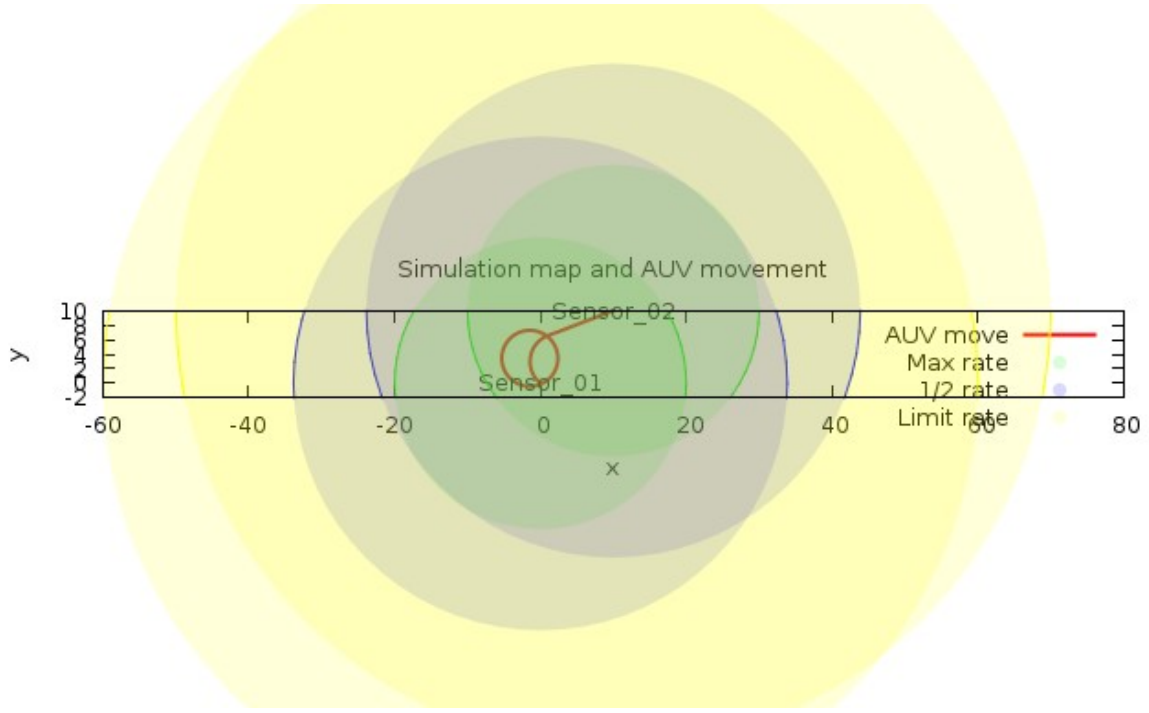

Figura 9: Mapa da simulação com dois sensores

O trajeto do veiculo é completamente alheio à biblioteca e fora do âmbito deste projeto. A biblioteca apenas disponibiliza um conjunto de informações que podem ser utilizadas pelo controlador do veículo para tomada de decisões.

Pode ser verificado que durante todo o percurso do veículo os sensores podem transmitir à taxa máxima, todo o percurso do veículo é realizado dentro da área verde, área que define a taxa que o sensor pode transmitir à taxa máxima.

Sabendo que a taxa máxima é de 1 para o “Sensor_01” e 0,5 para o “Sensor_02”, com uma largura de banda disponível de 1, é necessário escalonamento. Passamos gráfico que apresenta a quantidade de informação dos sensores por unidade de tempo.

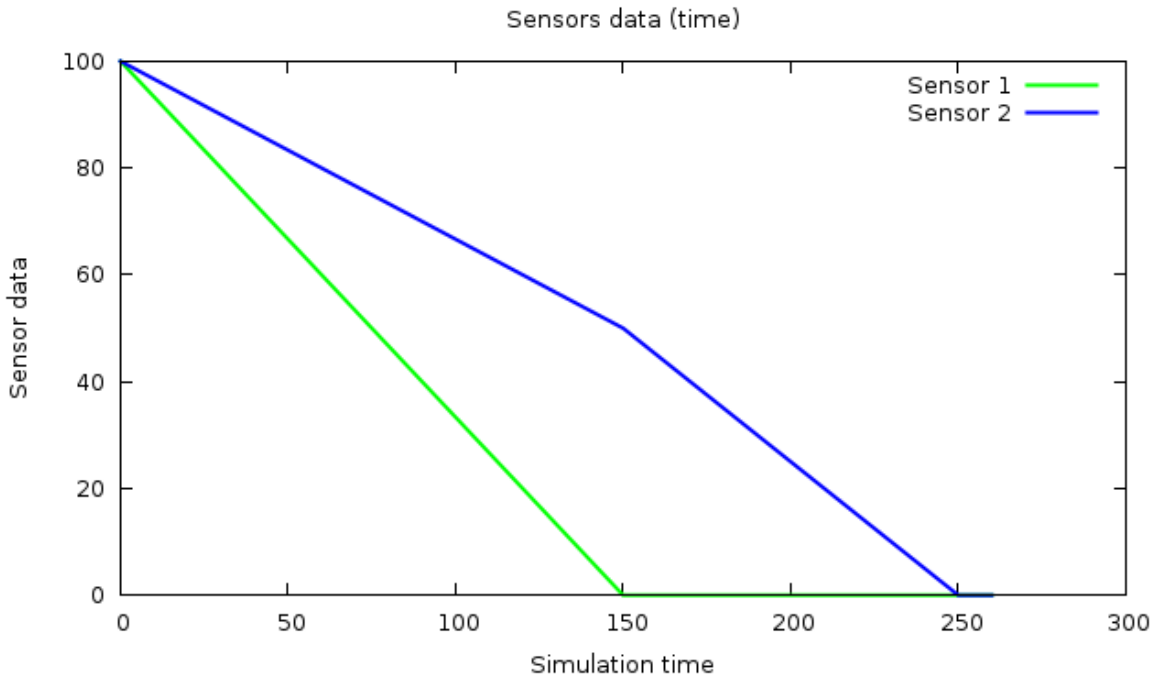

Figura 10: Decaimento de informação dos sensores por unidade de tempo

Da figura acima podemos retirar que até 150 unidades de tempo, ambos os sensores tem informação para transmitir. Como tal, durante esse intervalo é realizado escalonamento. Passo o referido intervalo, verifica-se que o decaimento de informação do "Sensor 1" é maior uma vez que a largura de banda disponível passa a ser toda utilizada por este.

Recorrendo à análise matemática das formulas definas para o escalonamento, conclui-se que entre 0 e 150 unidades de tempo as taxas de transmissão são:

$$t_1 = t_{1_{max}} = 1$$
$$t_2 = t_{2_{max}} = 0{,}5$$
$$LB_t = \sum_{i=1}^{2} t_i = t_1 + t_2 = 1 + 0{,}5 = 1{,}5$$
$$t'_1 = \frac{t_1 LB_d}{LB_t} = \frac{1 \times 1}{1{,}5} = 0{,}67$$
$$t'_2 = \frac{t_2 LB_d}{LB_t} = \frac{0{,}5 \times 1}{1{,}5} = 0{,}33$$

sendo:

- t_i a taxa de transmissão instantânea do sensor i ;
- $t_{i_{max}}$ a taxa máxima de transmissão do sensor i ;
- LB_t a largura de banda necessária, sem escalonamento;
- t'_i a taxa de transmissão instantâneo do sensor após escalonamento;
- LB_d a largura de banda disponível.

Das contas realizadas espera-se que no intervalo de 0 a 150 o sensor 1 tenha uma taxa de transmissão de 0,67 e o sensor 2 de 0,33. Fora do referido intervalo a taxa de transmissão do sensor 2 é de 0,5. De acordo com o esperado, e por comparação com a figura abaixo, podemos verificar que os resultados são os esperados.

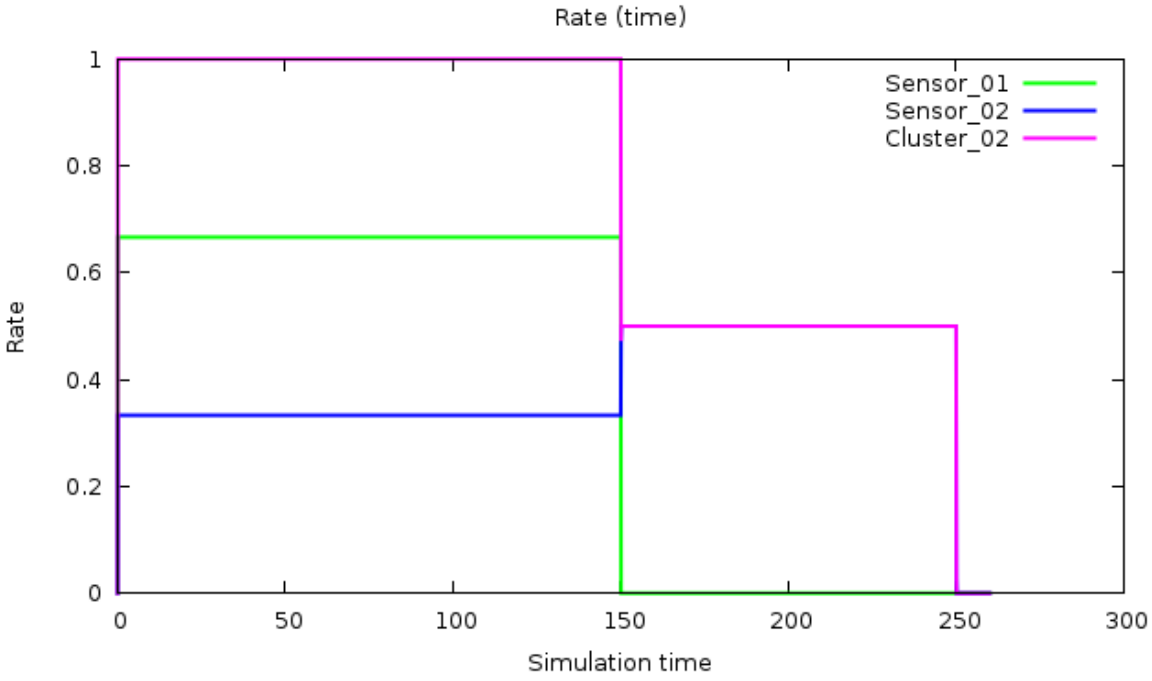

Figura 11: Taxas de transmissão dos sensores e largura de banda ocupada

Vou apresentar novamente os resultados de uma simulação nas mesmas circunstâncias que a anterior apenas igualando a 1 as taxa de transmissão máxima dos sensores e afastando os sensores do ponto 0, 0 e afastando-os entre-si, como podemos ver pela figura abaixo.

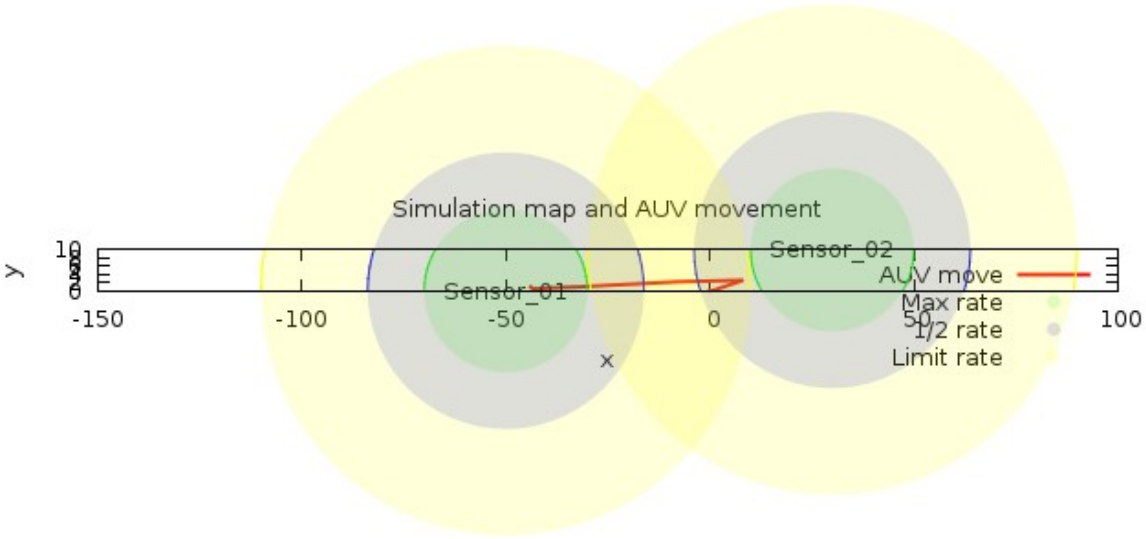

Figura 12: Mapa da simulação

Mais uma vez refiro que o trajeto do veículo é decidido pelo controlador do mesmo. Como se pode ver o veículo caminha em direção ao “Sensor_02” e de seguida volta para o “Sensor_01”. Entende-se que o veículo vai em direção do “Sensor_02” até que seja ocupada toda a largura de banda. Seguidamente caminha em direção ao “Sensor_01” dado não haver mais informação disponível no “Sensor_02”. Durante este trajeto deveremos verificar o aumento da ocupação da largura de banda até ser atingido o limite. Como se pode verificar no gráfico abaixo.

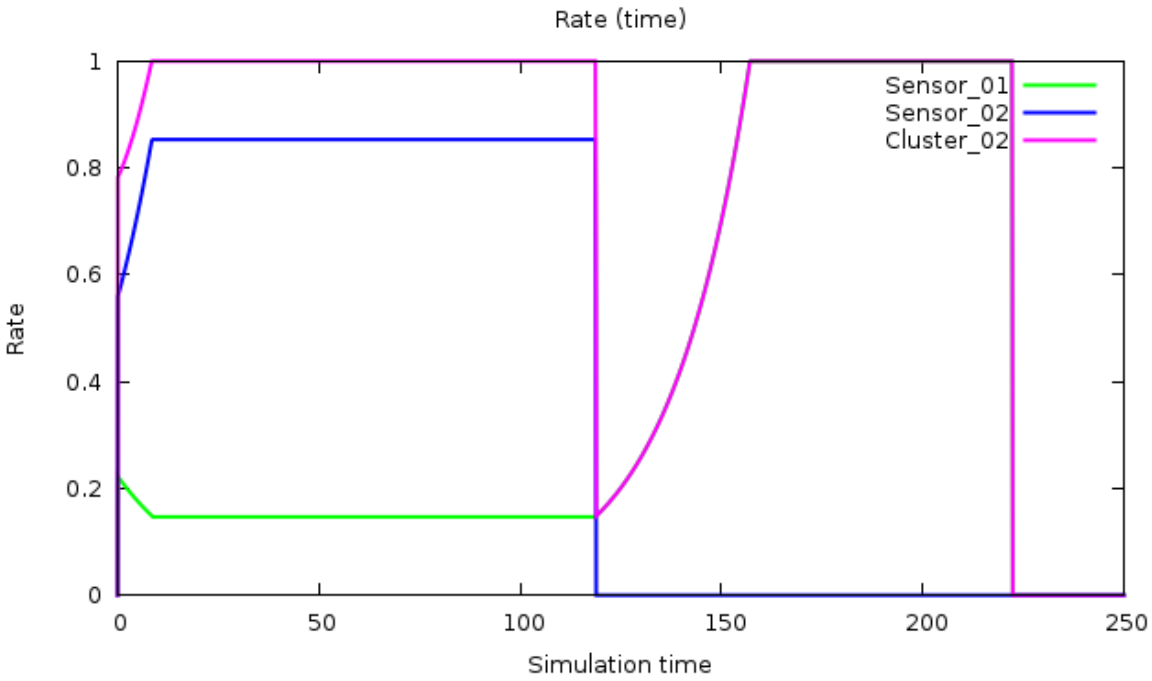

Figura 13: Taxas de transmissão dos sensores e largura de banda ocupada

Dou por concluída a explicação e consequente validação dos resultados das simulações, bem como a funcionalidade da biblioteca desenvolvida.

Mas, como em todo, o código desenvolvido é passível de evoluções. Evoluções em termos de maturidade dos algoritmo ou em termos de incremento de funcionalidades. Vou apenas referir as evoluções em termos de funcionalidades. Para efeitos de desenvolvimento futuro faço referencia aos dois pontos que considero mais relevantes:

- Gestão de clusters – Seguindo a definição de software usada para o desenvolvimento da classe sensor e da classe cluster, uma da evoluções que pode ser realizada é a gestão de clusters. Neste momento, de forma controlada, em cada simulação, apenas pode existir um cluster, isto é, todos os sensores interagem entre sim mas sempre pertencentes ao mesmo cluster. Desenvolvendo uma gestão de clusters, pode-se simular cenários com diferentes clusters passando os clusters a gerir-se entre si e concordância com o veículo.
- Mapa de comunicação – Desenvolver uma ferramenta que permita a importação do mapa de comunicação de cada sensor. O código apresenta um mapa de comunicação de cada sensor fixo, variando apenas a taxa máxima de transmissão. Será de alguma relevância alterar este mapa. Neste momento para alterar o mapa de cada sensor é necessário criar uma classe deriva da classe ComRate.

Referências Documentais

Anexo A. Ficheiros de código C++

Neste anexo são apresentados todos os ficheiros de código C++ que constituem a biblioteca.

Accumulador.cpp

Histórico

- 3 de Setembro de 2012, Versão 1.0, mailto:1000146@isep.ipp.pt

  $Id: report.odt v1.0 Date: 03-11-2012$