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

7 de Novembro de 2012

Agradecimentos

Agradeço o apoio inconstitucional da minha família e amigos que como sempre quebram a rocha que impede a 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 capítulo 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 activa, na qual se pretende estudar diferentes cenários de aquisição de informação/dados em localizações remotas com recurso a veículos autónomos, tornou-se necessário a estruturação e desenvolvimento de código a utilizar em simulação computacional. 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 se entra numa situação em que se considera que simultaneamente pode haver 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 objectivos chave a ter em consideração no desenvolvimento.

Com base na informação recolhida e de uma forma geral, foram definidos os seguintes objectivos chave: abstracção na codificação, para permitir a reutilização de código e expansão futura; simplicidade de conceitos e políticas de escalonamento, para permitir a validação do correcto funcionamento da biblioteca; e optimização 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 controlador de veículos autónomos, e são apenas utilizadas as bibliotecas standard de C++, com excepção da parte referente à importação de ficheiros de configuração – ficheiros XML – caso em que é usada a "libxml++", *wrapper* para C++ da biblioteca "libxml2".

A validação de funcionamento do código desenvolvido é realizada pela utilização da biblioteca 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

Inline with an active research area, in which is intended to study different scenarios of data collection from remote locations using autonomous vehicles, arises the necessary to structure and to develop a set of code to use in computer simulation. It raises the problem to solve.

It’s intended to develop a library, for simplification of concepts designated as library of sensors, that shall be able to interact with existing simulation code, code tested and developed in C + +.

Considering that in a set of sensors there’s the possibility of more than one sensor being able to communicate at the same time, arises the scenario where there’s various sensors communicating simultaneously and/or it’s exceeded the available channel bandwidth. So, the library shall be able to manage the information at each instant and apply the proper bandwidth scheduling in accordance with the communication channel availability.

Thus, it was developed the knowledge, concepts and methods of programming in language C + +, it has been studied and analyzed different scenarios of development and were defined the key objectives to be taken into account during the development.

Based on the collected information and in general, the following key objectives were set: coding abstraction, to enable code reuse and future expansion; simplicity of concepts and scheduling policies, to enable the validation of the library correct operation; and execution performance optimization, to allow the use of the library with simulators extremely "heavy" in terms of computational resources.

Realizing, the code is developed in C + + language, ensuring continuity of existing work, namely the autonomous vehicles simulator, and are only used standard libraries of C + +, with the exception to the code related with the importation of configuration files - XML files - in which is use "libxml + +", a C++ wrapper of library "libxml2".

The validation of the developed code is done by the use of the library with the existing simulation code and by the analysis of the outputs alongside with the technical specification defined in this document.

Keywords

AUV, AGV, sensors, cluster, scheduling, C++, library, data collection, simulation.

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

AGV – Automated Guided Vehicle

AUV – Autonomous Unamed Vehicle

GCC – GNU Compiler Collection

GDB – GNU Project Debugger

GSL – GNU Scientific Library

STL – Standard Template Library

UNIX™ – UNiplexed Information and Computing System

XML – eXtensible Markup Language

™ UNIX is a registered trademark of The Open Group

1. INTRODUÇÃO

Neste capítulo é apresentada a orientação, a contextualização e o guia do projecto, 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, obriga o desenvolvimento de protótipos de simulação computacional para recriar cenários reais, sistemas reais. Com os resultados das referidas simulações são desenvolvidas as devidas 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. Evoluir é, também, acrescentar novas funcionalidades.

Porquê um modelo simples? Na realidade a validação das aplicações é sempre o objectivo mais complexo. Assim, para facilitar a validação de resultados, regra geral, é criado um modelo simples mas preparado para evoluir.

1.2. CONTEXTUALIZAÇÃO

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

Assim, decide-se criar uma biblioteca de sensores e respectivos métodos de interacção com a mesma, permitindo a utilização da referida biblioteca em modelos de simulação já desenvolvidos e testados.

O código de simulação existente permite usar os algoritmos de resolução numérica de equações diferenciais da GNU Scientific Library (GSL) ou outros implementados pelo próprio programador, desde que obedeçam à interface definida. Como tal, o código assume que os modelos são descritos por sistemas de equações diferenciais de primeira ordem, de forma semelhante ao estudado na unidade curricular de Teoria dos Sistemas.

Partindo da base do código já existente e tendo em conta que todo este código faz uso da linguagem de programação C++, torna-se evidente a utilização do mesmo tipo de linguagem sendo mesmo um dos requisitos do projecto.

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 interacção com código desenvolvido por terceiros.

1.3. OBJECTIVOS

De acordo com os pontos anteriores define-se como principal objectivo a criação de uma biblioteca de sensores. Biblioteca que terá de gerir a informação contida em cada sensor, bem como a largura de banda disponível no canal de comunicação e respectiva interacção com os referidos modelos de simulação.

Partindo do zero, e para viabilizar a realização deste projecto, é necessária a estruturação e consequente divisão do objectivo principal num conjunto de tarefas e objectivos. Desta forma, definem-se os seguintes pontos:

- Estudar a linguagem de programação C++, incluindo conceitos, modos de programação e de optimização; Objectivo: familiarização com a linguagem de programação e aquisição de conhecimentos suficientes ao desenvolvimento necessário;
- 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;
- Estudar e implementar um método de importação das configurações dos sensores; Objectivo: 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

Com a concretização das tarefas acima enumeradas é garantido o sucesso deste projecto e consequente concretização do objectivo principal. O detalhe expresso pretende obrigar a consideração dos princípios a ter num conceito dito de desenvolvimento estruturado e organizado, nomeadamente: modelização, abstracção, simplicidade e performance.

Questões relacionadas com o desempenho, tornam-se severas se considerarmos que não se pretende comprometer futuras aplicações, bem como a expansão do sistema desenvolvido no presente.

1.4. CALENDARIZAÇÃO

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

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

- Lançamento do projecto 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, tendo sido a validação da biblioteca 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 projecto, 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 capítulo, 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 perspectiva 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 objectivo, o código desenvolvido permite a utilização na forma de aplicação.

Por último temos o capítulo Conclusões, onde normalmente se escreve que os objectivos foram alcançados com sucesso. Pois embora de uma forma simplista é exactamente 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 para o desenvolvimento do mesmo, incluindo métodos de desenvolvimento e mecanismos de suporte ao programador.

2.1. 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 dialecto, 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 dialectos 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 dialectos 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 actual 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 seguinte:

```
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 determinada tarefa, por exemplo, uma operação matemática. Uma função pode devolver um parâmetro e aceita vários parâmetros de entrada. À semelhança do C, uma função, em C++, 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, respectivamente, 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 directivas do pré-processador. Os detalhes associados ao pré-processador não são apresentados neste relatório. Alias, não é apresentado qualquer detalhe sobre o processo de complilação. Interessa reter que esta linha de código, directiva, solicita ao pré-processador a introdução da biblioteca contida entre os símbolos <>, neste caso a biblioteca `iostream`. A biblioteca `iostream` 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 respectiva identificação, por exemplo, `std::cout`, `std::endl`.

Para evitar a utilização do identificador `std` pode ser usada a seguinte linha de código `using namespace std`.

Assim, em termos de codificação pode ser utilizado:

```
using namespace std;

cout << “Hello wolrd!” << endl;
```

ou

```
std::cout << “Hello wolrd!” << std::endl;
```

Ambas as linhas de código, `cout << “Hello wolrd!” << endl` e `std::cout << “Hello wolrd!” << std::endl` fazem uso das funcionalidades da biblioteca padrão `iostream`.

De notar que todas a linhas de código são terminadas com ponto e vírgula. O ponto e vírgula faz parte da sintaxe do C++. Este símbolo diz ao compilador que é o fim de uma expressão.

Como referido atrás, uma função devolve um parâmetro. A expressão `return`, para além de causar a conclusão da função (neste caso função `main`), é usada para devolver o parâmetro.

No código é possível introduzir 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 diálogo é necessário manipular e reter um conjunto de dados durante a execução da referida aplicação. Em programação estes dados são guardados em variáveis. Há diferentes tipos de variáveis para o efeito. As variáveis são 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 projecto 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 abaixo – Tabela 1. Mais à frente é mencionado como os referidos operadores podem ser utilizados em classes.

A tabela 2, Operadores, apresenta um resumo dos principais tipos de dados em C++.

Tabela 1 Tipos de dados

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

Tabela 2 Operadores

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

No campo das variáveis, nomeadamente no que diz respeito à criação das mesmas, há ainda dois aspectos 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

A execução de um programa não se 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 projecto são tipicamente utilizadas duas estruturas de controlo: uma condicional e uma de repetição, `if` e `for` respectivamente.

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 uma cadeia condicional, `if else`, a sintaxe e comportamento é:

```
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. CLASSES

A decisão de utilização da linguagem de programação C++ deve-se muitas vezes ao facto da existência de classes. Uma classe é um conjunto de código, um conjunto de 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
};
```

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.

Para alterar os dados da classe, membros de dados, utiliza-se 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. Os métodos são definidos de acordo com a seguinte sintaxe:

```
nome_da_classe::método
```

concretizando:

```
// 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 correcto 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 usa-se a seguinte:

```
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 se pode 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.

2.7. CLASSES DERIVADAS

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”. Pode-se 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 gráfica é apresentada na figura seguinte:

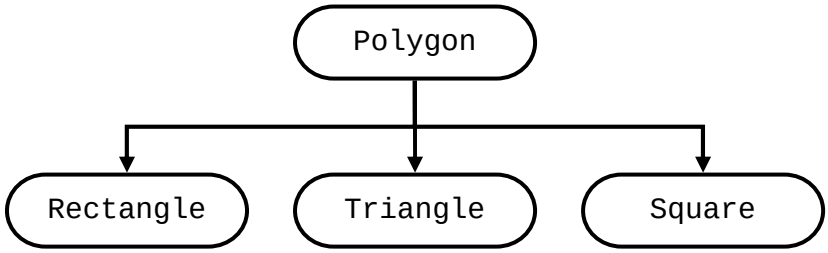

Figura 1 Classes derivadas

2.8. VECTORES

O C++ disponibiliza um conjunto de estruturas de dados e respectivos operadores/funções. Todas estas estruturas estão disponíveis na Standard Template Library (STL). No âmbito deste projecto apenas é detalhada a estrutura de vectores.

Por estrutura de vectores, entenda-se um contentor sequencial de objectos de determinado tipo e com acesso directo. A grande vantagem da utilização de vectores 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, os diferentes elementos dos vectores são de acesso directo, isto é, é possível aceder a cada elemento do vector 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;
```

2.9. APLICAÇÕES DE SUPORTE AO DESENVOLVIMENTO

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

- Vim: editor de texto, desenvolvido para maximizar a eficiência da escrita;
- GCC: ferramentas de complicação do projecto 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 capítulo são apresentadas as principais funções desenvolvidas e toda a estrutura do código. Após um conjunto de considerações introdutórias, para simplicidade de enquadramento e apresentação, o capítulo 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 eXtensible Markup Language (XML). No contexto deste projecto 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 o Gnuplot e o 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 directório `config` agrupa um conjunto de ficheiros XML, ficheiros de configuração de sensores para utilizar nas simulações. O directório `gnuplot` contem um script para criar um conjunto de representações gráficas para apresentação de resultados. O directório `log`, agrupa os ficheiros de dados resultantes da simulação. O directório `sim`, contem o código da ferramenta de simulação não objecto de detalhe neste documento uma vez que foi desenvolvida por terceiros.

Voltando ao ficheiros da raiz o ficheiro `comlibsim.{cpp,hpp}`[2] 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>.
```

[2] 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:

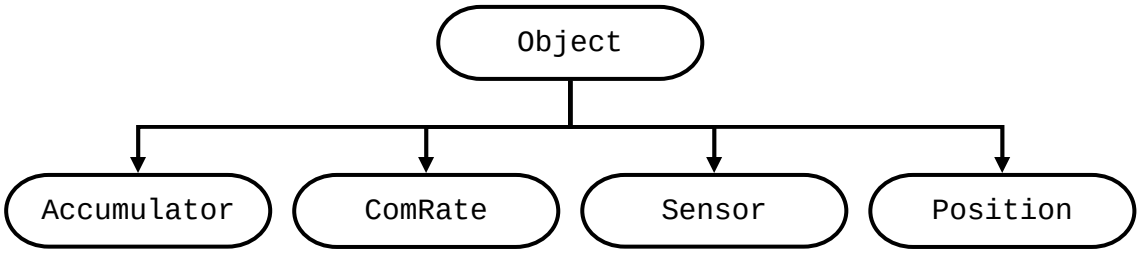

Figura 2 Ligação entre classes

3.3. ESPAÇO DE NOMES (NAMESPACE)

De acordo com o capítulo anterior, para o desenvolvimento da biblioteca, para todo o código 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 referido 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 respectivos 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 distância 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 trajectória do veículo 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, sabe-se que a taxa de transmissão varia em função da distância entre emissor receptor. Assim, a classe `ComRate` define o mapa de comunicação e respectiva 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 o 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 activa; 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 é também disponibilizado 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 (três) 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 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 é também disponibilizado 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 deste projecto 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 mesmos interagem com outro elemento comunicante, nomeadamente um veículo. Considerando um veículo 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 objectivo de orientar o veiculo;
- Seleccionar quais os sensores com condições para transmitir em cada instante;
- E atribuir a cada sensor a taxa de transmissão para negociação com o veículo.

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 veículo, 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:

$$t_i(d) = t_{imax} \text{ , se } 0 < d < 20 \text{ .} \quad (\mathbf{1})$$

$$t_i(d) = t_{max}\, e^{1-(d/20)} \text{ , se } 20 \leq d < 60 \text{ .} \quad (\mathbf{2})$$

$$t_i(d) = 0 \text{ , se } d \geq 60 \text{ .} \quad (\mathbf{3})$$

sendo:

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

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

$$LB_t = \sum_i^n t_i \text{ .} \quad (\mathbf{4})$$

sendo:

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

De acordo com a largura disponível (LB_d) 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:

$$t'_i(d) = t_i(d)\, LB_d / LB_t \text{ .} \quad (\mathbf{5})$$

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 política de escalonamento apresentada é apenas uma das possíveis, por exemplo, atribuir prioridades a sensores gerindo o escalonamento de acordo com as referidas prioridades.

O principal elemento que constituem a classe é:

- `std::vector<Sensor> m_sensors` – vector 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 veículo;
- `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 seleccionar 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 é também disponibilizado 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 respectivos sensores.

À semelhança das classes anteriores, nesta é também disponibilizado 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

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 capítulo, 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 a instalar 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 executa-se 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 ficheiros XML na pasta `config` para simulações.

4.3. SIMULAÇÕES E RESPECTIVA ANÁLISE

Realiza-se a primeira simulação, 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
```

O resultado gráfico é apresentado nas 3 (três) figuras abaixo:

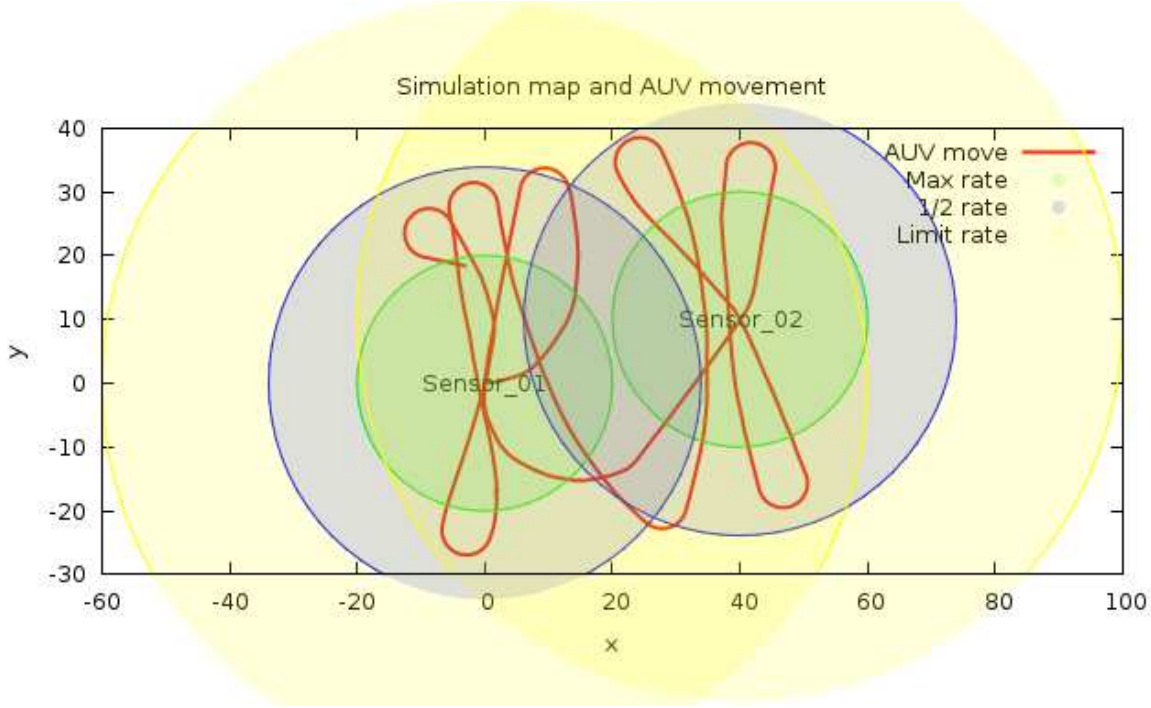

Figura 3 Mapa da simulação com dois sensores – Simulação I

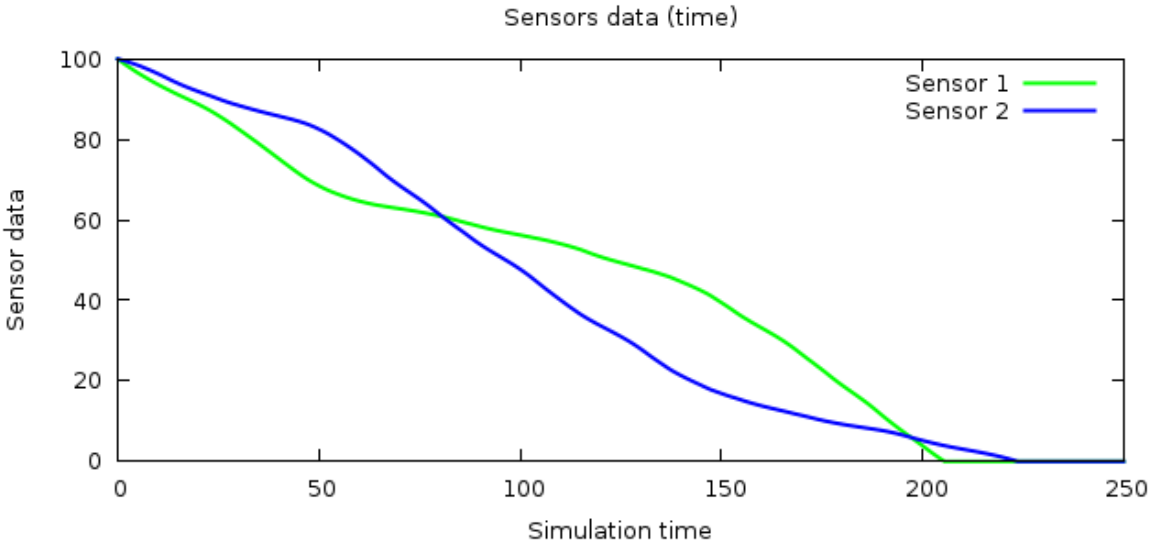

Figura 4 Quantidade de informação disponível, simulação com dois sensores – Simulação I

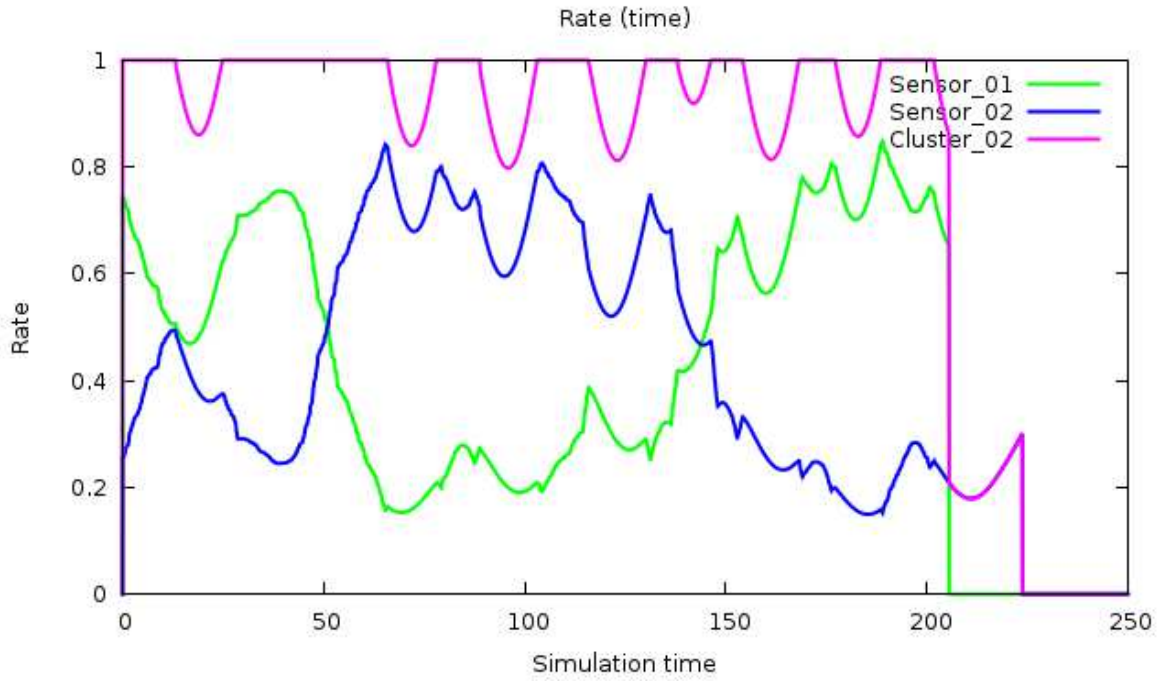

Figura 5 Taxas de transmissão, simulação com dois sensores – Simulação I

Verifica-se 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 têm dados, isto é, durante a aquisição de dados o veículo nunca para.

Ao longo do tempo observar-se 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 – resultante da aplicação do escalonamento.

Mais uma prova do descrito é o resultado obtido numa 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
```

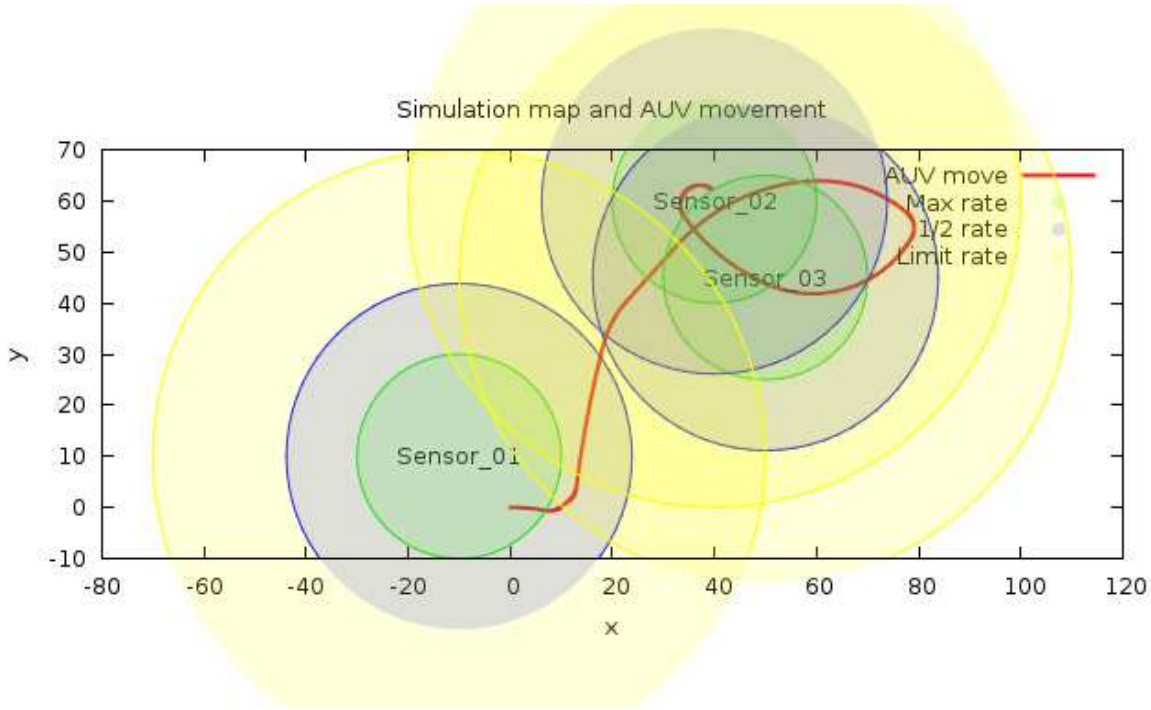

Figura 6 Mapa da simulação com três sensores – Simulação II

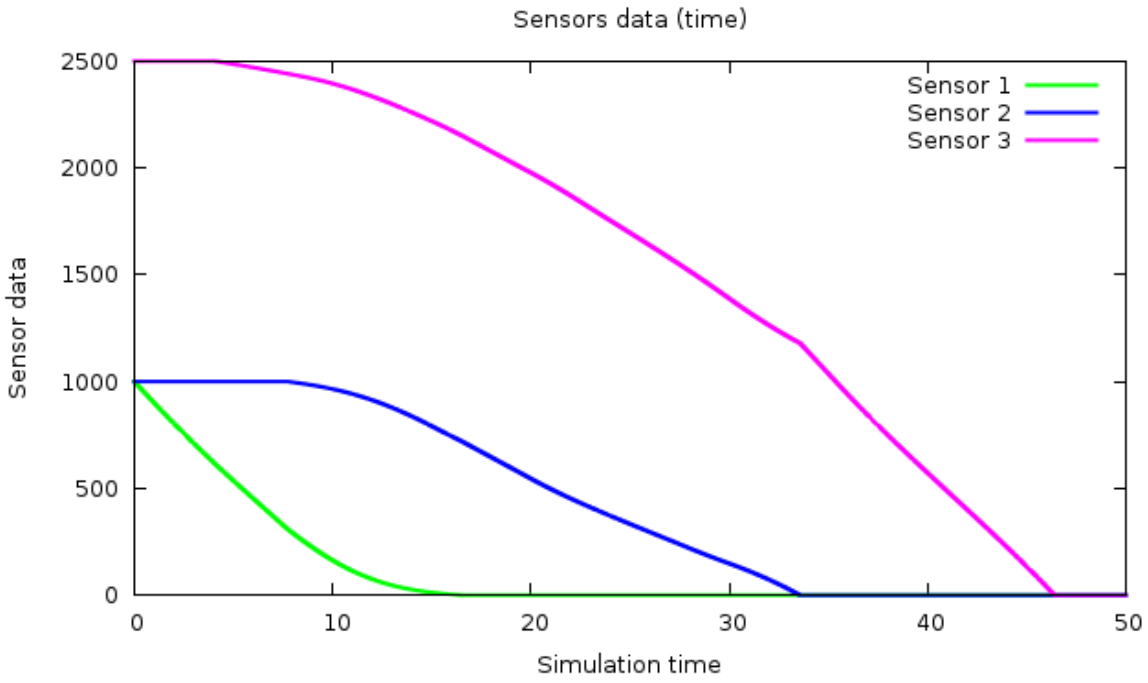

Figura 7 Quantidade de informação disponível, simulação com três sensores – Simulação II

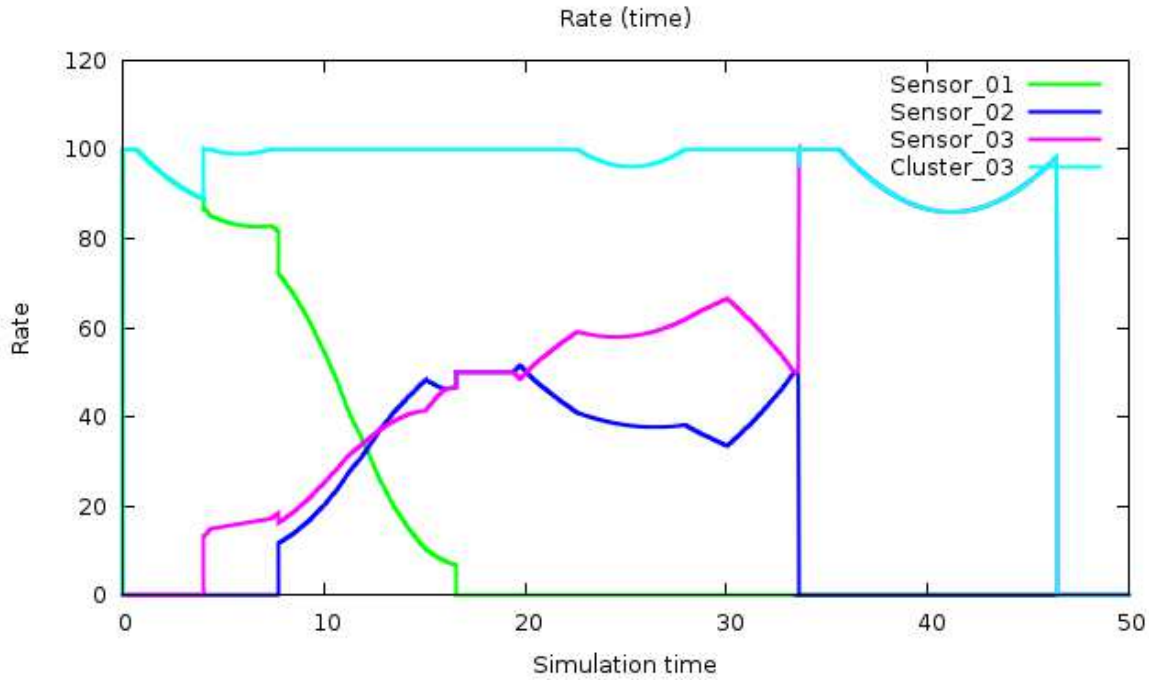

Figura 8 Taxas de transmissão, simulação com três sensores – Simulação II

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

Em ambas as simulações 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.

As simulações apresentadas até ao momento, são complexas e consequentemente os resultados são complexos tornando a realização de um texto de análise extenso. De seguida é apresentada 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:

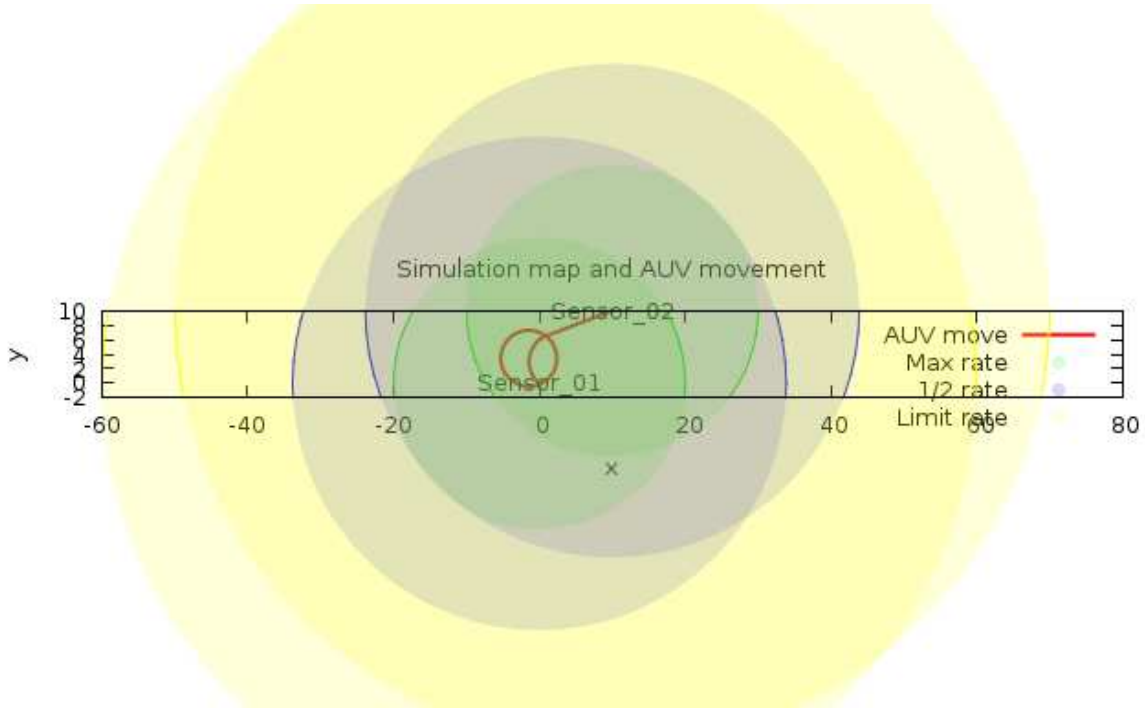

Figura 9 Mapa da simulação com dois sensores – Simulação III

O trajecto do veículo é completamente alheio à biblioteca e fora do âmbito deste projecto. 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.

No gráfico abaixo é apresentada a quantidade de informação dos sensores por unidade de tempo.

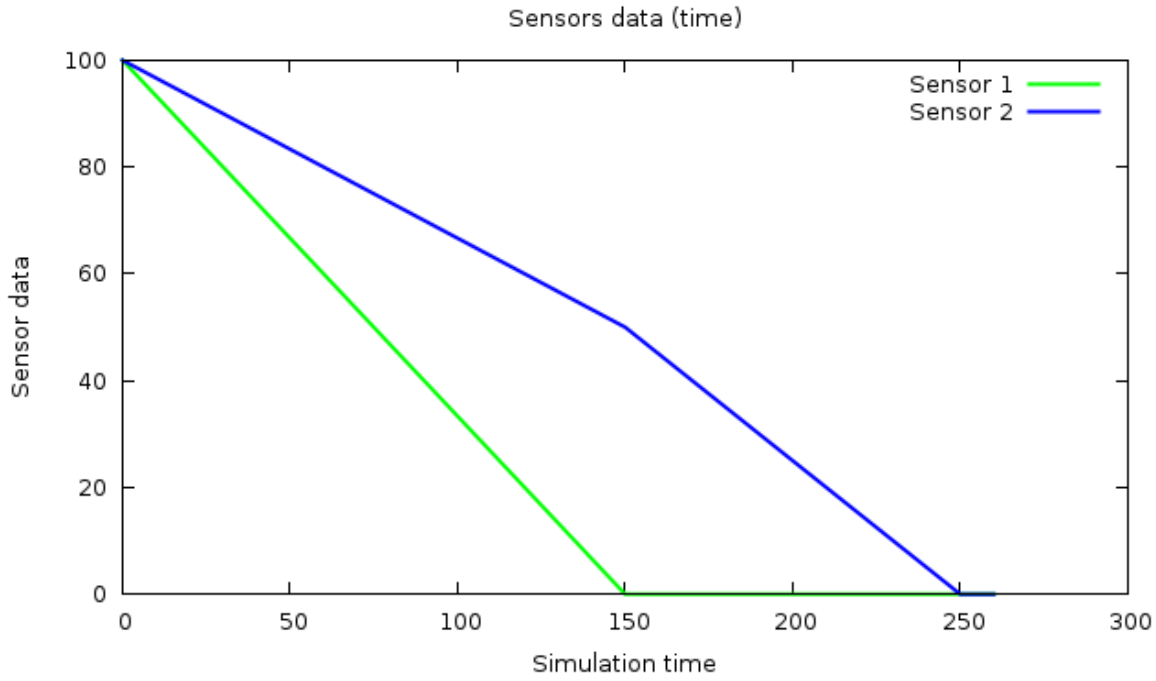

Figura 10 Decaimento de informação dos sensores por unidade de tempo – Simulação III

Da figura acima verifica-se que até 150 unidades de tempo, ambos os sensores tem informação para transmitir. Como tal, durante esse intervalo é realizado escalonamento. Passado 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 e de acordo com as equações definidas para o escalonamento, conclui-se que entre 0 e 150 unidades de tempo as taxas de transmissão são:

$$t_1 = t_{1max} = 1 \, . \qquad (\mathbf{6})$$

$$t_2 = t_{2max} = 0{,}5 \, . \qquad (\mathbf{7})$$

$$LB_t = \sum_i^2 t_1 + t_2 = 1 + 0{,}5 = 1{,}5. \qquad (\mathbf{8})$$

$$t'_1 = t_1 \, LB_d / LB_t = 1*1 / 1{,}5 = 0{,}67 \, . \qquad (\mathbf{9})$$

$$t'_2 = t_2 \, LB_d / LB_t = 0{,}5*1 / 1{,}5 = 0{,}33 \, . \qquad (\mathbf{10})$$

sendo:

- t_i a taxa de transmissão instantânea de transmissão do sensor i;
- t_{imax} a taxa máxima de transmissão do sensor i;
- LB_t a largura de banda necessária pelo *cluster*, sem escalonamento;
- LB_d a largura de banda disponível no canal de transmissão.

Dos cálculos realizados 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, verificar-se que os resultados são os esperados.

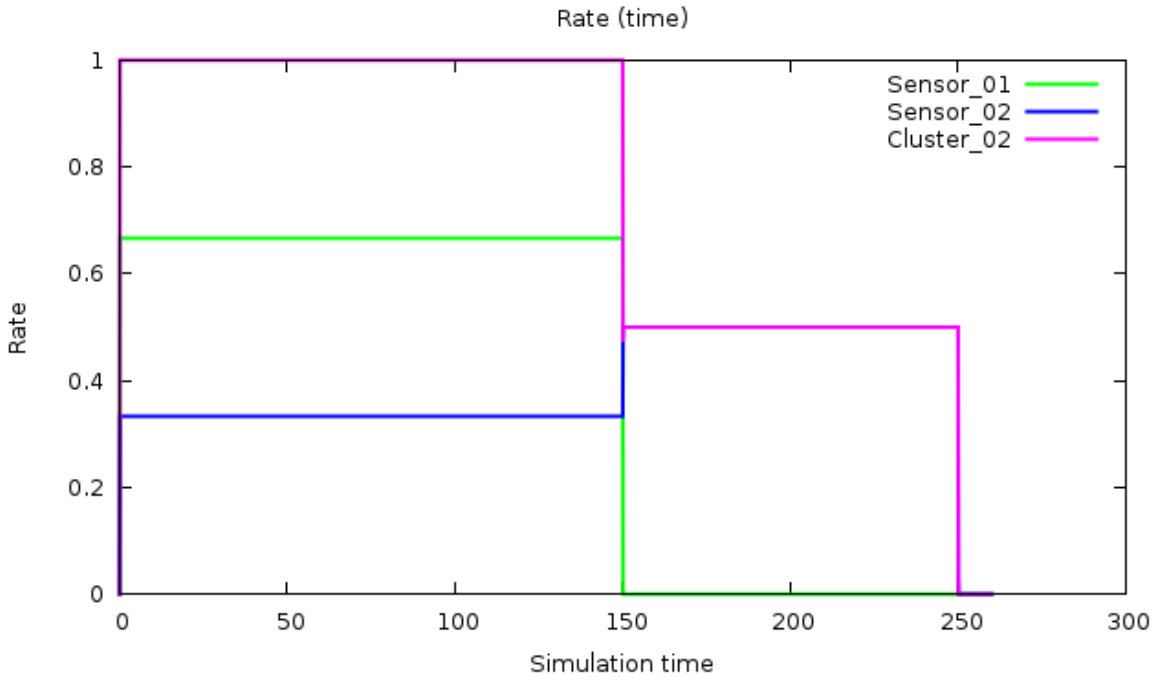

Figura 11 Taxas de transmissão dos sensores e largura de banda ocupada – Simulação III

De seguida apresentam-se 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 se observa na figura abaixo.

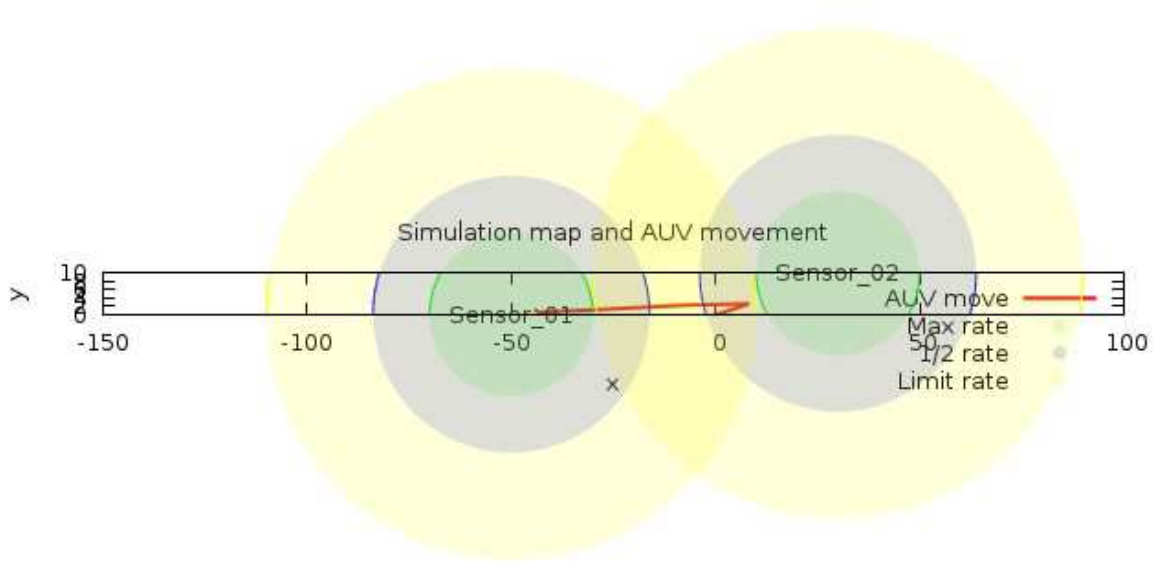

Figura 12 Mapa da simulação – Simulação IV

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

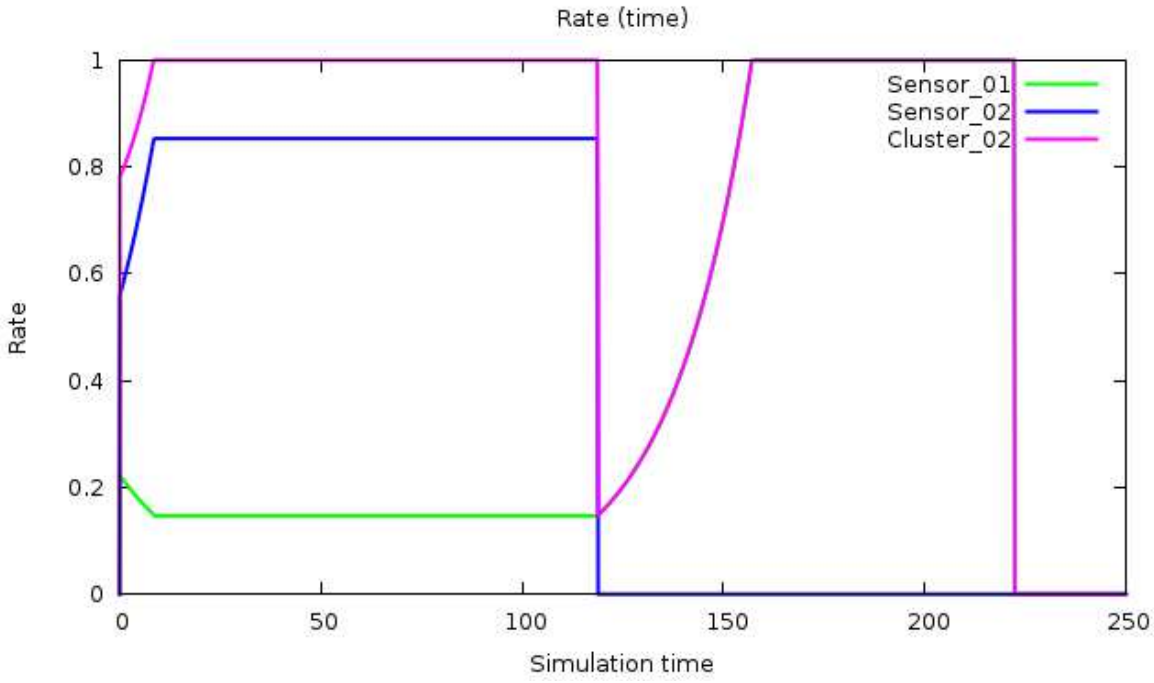

Figura 13 Taxas de transmissão dos sensores e largura de banda ocupada – Simulação IV

5. CONCLUSÕES

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

Pode-se facilmente verificar que os resultados apresentados no capítulo anterior traduzem à conclusão de que as decisões tomadas e as definições levadas em linha de conta no desenvolvimento foram correctas, nomeadamente pelo facto da taxa de ocupação da largura de banda e correcto 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.

Com base nos gráficos apresentados 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 correcto 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 em comparação com os valores esperados.

Relativamente a futuros desenvolvimentos, o código é passível de evoluções. Evoluções em termos de maturidade dos algoritmos ou em termos de incremento de funcionalidades. Refere-se apenas as evoluções em termos de funcionalidades. Para efeitos de desenvolvimento futuro faz-se referência a dois pontos que se considera serem os 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

[1] ALLAN Alex (creator of cprogramming.com), *Jumping into C*.

[2] Website, http://www.cplusplus.com/.

[3] man development pages of linux.

Anexo A. Listagem de código

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

```
/*
 *  accumulator.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "accumulator.hpp"

namespace ComLibSim
{

Accumulator::Accumulator ():
  m_amount_data (0.0)
{
}

Accumulator::Accumulator (double amount_data):
  m_amount_data (amount_data)
{
}

Accumulator::Accumulator (const Accumulator& accumulator):
  m_amount_data (accumulator.m_amount_data)
{
}

Accumulator::~Accumulator ()
{
}

Object* Accumulator::object () const
{
  return new Accumulator (*this);
}

void Accumulator::copy (const Accumulator& accumulator)
{
  if (this != &accumulator)
    m_amount_data = accumulator.m_amount_data;
}

Accumulator& Accumulator::operator = (const Accumulator& accumulator)
{
  this->copy (accumulator);
```

```
  return *this;
}

bool Accumulator::operator == (const Accumulator& accumulator) const
{
  return m_amount_data == accumulator.m_amount_data;
}

bool Accumulator::operator != (const Accumulator& accumulator) const
{
  return ! operator == (accumulator);
}

bool Accumulator::operator > (const Accumulator& accumulator) const
{
  return m_amount_data > accumulator.m_amount_data;
}

bool Accumulator::operator < (const Accumulator& accumulator) const
{
  return m_amount_data < accumulator.m_amount_data;
}

bool Accumulator::operator >= (const Accumulator& accumulator) const
{
  return m_amount_data >= accumulator.m_amount_data;
}

bool Accumulator::operator <= (const Accumulator& accumulator) const
{
  return m_amount_data <= accumulator.m_amount_data;
}

double Accumulator::get_amount_data () const
{
  return m_amount_data;
}

void Accumulator::set_amount_data (double amount_data)
{
  m_amount_data = amount_data;
}

bool Accumulator::is_empty () const
{
  return m_amount_data <= 0.0;
}

void Accumulator::write (std::ostream& output) const
{
  output << "Amount data = " << m_amount_data;
}

void Accumulator::write_ln (std::ostream& output) const
{
  this->write (output);

  output << std::endl;
}
```

```
void Accumulator::write_log (std::ostream& output) const
{
  output << std::fixed << std::setprecision (10);
  output << m_amount_data;
}

std::ostream& operator << (std::ostream& output,
                           const Accumulator& accumulator)
{
  accumulator.write (output);

  return output;
}

}
```

```
/*

 *  accumulator.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_ACCUMULATOR__
#define __H_ACCUMULATOR__

#include <iostream>
#include <iomanip>

#include "object.hpp"

namespace ComLibSim
{

class Accumulator: public Object
{
private:
  double m_amount_data;

public:
  Accumulator ();
  Accumulator (double amount_data);
  Accumulator (const Accumulator& accumulator);
  virtual ~Accumulator ();

  virtual Object*      object     () const;
  virtual void         copy       (const Accumulator& accumulator);
  virtual Accumulator& operator = (const Accumulator& accumulator);

  virtual bool operator == (const Accumulator& accumulator) const;
  virtual bool operator != (const Accumulator& accumulator) const;
  virtual bool operator >  (const Accumulator& accumulator) const;
  virtual bool operator <  (const Accumulator& accumulator) const;
  virtual bool operator >= (const Accumulator& accumulator) const;
  virtual bool operator <= (const Accumulator& accumulator) const;

  virtual double get_amount_data () const;

  virtual void set_amount_data (double amount_data);

  virtual bool is_empty () const;

  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);
};
```

```
}

#endif
```

```
/*
 *  cluster.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "cluster.hpp"

namespace ComLibSim
{

double Cluster::ComMap::select (std::vector<Sensor>& sensors,
                                const Position& position)
{
  double bandwidth = 0.0;

  for (std::vector<Sensor>::iterator i = sensors.begin ();
       i != sensors.end ();
       i++)
  {
    double rate = i->rate_at (position);

    if (!i->is_empty () && rate != 0.0)
    {
      bandwidth += rate;
      i->rate (rate);
      this->push_back (i);
    }
    else
      i->rate (0.0);
  }

  return (- bandwidth);
}

void Cluster::ComMap::write (std::ostream& output) const
{
  for (ComMap::const_iterator i = begin ();
       i != end ();
       i++)
  {
    (*i)->write_ln (output);
  }
}

Cluster::Cluster (int nb):
  m_sensors(),
  m_nb_act_sensors(0),
  m_act_bandwidth(0.0),
  m_scheduling(false)
{
  m_sensors.reserve (nb);
}
```

```
Cluster::~Cluster ()
{
}

int Cluster::nb_act_sensors () const
{
  return m_nb_act_sensors;
}
double Cluster::act_bandwidth () const
{
  return m_act_bandwidth;
}

bool Cluster::scheduling () const
{
  return m_scheduling;
}

bool Cluster::is_empty () const
{
  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    if (!i->is_empty ())
      return false;
  }
      return true;
}
int Cluster::nb_sensors () const
{
  return static_cast<int>(m_sensors.size ());
}

double Cluster::bandwidth () const
{
  return m_bandwidth;
}

void Cluster::tag (const std::string& tag)
{
  m_tag = tag;
}

void Cluster::add (const Sensor& sensor)
{
  m_sensors.push_back (sensor);
}

void Cluster::init_int ()
{
  m_sensors_int = m_sensors;
}

void Cluster::copy_int ()
{
  m_sensors = m_sensors_int;
}

void Cluster::get_data (double *data) const
{
```

```
  int j = 0;

  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    data[j] = i->data ();
    j++;
  }
}

void Cluster::get_data_int (double *data) const
{
  int j = 0;

  for (std::vector<Sensor>::const_iterator i = m_sensors_int.begin ();
       i != m_sensors_int.end ();
       i++)
  {
    data[j] = i->data ();
    j++;
  }
}

void Cluster::get_rate (double *rate) const
{
  int j = 0;

  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    rate[j] = i->rate ();
    j++;
  }
}

void Cluster::get_rate_int (double *rate,
                            const double *position,
                            double agv_bandwidth)
{
  int j = 0;
  Position position_int (position[0], position[1]);

  this->map_int (position_int, agv_bandwidth);

  for (std::vector<Sensor>::const_iterator i = m_sensors_int.begin ();
       i != m_sensors_int.end ();
       i++)
  {
    rate[j] = i->rate ();
    j++;
  }
}

void Cluster::set_data (double *data, double delta_time)
{
  int j = 0;
  m_bandwidth = 0.0;

  for (std::vector<Sensor>::iterator i = m_sensors.begin ();
```

```
       i != m_sensors.end ();
       i++)
  {
    i->data (data[j], delta_time);
    m_bandwidth += i->bandwidth ();
    j++;
  }
}

void Cluster::set_data_int (const double *data)
{
  int j = 0;

  for (std::vector<Sensor>::iterator i = m_sensors_int.begin ();
       i != m_sensors_int.end ();
       i++)
  {
    i->data (data[j]);
    j++;
  }
}

void Cluster::set_rate (double *rate)
{
  int j = 0;

  for (std::vector<Sensor>::iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    i->rate (rate[j]);
    j++;
  }
}

Sensor& Cluster::closest (const Position& position)
{
  double ref_distance = 0.0;
  double distance = 0.0;

  std::vector<Sensor>::iterator selected = m_sensors.begin ();

  for (std::vector<Sensor>::iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    distance = i->distance_to (position);

    if ((!i->is_empty ()) &&
        ((distance < ref_distance) || (ref_distance == 0.0)))
    {
      ref_distance = distance;
      selected = i;
    }
  }

  return *selected;
}

const Sensor& Cluster::closest (const Position& position) const
{
```

```
  return const_cast<Cluster*>(this)->closest (position);
}

Cluster::ComMap Cluster::map (const Position& position,
                              double agv_bandwidth)
{
  Cluster::ComMap selected; // Vector of sensors to connect

  m_act_bandwidth = selected.select (m_sensors, position);
  m_nb_act_sensors = static_cast<int> (selected.size ());
  m_scheduling = ((agv_bandwidth < m_act_bandwidth) &&
                  (m_nb_act_sensors > 0));

  if (m_scheduling)
  {
    for (ComMap::iterator i = selected.begin ();
         i != selected.end ();
         i++)
    {
      (*i)->rate (((*i)->rate () * agv_bandwidth) / m_act_bandwidth);
    }
  }

  return selected;
}

Cluster::ComMap Cluster::map_int (const Position& position,
                                  double agv_bandwidth)
{
  Cluster::ComMap selected_int; // Vector of sensors to connect

  m_act_bandwidth_int = selected_int.select (m_sensors_int, position);
  m_nb_act_sensors_int = static_cast<int> (selected_int.size ());
  m_scheduling_int = ((agv_bandwidth < m_act_bandwidth_int) &&
                      (m_nb_act_sensors_int > 0));

  if (m_scheduling_int)
  {
    for (ComMap::iterator i = selected_int.begin ();
         i != selected_int.end ();
         i++)
    {
      (*i)->rate (((*i)->rate () * agv_bandwidth) / m_act_bandwidth_int);
    }
  }

  return selected_int;
}

void Cluster::write (std::ostream& output) const
{
  output << "Cluster size: " << this->nb_sensors () <<
            std::endl <<
            "Nb of active sensors: " << m_nb_act_sensors <<
            std::endl <<
            "Cluster active bandwidth: " << m_act_bandwidth <<
            std::endl <<
            "Scheduling: " << (m_scheduling == 0 ? "false" : "true") <<
            std::endl;

  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
```

```
       i != m_sensors.end ();
       i++)
  {
    i->write_ln (output);
  }
}

void Cluster::write_accumulator_log (std::ostream& output) const
{
  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    i->write_accumulator_log (output);

    if (i != m_sensors.end ())
      output << " ";
  }
}

void Cluster::write_accumulator_log_ln (double t, std::ostream& output)
const
{
  output << t << " ";

  this->write_accumulator_log (output);

  output << std::endl;
}

void Cluster::write_rate_log (std::ostream& output) const
{
  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    i->write_rate_log (output);
    output << " ";
  }

  output << m_bandwidth;
}

void Cluster::write_rate_log_ln (double t, std::ostream& output) const
{
  output << t << " ";

  this->write_rate_log (output);

  output << std::endl;
}

void Cluster::write_map (std::ostream& output) const
{
  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    i->write_map (output);

    if (i != m_sensors.end ())
```

```
      output << std::endl;
  }
}

void Cluster::write_tag (std::ostream& output) const
{
  output << "timestamp ";

  for (std::vector<Sensor>::const_iterator i = m_sensors.begin ();
       i != m_sensors.end ();
       i++)
  {
    i->write_tag (output);
    output << " ";
  }

  output << m_tag << std::endl;
}

}
```

```
/*
 *  cluster.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_CLUSTER__
#define __H_CLUSTER__

#include <iostream>
#include <vector>

#include "object.hpp"
#include "accumulator.hpp"
#include "comrate.hpp"
#include "position.hpp"
#include "sensor.hpp"

namespace ComLibSim
{

class Cluster
{
private:
  std::vector<Sensor> m_sensors;
  std::vector<Sensor> m_sensors_int;  // Copy of sensors vector to
integrate
  std::string         m_tag;

  double m_bandwidth;

  int    m_nb_act_sensors; // Nb of sensors to connect (selected)
  double m_act_bandwidth;  // Bandwidth used by sensors (selected)
  bool   m_scheduling;     // Need or not scheduling

  // To integrate
  int    m_nb_act_sensors_int; // Nb of sensors to connect (selected)
  double m_act_bandwidth_int;  // Bandwidth used by sensors (selected)
  bool   m_scheduling_int;     // Need or not scheduling

public:
  class ComMap: public std::vector<std::vector<Sensor>::iterator>
  {
  public:
    double select (std::vector<Sensor>& sensors,
                   const Position& position);

    void write (std::ostream& output = std::cout) const;
  };

  Cluster (int nb = 0);
  virtual ~Cluster ();

  virtual int    nb_act_sensors () const;
```

```
  virtual double act_bandwidth  () const;
  virtual bool   scheduling     () const;
  virtual bool   is_empty       () const;
  virtual int    nb_sensors     () const;
  virtual double bandwidth      () const;

  virtual void tag (const std::string& tag);

  virtual void add (const Sensor& sensor);

  virtual void init_int ();
  virtual void copy_int ();

  virtual void get_data     (double *data) const;
  virtual void get_data_int (double *data) const;
  virtual void get_rate     (double *rate) const;
  virtual void get_rate_int (double *rate,
                             const double *position,
                             double agv_bandwidth);

  virtual void set_data     (double *data, double delta_time);
  virtual void set_data_int (const double *data);
  virtual void set_rate     (double *rate);

  virtual Sensor&       closest (const Position& position);
  virtual const Sensor& closest (const Position& position) const;

  virtual ComMap map     (const Position& position, double
agv_bandwidth);
  virtual ComMap map_int (const Position& position, double
agv_bandwidth);

  virtual void write                    (std::ostream& output =
std::cout) const;
  virtual void write_accumulator_log    (std::ostream& output =
std::cout) const;
  virtual void write_accumulator_log_ln (double t,
                                         std::ostream& output =
std::cout) const;
  virtual void write_rate_log           (std::ostream& output =
std::cout) const;
  virtual void write_rate_log_ln        (double t,
                                         std::ostream& output =
std::cout) const;
  virtual void write_map                (std::ostream& output =
std::cout) const;
  virtual void write_tag                (std::ostream& output =
std::cout) const;
};

}

#endif
```

```
#include "comlibsim.hpp"

#define SIMULATION_LOG_FILE  "log/simulation.log"
#define ACCUMULATOR_LOG_FILE "log/accumulator.log"
#define RATE_LOG_FILE        "log/rate.log"
#define SENSORS_MAP_FILE     "log/sensors.map"
#define PRINT_SIM_APP        "gnuplot"

const char *argp_program_version     = "ComLibSim v0.0";
const char *argp_program_bug_address = "<da.arada@gmail.com>";

static char doc[]       = "Communication Library Simulator";
static char args_doc [] = "<filename>.xml";

static struct argp_option options[] = {
  {"verbose",     'v', 0,        0, "Verbose output"},
  {"step",        's', "NUMBER", 0, "Simulation step size [0.1]"},
  {"time",        't', "NUMBER", 0, "Simulation time [500]"},
  {"simulation",  'l', "FILE",   0, "Simulation log file
[log/simulation.log]"},
  {"accumulator", 'a', "FILE",   0, "Accumulator log file
[log/accumulator.log]"},
  {"rate",        'r', "FILE",   0, "Rate log file [log/rate.log]"},
  {"sensors",     'e', "FILE",   0, "Sensors map file
[log/sensors.map]"},
  {"app",         'p', "FILE",   0, "App to print sim [gnuplot]"},
  {0}
};

static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  struct arguments *arguments = (struct arguments*) state->input;

  switch (key)
  {
    case 'v':
      arguments->verbose = 1;
      break;
    case 's':
      arguments->step_size = strtod (arg, NULL);
      break;
    case 't':
      arguments->sim_time = strtod (arg, NULL);
      break;
    case 'l':
      arguments->simulation_log_file = arg;
      break;
    case 'a':
      arguments->accumulator_log_file = arg;
      break;
    case 'r':
      arguments->rate_log_file = arg;
      break;
    case 'e':
      arguments->sensors_map_file = arg;
      break;
    case 'p':
      arguments->print_sim_app = arg;
      break;
    case ARGP_KEY_ARG:
```

```
      if (state->arg_num >= 1)
        argp_usage (state);
      arguments->cluster_xml_file[state->arg_num] = arg;
      break;
    case ARGP_KEY_END:
      if (state->arg_num < 1)
        argp_usage (state);
      break;
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

static struct argp argp = {options, parse_opt, args_doc, doc};

int sim_main(const struct arguments *arguments);

int
main (int argc, char **argv)
{
  int nb_sensors;
  int status;

  // Arguments management
  struct arguments arguments;

  // Default values
  arguments.verbose              = 0;
  arguments.step_size            = 0.1;
  arguments.sim_time             = 500;
  arguments.simulation_log_file  = (char *) SIMULATION_LOG_FILE;
  arguments.accumulator_log_file = (char *) ACCUMULATOR_LOG_FILE;
  arguments.rate_log_file        = (char *) RATE_LOG_FILE;
  arguments.sensors_map_file     = (char *) SENSORS_MAP_FILE;
  arguments.print_sim_app        = (char *) PRINT_SIM_APP;

  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  // Welcome message
  std::cout << doc << std::endl;
  std::cout << argp_program_version << std::endl;

  // Run simulation
  nb_sensors = sim_main (&arguments);

  // Print simulation log files
  if (strcmp (arguments.print_sim_app, (char *) PRINT_SIM_APP) == 0)
  {
    char gnuplot_script[50];

    sprintf (gnuplot_script,
             "%s -p -e \"NbSensors=%d\" gnuplot/map.p",
             arguments.print_sim_app,
             nb_sensors);
    status = system (gnuplot_script);
  }

  return status;
}
```

```
/*
 *  comlibsim.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_COMLIBSIM__
#define __H_COMLIBSIM__

#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <argp.h>

struct arguments
{
  char   *cluster_xml_file[1];
  int     verbose;
  double  step_size;
  double  sim_time;
  char   *simulation_log_file;
  char   *accumulator_log_file;
  char   *rate_log_file;
  char   *sensors_map_file;
  char   *print_sim_app;
};

#endif
```

```
/*
 *  comrate.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "comrate.hpp"

namespace ComLibSim
{

ComRate::ComRate ():
  m_reference (new Position()),
  m_max_rate (0.0),
  m_act_rate (0.0)
{
}

ComRate::ComRate (const Position& reference,
                  double max_rate,
                  double act_rate):
  m_reference (new Position (reference)),
  m_max_rate (max_rate),
  m_act_rate (act_rate)
{
}

ComRate::ComRate (const ComRate& com_rate):
  m_reference (com_rate.m_reference),
  m_max_rate (com_rate.m_max_rate),
  m_act_rate (com_rate.m_act_rate)
{
}

ComRate::~ComRate ()
{
}

Object* ComRate::object () const
{
  return new ComRate (*this);
}

bool ComRate::operator == (const ComRate& com_rate) const
{
  return m_max_rate == com_rate.m_max_rate;
}

bool ComRate::operator != (const ComRate& com_rate) const
{
  return ! operator == (com_rate);
}

bool ComRate::operator > (const ComRate& com_rate) const
```

```
{
  return m_max_rate > com_rate.m_max_rate;
}

bool ComRate::operator < (const ComRate& com_rate) const
{
  return m_max_rate < com_rate.m_max_rate;
}

bool ComRate::operator >= (const ComRate& com_rate) const
{
  return m_max_rate >= com_rate.m_max_rate;
}

bool ComRate::operator <= (const ComRate& com_rate) const
{
  return m_max_rate <= com_rate.m_max_rate;
}

void ComRate::set_act_rate (double act_rate)
{
  m_act_rate = act_rate;
}

double ComRate::get_max_rate () const
{
  return m_max_rate;
}

double ComRate::get_act_rate () const
{
  return m_act_rate;
}

double ComRate::rate_at (double distance) const
{
  if (distance < m_radius_low)
    return (- m_max_rate);

  if (distance < m_radius_high)
    return (- (m_max_rate * ::exp (1 - (distance / m_radius_low))));
//    return - m_max_rate * ::pow (10.0, -0.024 * distance);

  return 0.0;
}

double ComRate::rate_at (const Position& position) const
{
  double distance = m_reference->distance_to (position);

  return this->rate_at (distance);
}

void ComRate::write (std::ostream& output) const
{
  m_reference->write ();

  output << " Actual rate=" << this->get_act_rate () <<
            std::endl << "Rates: ";

  for (int i = 0; i < 100; i=i+20)
```

```
  {
    output << this->rate_at ((double) i) << " ";
  }
}

void ComRate::write_ln (std::ostream& output) const
{
  this->write (output);

  output << "END" << std::endl;
}

void ComRate::write_log (std::ostream& output) const
{
  output << std::fixed << std::setprecision (10);
  if (m_act_rate >= 0.0)
    output << m_act_rate;
  else
    output << 0.0;
}

std::ostream& operator << (std::ostream& output,
                           const ComRate& com_rate)
{
  com_rate.write (output);

  return output;
}

}
```

```
/*
 *  comrate.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_COMRATE__
#define __H_COMRATE__

#include <cmath>
#include <iostream>
#include <iomanip>

#include "object.hpp"
#include "position.hpp"

namespace ComLibSim
{

class ComRate: public Object
{
private:
  const Position*     m_reference;
  double              m_max_rate;
  double              m_act_rate;
  static const double m_radius_low   = 20.0;
  static const double m_radius_high  = 60.0;

public:
  ComRate ();
  ComRate (const Position& reference,
           double max_rate,
           double act_rate = 0);
  ComRate (const ComRate& com_rate);
  virtual ~ComRate ();

  virtual Object* object () const;

  virtual bool operator == (const ComRate& com_rate) const;
  virtual bool operator != (const ComRate& com_rate) const;
  virtual bool operator >  (const ComRate& com_rate) const;
  virtual bool operator <  (const ComRate& com_rate) const;
  virtual bool operator >= (const ComRate& com_rate) const;
  virtual bool operator <= (const ComRate& com_rate) const;

  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;

  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 ComRate& com_rate);
};

}

#endif
```

```
/*
 *  object.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_OBJECT__
#define __H_OBJECT__

namespace ComLibSim
{

class Object
{
public:
  virtual Object* object () const = 0;
  virtual ~Object () {};
};

}

#endif
```

```
/*
 *  parser.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "parser.hpp"

namespace ComLibSim
{

Parser::Parser (std::string filepath):
  m_filepath (filepath)
{
}

Parser::~Parser ()
{
}

int Parser::to_cluster (Cluster& cluster,
                        std::ostream& output) const
{

  static int nb_sensors_imported = 0;

  #ifdef LIBXMLCPP_EXCEPTIONS_ENABLE
  try
  {
  #endif
  xmlpp::DomParser parser (m_filepath, false);
  if (parser)
  {
    const xmlpp::Node* root_node = parser.get_document ()->get_root_node
();
    const xmlpp::Element* cluster_node =
                             dynamic_cast<const
xmlpp::Element*>(root_node);
    cluster.tag (cluster_node->get_attribute_value ("tag"));

    xmlpp::Node::NodeList sensors_list =
                             root_node->get_children ("Sensor");

    xmlpp::Node::NodeList::iterator sensor_iter;
    for (sensor_iter = sensors_list.begin ();
         sensor_iter != sensors_list.end ();
         sensor_iter++)
    {
      const xmlpp::Element* sensor_element =
                               dynamic_cast<const
xmlpp::Element*>(*sensor_iter);

      double x = Glib::Ascii::strtod (
                   sensor_element->get_attribute_value ("x"));
```

```
      double y = Glib::Ascii::strtod (
                  sensor_element->get_attribute_value ("y"));
      double max_rate = Glib::Ascii::strtod (
                           sensor_element->get_attribute_value
("max_rate"));
      double data = Glib::Ascii::strtod (
                      sensor_element->get_attribute_value ("data"));
      Glib::ustring tag = sensor_element->get_attribute_value ("tag");

      cluster.add (Sensor (Position (x, y), max_rate, data, tag));

      nb_sensors_imported++;
    }
  }
  #ifdef LIBXMLCPP_EXCEPTIONS_ENABLE
  }
  catch (const std::exception& ex)
  {
    std::cout << "Exception caught: " << ex.what () << std::endl;
    return -1;
  }
  #endif

  if (nb_sensors_imported > 0)
  {
    std::ostringstream total_sensors;
    total_sensors << nb_sensors_imported;
    this->write_ln ("Imported " +
                    total_sensors.str() +
                    " sensors from file \"" +
                    m_filepath + "\"", output);
  }
  else
    this->write_ln ("No sensors imported from file \"" +
                    m_filepath + "\"", output);

  return nb_sensors_imported;
}

void Parser::write (const Glib::ustring& text, std::ostream& output)
const
{
  output << text;
}

void Parser::write_ln (const Glib::ustring& text, std::ostream& output)
const
{
  this->write (text, output);
  output << std::endl;
}

}
```

```
/*
 *  parser.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_PARSER__
#define __H_PARSER__

#include <libxml++/libxml++.h>
#include <glibmm.h>
#include <iostream>
#include <fstream>
#include <sstream>

#include "cluster.hpp"

namespace ComLibSim
{

class Parser
{
private:
  std::string  m_filepath;

public:
  Parser (std::string filepath = "config/cluster.xml");
  virtual ~Parser ();

  virtual int to_cluster (Cluster& cluster,
                          std::ostream& output = std::cout) const;

  virtual void write        (const Glib::ustring& text,
                             std::ostream& output = std::cout) const;
  virtual void write_ln     (const Glib::ustring& text,
                             std::ostream& output = std::cout) const;
};

}

#endif
```

```
/*
 *  position.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "position.hpp"

namespace ComLibSim
{

Position::Position ():
  m_x(0.0),
  m_y(0.0)
{
}

Position::Position (double x, double y):
  m_x(x),
  m_y(y)
{
}

Position::Position (const Position& position):
  m_x(position.m_x),
  m_y(position.m_y)
{
}

Position::~Position ()
{
}

Object* Position::object () const
{
  return new Position (*this);
}

void Position::copy (const Position& position)
{
  if (this != &position)
  {
    m_x = position.m_x;
    m_y = position.m_y;
  }
}

Position& Position::operator = (const Position& position)
{
  this->copy (position);

  return *this;
}
```

```
bool Position::operator == (const Position& position) const
{
  return m_x == position.m_x && m_y == position.m_y;
}

bool Position::operator != (const Position& position) const
{
  return ! operator == (position);
}

double Position::get_x () const
{
  return m_x;
}

double Position::get_y () const
{
  return m_y;
}

void Position::get_xy (double *xy) const
{
  xy[0] = this->get_x ();
  xy[1] = this->get_y ();
}

double Position::distance_to (const Position& position) const
{
  return ::sqrt (::pow (m_x - position.m_x, 2) +
                 ::pow (m_y - position.m_y, 2));
}

void Position::write (std::ostream& output) const
{
  output << "(" << m_x << ", " << m_y << ")";
}

void Position::write_ln (std::ostream& output) const
{
  this->write (output);
  output << std::endl;
}

void Position::write_map (std::ostream& output) const
{
  output << m_x << " " << m_y;
}

std::ostream& operator << (std::ostream& output,
                           const Position& position)
{
  position.write (output);

  return output;
}

}
```

```
/*
 *  position.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_POSITION__
#define __H_POSITION__

#include <cmath>
#include <iostream>

#include "object.hpp"

namespace ComLibSim
{

class Position: public Object
{
private:
  double m_x;
  double m_y;

public:
  Position ();
  Position (double x, double y);
  Position (const Position& position);
  virtual ~Position ();

  virtual Object*   object     () const;
  virtual void      copy       (const Position& position);
  virtual Position& operator = (const Position& position);

  virtual bool operator == (const Position& position) const;
  virtual bool operator != (const Position& position) const;

  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;

  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);
};

}

#endif
```

```
/*
 *  sensor.cpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#include "sensor.hpp"

namespace ComLibSim
{

Sensor::Sensor (const Position& position,
                double max_rate,
                double data,
                const std::string& tag):
  m_accumulator(Accumulator (data)),
  m_position(position),
  m_com_rate(ComRate (position, max_rate)),
  m_tag(tag),
  m_rate(0.0)
{
}

Sensor::Sensor (const Sensor& sensor):
  m_accumulator(sensor.m_accumulator),
  m_position(sensor.m_position),
  m_com_rate(sensor.m_com_rate),
  m_tag(sensor.m_tag),
  m_rate(sensor.m_rate)
{
}

Sensor::~Sensor ()
{
}

Object* Sensor::object () const
{
  return new Sensor (*this);
}

bool Sensor::operator == (const Sensor& sensor) const
{
  return m_accumulator == sensor.m_accumulator &&
         m_position == sensor.m_position &&
         m_com_rate == sensor.m_com_rate &&
         m_tag == sensor.m_tag &&
         m_rate == sensor.m_rate;
}

bool Sensor::operator != (const Sensor& sensor) const
{
  return ! operator == (sensor);
}
```

```
void Sensor::data (double data, double delta_time)
{
  double delta_data;

  if (data < 0.0)
  {
    delta_data = m_accumulator.get_amount_data ();
    m_accumulator.set_amount_data (0.0);
  }
  else
  {
    delta_data = m_accumulator.get_amount_data () - data;
    m_accumulator.set_amount_data (data);
  }

  m_rate = delta_data / delta_time;
}

void Sensor::rate (double rate)
{
  m_com_rate.set_act_rate (rate);
}

void Sensor::get_xy (double *xy) const
{
  xy[0] = m_position.get_x ();
  xy[1] = m_position.get_y ();
}

bool Sensor::is_empty () const
{
  return m_accumulator.is_empty ();
}

double Sensor::data () const
{
  return m_accumulator.get_amount_data ();
}

double Sensor::max_rate () const
{
  return m_com_rate.get_max_rate ();
}

double Sensor::rate () const
{
  return m_com_rate.get_act_rate ();
}

double Sensor::bandwidth () const
{
  return m_rate;
}

double Sensor::rate_at (const Position& position) const
{
  return m_com_rate.rate_at (position);
}

double Sensor::distance_to (const Position& position) const
```

```
{
  return m_position.distance_to (position);
}

void Sensor::write (std::ostream& output) const
{
  output << "Sensor at " << m_position << " " <<
            m_accumulator << " " << m_com_rate;
}

void Sensor::write_ln (std::ostream& output) const
{
  this->write (output);

  output << std::endl;
}

void Sensor::write_accumulator_log (std::ostream& output) const
{
  m_accumulator.write_log (output);
}

void Sensor::write_rate_log (std::ostream& output) const
{
  output << m_rate;
}

void Sensor::write_map (std::ostream& output) const
{
  m_position.write_map (output);
  output << " " << m_tag;
}

void Sensor::write_tag (std::ostream& output) const
{
  output << m_tag;
}

std::ostream& operator << (std::ostream& output,
                           const Sensor& sensor)
{
  sensor.write (output);

  return output;
}

}
```

```
/*
 *  sensor.hpp
 *
 *  Copyright (C) 2011-2012  Daniel Sousa   <da.arada@gmail.com>
 *  Copyright (C) 2011-2012  Jorge Estrela  <jes@isep.ipp.pt>
 *
 *  Created by:
 *    Daniel Sousa  <da.arada@gmail.com>
 *
 *  Sponsor:
 *    Jorge Estrela  <jes@isep.ipp.pt>
 */

#ifndef __H_SENSOR__
#define __H_SENSOR__

#include <iostream>

#include "object.hpp"
#include "accumulator.hpp"
#include "comrate.hpp"
#include "position.hpp"

namespace ComLibSim
{

class Sensor: public Object
{
private:
  Accumulator m_accumulator;
  Position    m_position;
  ComRate     m_com_rate;
  std::string m_tag;
  double      m_rate;

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

  virtual Object* object () const;

  virtual bool operator == (const Sensor& sensor) const;
  virtual bool operator != (const Sensor& sensor) const;

  virtual void data (double data, double time = 0.0);
  virtual void rate (double rate);

  virtual void get_xy (double *xy) const;

  virtual bool   is_empty    () const;
  virtual double data        () const;
  virtual double max_rate    () const;
  virtual double rate        () const;
  virtual double bandwidth   () const;
  virtual double rate_at     (const Position& position) const;
  virtual double distance_to (const Position& position) const;
```

```
  virtual void write                (std::ostream& output = std::cout)
const;
  virtual void write_ln             (std::ostream& output = std::cout)
const;
  virtual void write_accumulator_log (std::ostream& output = std::cout)
const;
  virtual void write_rate_log       (std::ostream& output = std::cout)
const;
  virtual void write_map            (std::ostream& output = std::cout)
const;
  virtual void write_tag            (std::ostream& output = std::cout)
const;

  friend std::ostream& operator << (std::ostream& output,
                                    const Sensor& sensor);
};

}

#endif
```

Histórico

- 3 de Setembro de 2012, Versão 1.0, mailto:1000146@isep.ipp.pt
- 7de Setembro de 2012, Versão 2.0, mailto:1000146@isep.ipp.pt
- 7de Setembro de 2012, Versão 3.0, mailto:1000146@isep.ipp.pt

$Id: report_v1.doc v2.0.a Date:07-11-2012$