WiSE-MNet

Wireless Simulation Environment for Multimedia Networks

User's manual

(version 1.1)

Original version by Christian Nastasi (nastasichr@gmail.com)
Updated by Juan C. SanMiguel (juan.carlos.sanmiguel@qmul.ac.uk)

June 9, 2014

Contents

1 Introduction

The Wireless Simulation Environment for Multimedia Sensor Networks (WiSE-MNet) has been designed to simulate distributed algorithms for Wireless Multimedia Sensor Networks (WMSNs) under realistic network conditions. The simulation environment is based on one of the most popular network simulator: *OMNeT++* . Among the several simulation models for the OMNeT++ environment, *Castalia* is the one that has been designed with similar goals, although it focuses on classic Wireless Sensor Networks (WSNs).

WiSE-MNet is proposed as an extension of the Castalia/OMNeT++ simulator. The main extensions provided to the Castalia simulation model can be summarized as:

- generalization of the sensor data-type (from scalar-based to any type) in Section 3.1;
- idealistic communication and direct application communication in Section 3.3;
- concrete modules for sensing and processing: moving target, camera modeling, target tracking application in Sections 3.2 and 3.4.
- simple GUI for 2D world representation in Section 3.5;

We assume the reader to be familiar with the OMNeT++ environment and to know the basics about the Castalia simulation model. The reference versions over OMNeT++ and Castalia are respectively the 4.4.1 and the 3.1 . Documentations and tutorials about OMNeT++ can be found at the project documentation page `http://www.omnetpp.org/documentation`. We particularly suggest to read the User Manual first and then to use the API Reference when developing new modules. A copy of the Castalia user's manual is available at `http://castalia.npc.nicta.com.au/documentation.php`.

2 Installation

WiSE-MNet is based on OMNeT++ and is an extension of the Castalia simulation model. WiSE-MNet has been developed using the version 4.4.1 of OMNeT++ and the 3.1 version of Castalia. Although OMNeT++ is available for Windows systems, Castalia has been designed for *GNU/Linux*-like systems (see Castalia reference manual). For this reason we strongly recommend to use a *GNU/Linux*-like system to use WiSE-MNet (the Ubuntu GNU/Linux distribution has been successfully used). However, installation for Windows systems might be possible through the *Cygwin* environment, although this has not been tested.

2.1 Prerequisites

WiSE-MNet requires to install OMNeT++ and OpenCV. For this tutorial, Ubuntu 12.04 LTS is used and the Ubuntu-specific command are indicated. Please refer to the Omnet or OpenCV installation guides for other operative systems (OS). Please launch a *bash* shell and type the following commands:

1. Before installing please make sure that the system is updated and upgraded (*Ubuntu-specific* command)

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

2. We assume to work in the home directory

```
$ cd ~
```

Installing OMNeT++

For full instructions and details about the installation of OMNeT++ , refer to the Linux section of the OMNeT++ installation guide (also available at `http://www.omnetpp.org/pmwiki/index.php?n=Main.InstallingOnUnix`). The following steps should be performed for a fresh installation of OMNeT++ . In this guide, we used the version 4.4.1 which is available at `http://www.omnetpp.org/omnetpp/doc_details/2272-omnet-441-source--ide-tgz`.

1. Install required dependencies for OMNeT++ . (*Ubuntu-specific* command)

```
$ sudo apt-get install build-essential gcc g++ bison flex perl tcl-dev tk-dev blt
    libxml2-dev zliblg-dev openjdj-6-jre doxygen graphviz openmpi-bin libopenmpi-dev
    libpcap-dev
```

2. Get the OMNeT++ sources from the download page or type.

```
$ wget -O omnetpp-4.4.1-src.tgz http://www.omnetpp.org/omnetpp/doc_download/2272-
    omnet-441-source--ide-tgz
```

   A file `omnetpp-4.4.1-src.tgz` will be created.

3. Extract the source files

```
$ tar xvzf omnetpp-4.4.1-src.tgz
```

   A folder `omnetpp-4.4.1` will be created.

4. Set the environment variables to point to the OMNeT++ binary paths:

```
$ export PATH=$PATH:~/omnetpp-4.4.1/bin
$ export LD_LIBRARY_PATH=~/omnetpp-4.4.1/lib
```

   These two lines should be also appended to the `~/.bashrc` file.

5. Compile OMNeT++ [1]

```
$ cd omnetpp-4.4.1
$ . setenv
$ ./configure
$ make
```

6. OMNeT++ should be successfully installed. The following command can be used to verify that the OMNeT++ executables are in the execution path.

```
$ which opp_makmake
```

7. Testing the installation

```
$ cd samples/dyna
$ ./dyna
```

   After clicking through some options, you should see the output depicted in Figure 1.

[1] NOTE: if you have a multi-core machine, compilation will be faster by running the make command with the '-j' option and passing the number of cores plus one as argument. For instance, in a dual-core machine use '`make -j 3`'.

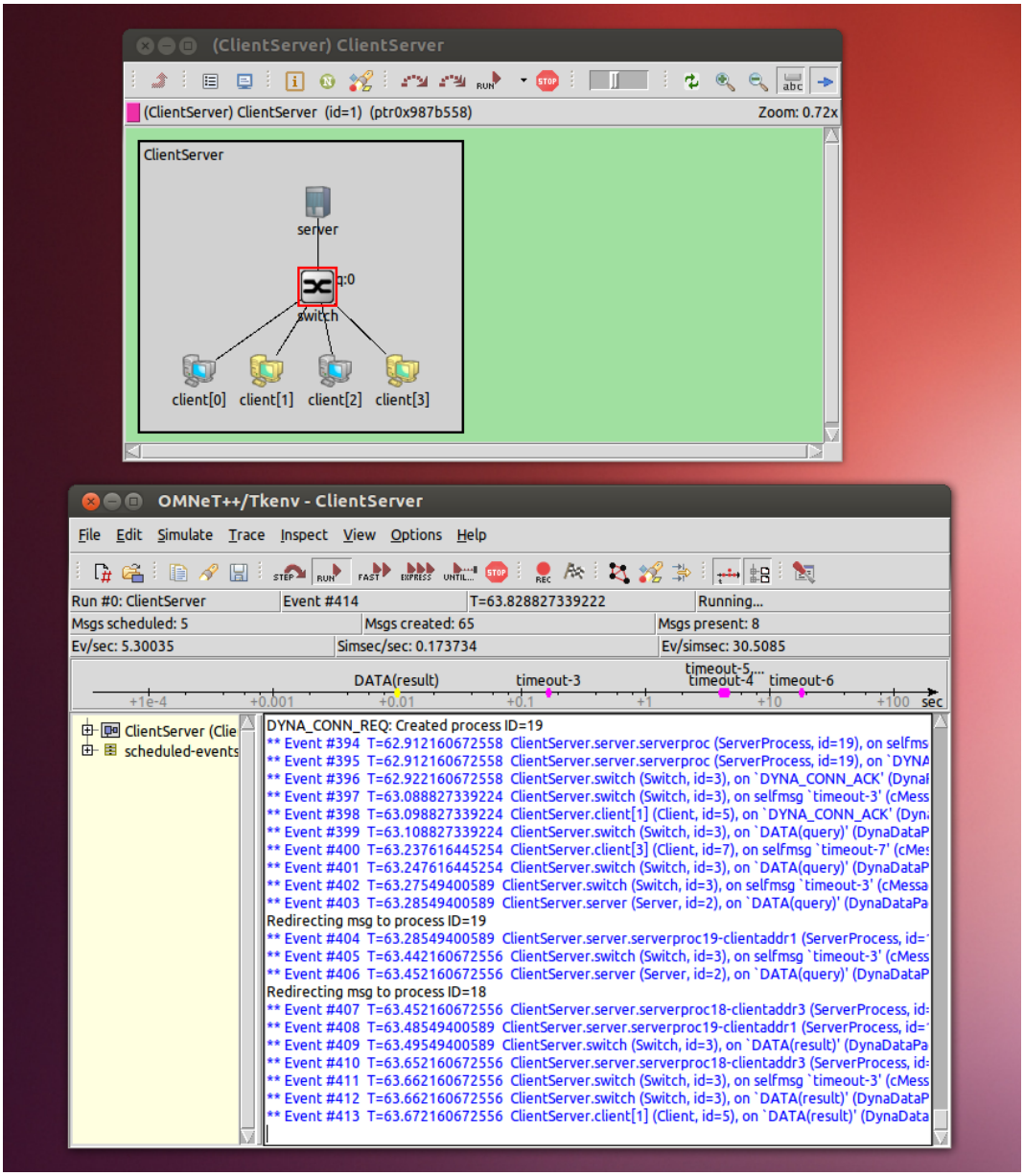


Figure 1: Output of the *dyna* application of OMNeT++ after a successful install.

Installing OpenCV

For installation instruction of the OpenCV library, please refer to `http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html`. The following steps should be performed for a fresh installation of OpenCV after OMNeT++ . In this guide, we used the version 2.4.1 which is available at `http://downloads.sourceforge.net/project/opencvlibrary/opencv-unix/2.4.1/OpenCV-2.4.1.tar.bz2`.

This tutorial is based on `https://help.ubuntu.com/community/OpenCV`.

1. Install required dependencies for OpenCV. (*Ubuntu-specific* command)

```
$ sudo apt-get install build-essential libgtk2.0-dev libjpeg-dev libtiff4-dev
    libjasper-dev libopenexr-dev cmake python-dev python-numpy python-tk libtbb-dev
```

```
    libeigen2-dev yasm libfaac-dev libopencore-amrnb-dev libopencore-amrwb-dev
    libtheora-dev libvorbis-dev libxvidcore-dev libx264-dev libqt4-dev libqt4-opengl-
    dev sphinx-common texlive-latex-extra libv4l-dev libdc1394-22-dev libavcodec-dev
    libavformat-dev libswscale-dev
```

2. Get the OpenCV sources from the download page or type.

```
$ cd ~
$ wget http://downloads.sourceforge.net/project/opencvlibrary/opencv-unix/2.4.1/
    OpenCV-2.4.1.tar.bz2
```

3. Extract the source files

```
$ tar -xvf OpenCV-2.4.1.tar.bz2
```

A folder `OpenCV-2.4.1` will be created.

4. Generate the Makefile to compile OpenCV

```
$ mkdir build
$ cd build
$ cmake -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON -D WITH_V4L=ON -D
    INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLES=ON -D
    WITH_QT=ON -D WITH_OPENGL=ON ..
```

A folder `build` will be created.

5. Compile OpenCV sources

```
$ make
$ sudo make install
```

6. Configure OpenCV (*Ubuntu-specific* command) by editing the configuration file

```
$ sudo gedit /etc/ld.so.conf.d/opencv.conf
```

Add the following line `/usr/local/lib` at the end of the file (it may be an empty file).

7. Configure OpenCV (*Ubuntu-specific* command)

```
$ sudo ldconfig
$ sudo gedit /etc/bash.bashrc
```

Add the following lines `PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig` and `export PKG_CONFIG_PATH` at the end of the file.

8. To check whether the OpenCV library are correctly installed

```
$ pkg-config opencv --cflags
$ pkg-config opencv --libs
```

These should print the include paths, compiler and linker options required to build the WiSE-MNet with the OpenCV library.

9. Testing the installation

```
$ cd ~/OpenCV-2.4.1/samples/c
$ chmod +x build_all.sh
$ ./build_all.sh
$ ./facedetect --cascade="/usr/local/share/OpenCV/haarcascades/
    haarcascade_frontalface_alt.xml" --scale=1.5 lena.jpg
```

After compiling and running the example, you should see the output depicted in Figure 2.

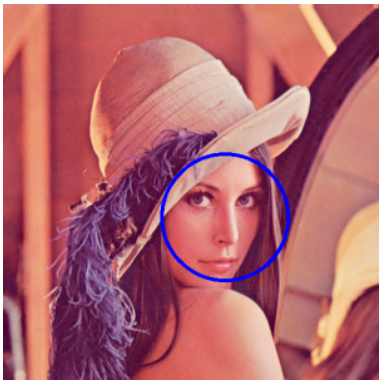


Figure 2: Output of the *facedetect* application of OpenCV 2.4.1 after a successful install.

2.2 Installing WiSE-MNet

The simulator source files are distributed as an extension of the Castalia simulation model. The steps required to compile the simulator are equivalent to those for Castalia. We describe two methods using the terminal and the OMNeT++ IDE.

From terminal

We assume to work in the home directory (type '`cd ~`' to enter it) in a *bash* shell. The following procedure creates an executable file located in `WiSE-MNet-v1.1/out/gcc-release/wise-mnet` which can be also accessed via the symbolic link `wise-mnet` located in the directory `WiSE-MNet-v1.1` .

1. Extract the source files

```
$ tar xvzf WiSE-MNet-v1.1.tar.gz
$ cd WiSE-MNet-v1.1
```

   A folder `WiSE-MNet-v1.1` will be created.

2. Set the environment variables to point to symbolic link `wise-mnet` of the WiSE-MNet executable:

```
$ export PATH=$PATH:~/WiSE-MNet-v1.1
```

   This line should be also appended to the `~/.bashrc` file.

3. Create the makefiles to compile Castalia with the WiSE-MNet extensions

```
$ ./makemake
```

4. Build [2]

```
$ make
```

   A symbolic link `wise-mnet` will be created in the directory `WiSE-MNet-v1.1` after the successful build.

5. To properly clean the last WiSE-MNet build, the following can be used

```
$ ./makeclean
```

6. Testing the installation

```
$ cd ~/Wise-MNet-v1.1/wise/simulations/wiseCamera_test_FOV_2D/
$ wise-mnet -c General
```

   After compiling and running the example, you should see the output depicted in Figure 3.

[2] NOTE: if you have a multi-core machine, compilation will be faster by running the make command with the '-j' option and passing the number of cores plus one as argument. For instance, in a dual-core machine use '`make -j 3`'.

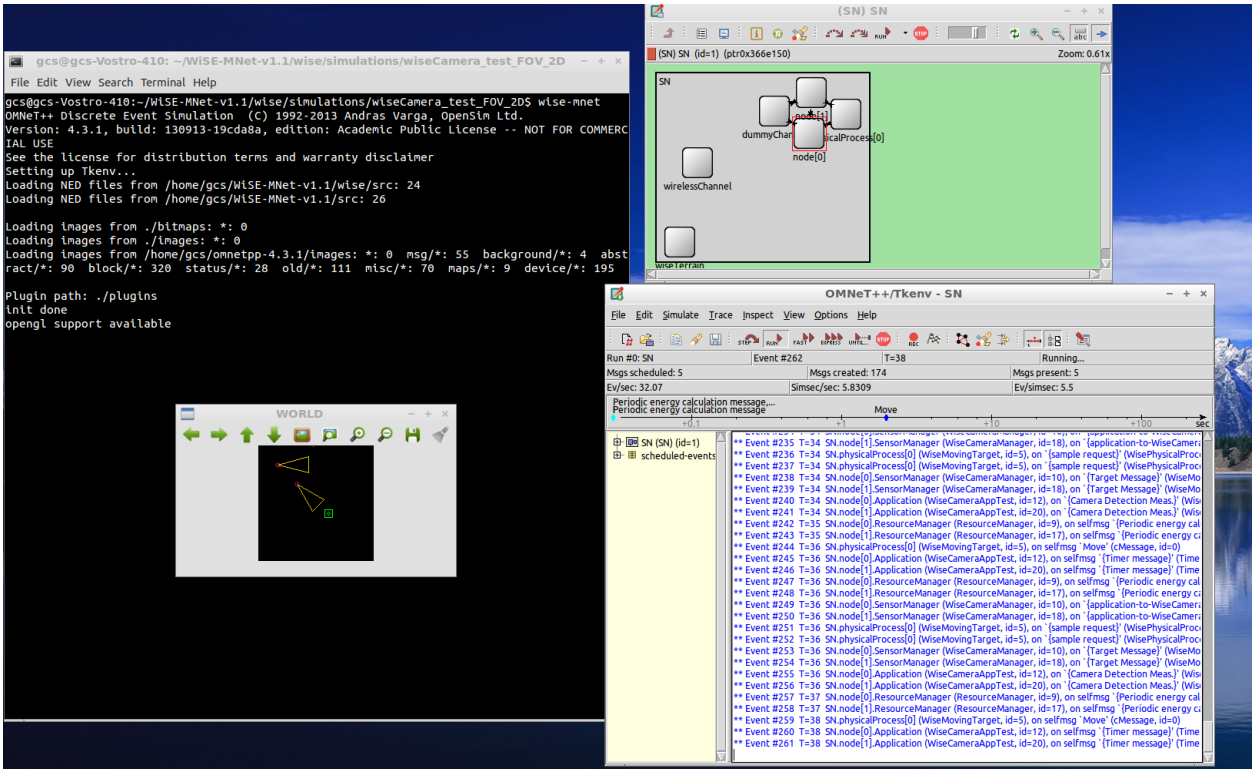


Figure 3: Example of succesffull execution of WiSE-MNet .

From OMNeT++ IDE

The OMNeT++ IDE can be also used to develop applications for WiSE-MNet . A quick overview of the IDE is available at `htpp://www.omnetpp.org/doc/omnetpp/IDE-Overview.pdf`.

1. Extract the source files

```
$ tar xvzf WiSE-MNet-v1.1.tar.gz
$ cd WiSE-MNet-v1.1
```

A folder `WiSE-MNet-v1.1` will be created.

2. Start the IDE

```
$ omnetpp
```

If the command does not work or there no symbolic link to the OMNeT++ executable, please go to the directory `~/omnetpp-4.4.1/ide` and type again the command. As a result, you should see the IDE environment as illustrated in Figure 4.

3. Create and configure the WiSE-MNet project. A video tutorial has been created at `http://www.eecs.qmul.ac.uk/~andrea/wise-mnet.html`. Here we summarize the main steps.

   (a) Create a new OMNeT++ empty project (*File→New→OMNeT++ project*)

   (b) Import all source files into the newly created project (*"Right Click" on the project name→Import*)

   (c) Go to the source NED files manager (*"Right Click" on the project name→Properties→*OMNeT++ *→*NED Source Folder) and tick two "src" boxes in the paths `WiSE-MNet-v1.1/src` and `WiSE-MNet-v1.1/wise/src`.

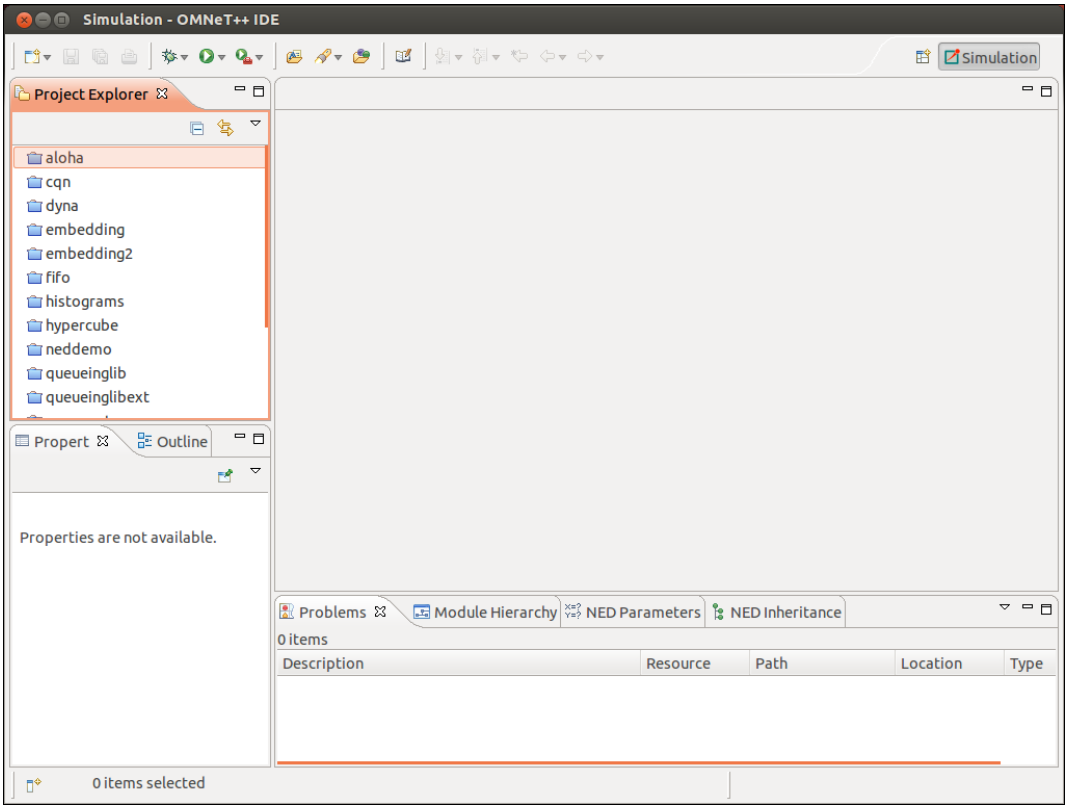


Figure 4: OMNeT++ IDE main window..

(d) Add required include OpenCV files. Go to the "includes" tab in the "Path and Symbols" menu (*"Right Click" on the project name→Properties→*C/C++ General→Path and Symbols). Then, add the paths `/usr/local/include/opencv` and `/usr/local/include/opencv2` in "GNU C" and "GNU C++". These paths depend on the OpenCV installation and can be obtained by running:

```
$ pkg-config opencv --cflags
```

(e) Add required libraries files for OpenCV. Go to *the Makemake* options (*"Right Click" on the project name-→Properties→*OMNeT++ →Makemake). Then, click on `makemake (deep, recurse->)` and the button on the left will be enable. After clicking on this button, go to the link tab and include the Opencv libraries `opencv_core, opencv_features2d opencv_imgproc opencv_highgui opencv_video opencv_legacy` in the "Additional libraries box". In the "Additional objects" box, include the path `wise/src/wise/utils/gmm/c-gmm/c-gmm-64bit.a` The final result can be checked in the "Preview" tab which should similar to the following:

```
--deep -O out -lopencv_core -lopencv_features2d -lopencv_imgproc -lopencv_highgui
    -lopencv_video -lopencv_legacy --meta:recurse --meta:auto-include-path --meta:
    export-library --meta:use-exported-libs --meta:feature-cflags --meta:feature-
    ldflags -- wise/src/wise/utils/gmm/c-gmm/c-gmm-64bit.a
```

(f) Finally, run the desired simulation by creating a configuration of the selected *.ini file (*"Right Click" on the project name-→Run or Debug as→*OMNeT++ *simulation*)

2.3 Software organization

The WiSE-MNet root directory (according to installation instructions is `~/WiSE-MNet-v1.1/`) contains the original Castalia source files and the extensions provided by WiSE-MNet . All WiSE-MNet files used to define/redefine modules and to run simulations are contained in the `wise/` folder in the root directory. The native Castalia files can still be found in their original position (`src/`, and `Simulations/`). In this section, we give an overview of the software organization in folders to help the reader browsing the source code.

The structure of the WiSE-MNet root directory is:

`bin/`	native Castalia python scripts
`out/`	output generated after compilation of WiSE-MNet files
`src/`	native Castalia NED/C++ sources
`doc/`	documentation of WiSE-MNet files
`Simulation/`	native Castalia simulation setups
`wise/`	WiSE-MNet NED/C++ sources and Simulation setups
`makemake`	Script to configure the WiSE-MNet makefiles
`makeclean`	Script to properly clean-up the WiSE-MNet build
`...`	others

The WiSE-MNet simulation setup files are contained in the `wise/Simulations/` sub-folder. The definitions/redefinitions of the OMNeT++ modules (NED/C++ files) can be found in the `wise/src/` sub-folder, and in particular the `wise/src/wise/` contains the main part of the software.

The structure of the `wise/src/wise/` subtree is the following:

`wise/src/wise/node/`	Definition of the node's components
`/world/`	Definition of the world's elements (PhysicalProcess, terrain)
`/wirelessChannel/`	WirelessChannel and WiseDummyWirelssChannel
`/gui/`	simple GUI code
`/utils/`	Utilities (GMM, ParticleFilter, helper classes)

The structure of the `wise/src/wise/node` subtree is the following:

`node/sensorManager/`	Sensor Manager modules
`/wiseEmptySensorManager`	Dummy sensor producing random numbers
`/wiseCameraManager`	Camera Manager module
`...`	
`node/application/`	Application modules
`/wiseCameraApplication/`	WiseCameraApplication base class
`/wiseCameraSimplePeriodicTracker/`	WiseCameraSimplePeriodicTracker base class
`/wiseAppTest/`	Example WiseAppTest module
`/wiseCamera_test_FOV_2D/`	Example wiseCamera_test_FOV_2D module
`/wiseCameraAppTest/`	Example WiseCameraAppTest module
`/wiseCameraTrackerTest/`	Example WiseCameraTrackerTest module
`/wiseCameraMultiVideo/`	Example WiseCameraMultiVideo module
`/wiseCameraDPF/`	WiseCameraDPF tracker
`/wiseCameraKCF/`	WiseCameraKCF tracker
`/wiseCameraICF/`	WiseCameraICF tracker
`/wiseCameraICF-NN/`	WiseCameraICF-NN tracker
`...`	

2.4 Documentation

Doxygen style documentation has been created in HTML format for WiSE-MNet . This documentation is available at `doc/html/index.html` and Figure 2.4 depicts an example for the module *WiseCameraICF*. Additionally, a copy of the user manuals for Castalia, OpenCV and OMNeT++ is included in the directory `doc`.

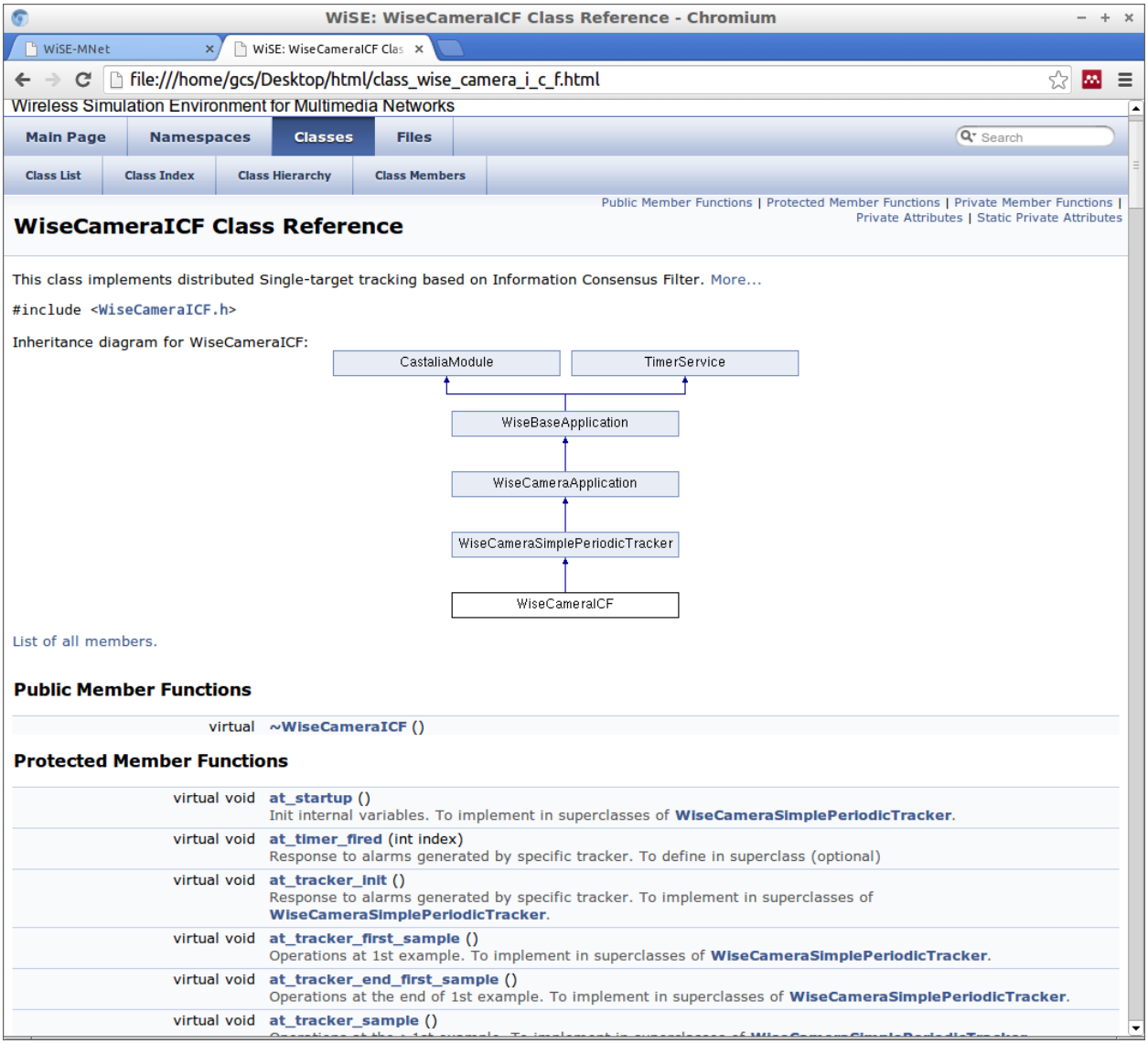


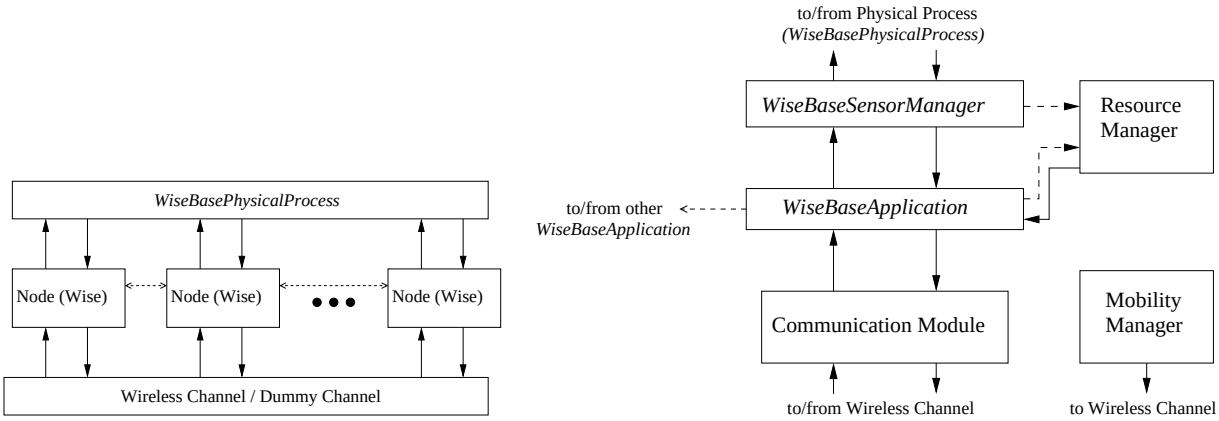
Figure 5: Example of documentation for module *WiseCameraICF* of WiSE-MNet .

2.5 Compatibility issues

WiSE-MNet has been successfully tested in various Linux systems: Ubuntu 12.04 LTS, Debian 7.5 and Fedora 20-17. However, a compatibility issue has been detected for Fedora 17 regarding the drivers for the video graphics card and the use of OpenCV within OMNeT++ , reporting the error *X Error: BadWindow (invalid Window parameter)*.

3 Overview

In this section, we present an overview of WiSE-MNet . The overall structure of the network and node model is depicted in Figure 6. The structure of the node model contains the same modules as the Castalia simulator (sensor, application, resource, communication and mobility). However, WiSE-MNet extends Castalia's



(a) Network Model

(b) Node (Wise) Model

Figure 6: WiSE-MNet network and node model overview

```
module Node {

 parameters:  //basic parameters
        double xCoor = default (0);
        double yCoor = default (0);
        //...

 gates: //connections from/to other modules
        output toWirelessChannel;
        input fromWirelessChannel;

        output toPhysicalProcess[];
        input fromPhysicalProcess[];

 submodules: //submodules of the node
        Communication: node.communication.CommunicationModule;
        MobilityManager: <MobilityManagerName> like node.mobilityManager.iMobilityManager;
        ResourceManager: node.resourceManager.ResourceManager;
        SensorManager: <SensorManagerName> like wise.node.sensorManager.WiseBaseSensorManager

 connections allowunconnected:  //connections between node and submodules
        Communication.toNodeContainerModule --> toWirelessChannel
        fromWirelessChannel --> Communication.fromNodeContainerModule
        Application.toSensorDeviceManager --> SensorManager.fromApplicationModule;
        Communication.toApplicationModule --> Application.fromCommunicationModule
        SensorManager.toApplicationModule --> Application.fromSensorDeviceManager;
        //...

        ResourceManager.toSensorDevManager --> SensorManager.fromResourceManager;
        //...
}
```

Figure 7: Selected content of file **`wise/src/node/node.ned`** .

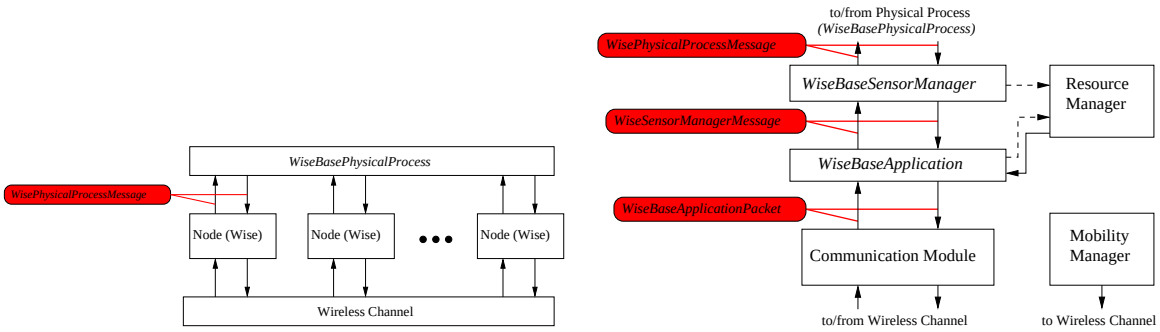
modules to provide functionalities for multimedia networks.

This node structure is described via the NED file **`node.ned`** available at the directory **`wise/src/node`** . The description contains the parameters, the submodules, the gates to define connections from/to other modules and the specific connections between modules. Figure 7 shows an extract of the file **`node.ned.`**

3.1 Generalized sensor data-types

The generalization of the sensor data-types is obtained by defining an abstract class *WisePhysicalProcessMessage* that has to be derived to define any type of physical process information. Accordingly, other abstract classes have been modified to redefine some of the original Castalia modules (see Figure 8). In particular:

- *WiseBasePhysicalProcess*, *WiseBaseSensorManager* and *WiseBaseApplication* that redefine respectively the base classes for the physical process, the sensor manager and the application layer;
- *WisePhysicalProcessMessage*, *WiseSensorManagerMessage* and *WiseApplicationPacket* information exchange classes.



(a) Network Model (b) Node (Wise) Model

Figure 8: Generic data-types (in red colour).

In the current distribution of WiSE-MNet , we included some concrete classes that have been used to simulate distributed target tracking algorithms in simplified context. Figure 9 depicts the block diagram of the modifications performed over the generalized sensor data-types. In the following sections, we describe them focusing on the sensing, communication and processing operations.

3.2 Sensing

3.2.1 *WiseMovingTarget*

This module extends the *WiseBasePhysicalProcess* base class/module to implement a moving target in a 2D ground plane. Targets are currently represented as (bounding) boxes and can move according to different types of motion: linear, circular, linear-circular and random. This configuration of the 2D target behavior can be established in the `omnetpp.ini` file (settings) of the defined simulation. An example is provided in Figure 10.

3.2.2 *WiseVideoFile*

This module extends the *WiseBasePhysicalProcess* base class/module to implement the capture process of a live video stream via files stored. The path to the video file must be defined in the *.ini file of the simulation

3.2.3 *WiseCameraManager*

This module extends the *WiseBaseSensorManager* module that implements the sensing logic of the node's camera. The module is strongly related to the type of physical process we are using. The *WiseCameraManager* has been designed to support different types of sensing through the *WiseCameraHandler* mechanism, allowing the user to easily add different camera models (e.g. projection models). We currently support only the *WiseCameraDetections* model, which is a simplified projection model that assumes a top-down facing camera observing targets modeled according to the *WiseMovingTarget* module.

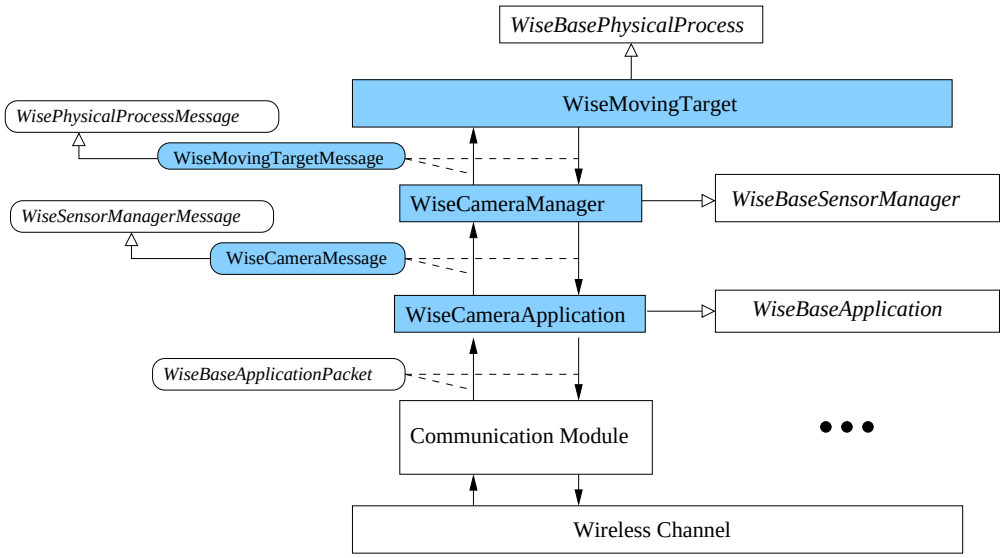


Figure 9: WiSE-MNet concrete modules (in blue those modified to support the new datatypes).

```
# ==========================================
# Physical Process (environment's events)
# ==========================================
SN.wiseTerrain.gui = "opencv"
#SN.wiseTerrain.gui_wait_start = true
SN.physicalProcessName = "WiseMovingTarget"

SN.physicalProcess[**].bb_width = 10
SN.physicalProcess[**].bb_height = 10
SN.physicalProcess[**].update_time = 2
SN.physicalProcess[**].noise = 0.5

SN.physicalProcess[0].x_init = 80
SN.physicalProcess[0].y_init = 60
SN.physicalProcess[0].move_type = "lin_random"
SN.physicalProcess[1].x_init = 12.00
SN.physicalProcess[1].y_init = 60.00
SN.physicalProcess[1].move_type = "lin_noise"
#SN.physicalProcess[2].x_init = 103.00
#SN.physicalProcess[2].y_init = 103.00
#SN.physicalProcess[2].move_type = "lincirc_noise"
```

Figure 10: Configuration of *WiseMovingTarget* in the `omnetpp.ini` file of the simulation.

```
cplusplus {{
        #include "WiseApplicationPacket_m.h"
        #include "WiseDefinitionsTracking.h"
        #include <opencv.hpp>
}};

class WiseApplicationPacket;
class noncobject cv::Mat;

packet WiseCameraICFMsg extends WiseApplicationPacket {

        unsigned long trackingCount;
        unsigned long iterationStep;
        unsigned int targetID;
        unsigned int TypeNeighbour;

        cv::Mat matrix; // OpenCV matrix
}
```

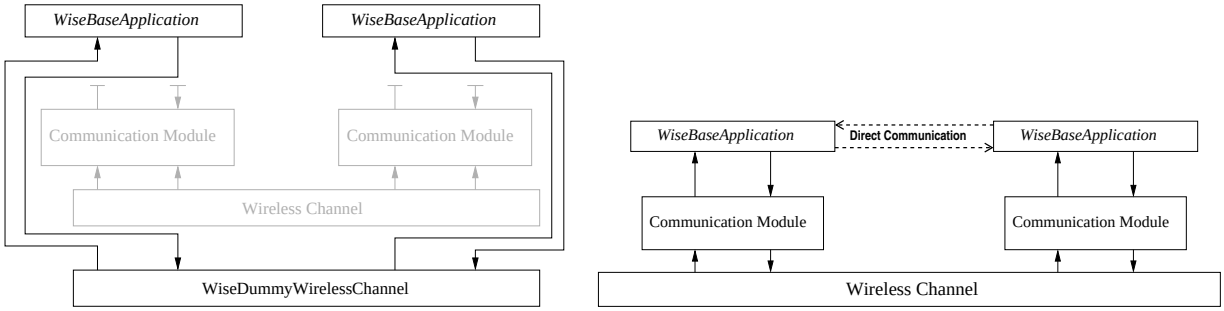
Figure 11: Example of a packet format (msg file) in WiSE-MNet .

3.3 Communication

In WiSE-MNet , communication is done via packets whose format is encoded in *.msg files. These packets depend on the developed application (e.g. the tracking algorithm) and contain all the variables and data to be exchanged among nodes. *Note that OMNeT++ automatically generates two files *_m.cc and *_m.h for every defined packet when the compilation of the project starts.* These two files should not be modified. Figure 11 shows an example for defining a packet.

3.3.1 Idealistic communication mechanisms

There are two "idealistic" communication mechanisms that have been introduced: the *WiseDummyWirelessChannel* and the *DirectApplicationMessage*. The first one changes the network properties (to idealistic) seemingly from the application point of view, the second one is rather a "magic" direct information exchange channel. Figure 12 represents the two mechanisms.



(a) (b)

Figure 12: Idealistic communication through (a) *WiseDummyWirelessChannel* and (b) *DirectApplicationMessage*.

The *WiseDummyWirelessChannel* is a module that is used to bypass the Castalia communication stack and wireless channel. This module allows to specify the node neighborhood and performs idealistic communication with no-delay or packet loss/corruption. The module is to be used alternatively to the *WirelessChannel* module proposed in Castalia. The interaction between application and communication module does not change, the user can decide whether to change the network capability in the simulation configuration, without changing the application logic. The *WiseDummyWirelessChannel* communication is selected in the configuration file of the simulation (*.ini file). Figure 13 presents an example of this configuration.

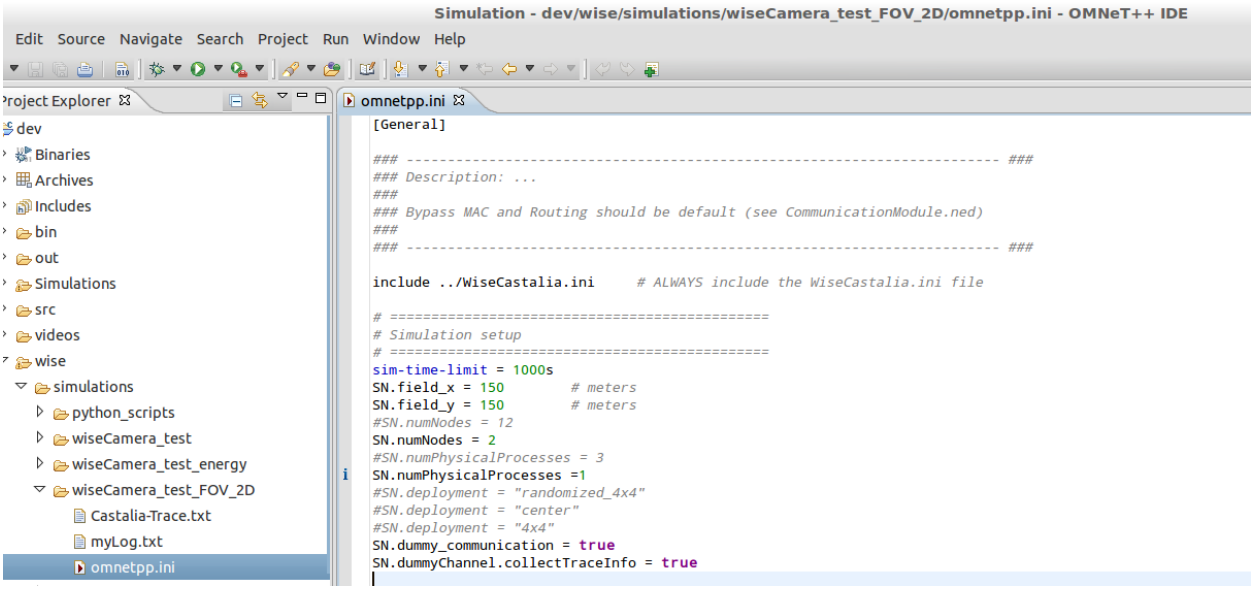


Figure 13: Content of a `omnetpp.ini` file to select the dummy communication protocol.

The second mechanism provided for idealistic communication is the *DirectApplicationMessage*. This is an OMNeT++ communication port that has been added to the application module so that two nodes' application layers can interact directly without bypassing the communication modules. With this mechanism, a part of the application can use a realistic (or idealistic) network communication (through either the *WirelessChannel* or the *WiseDummyWirelessChannel*), while some part might assume ideal node-to-node interaction.

3.3.2 Communication with other nodes

For communicating with other nodes, two mechanisms are provided for direct and neighbor data exchange.

Single nodes

For direct communication via the network or *DirectApplicationMessages*, the base class *WiseBaseApplication* provides the required functionality via the functions `toNetworkLayer` and `sendDirectApplicationMessage`. Both functions require to define the destination node and a packet (*cPacket* or *WiseApplicationPacket*) which contains the information to exchange. Figure 14 provides the location and example of the function `toNetworkLayer`.

Neighbors in the communication or vision graph

Additionally, default communication with all neighbor nodes is also provided in WiSE-MNet . Two types are supported to communicate with nodes in the vision and communication graph. To determine the vision graph (i.e. cameras sharing the Field of View), an initialization phase is performed via *DirectApplicationMessages*. For the communication graph, nodes exchange test packets within their communication range to gather information about the number of nodes that can be reached. This functionality is provided in the *WiSE-CameraSimplePeriodicTracker* via the functions `send_messageNeighboursCOM` and `send_messageNeighboursFOV` (see Figure 15).

```
class WiseBaseApplication: public CastaliaModule, public TimerService {
private:
        simtime_t initialize_time;   //!< First time of module initialization (in secs of the sim)
        bool first_initialize;       //!< FLAG to indicate initialization

protected:
        /*--- The .ned file's parameters ---*/
        //!< Type of application currently running in the node. Must be defined as a superclass of WiseBaseApplication
        string applicationID;
        ...

protected:
        WiseBaseApplication() : first_initialize(true) {} ;
        ...

        // FUNCTIONS TO SEND PACKETS TO OTHER MODULES (sensor, network, nodes)
        //!< Sends a request (packet) to SensorManager to sample data
        void requestSensorReading(int index = 0);
         //!< Sends a message to the entire network without destination (should not APPLICATION_PACKET type as it needs destination)
        void toNetworkLayer(cMessage *);
        //!< Sends a packet to the network with a specific delay (received by the CommunicationModule and delivered to all nodes)
        void toNetworkLayer(cPacket *, const char *, double delay=0);
        //!< Sends a packet to the network using direct node-to-node links (not using the CommunicationModule)
        void sendDirectApplicationMessage(WiseApplicationPacket*, const char*, unsigned type=WISE_APP_DIRECT_COMMUNICATION);
```

(a)

```
WiseApplicationPacket *m = new WiseApplicationPacket("Neighbor Discover", APPLICATION_PACKET);
m->setType(WISE_APP_NEIGHBOUR_DISCOVER_REQUEST);
toNetworkLayer(m, BROADCAST_NETWORK_ADDRESS, startup_delay);
BASE_TRACE << "SEND: 'Neighbor Discover Request' (broadcast) with delay=" << startup_delay;
```

(b)

Figure 14: (a) Location of functions for node-to-node communication in *WiseBaseApplication* class and (b) usage example of **`toNetworkLayer`** function.

3.4 Processing

The processing is performed in the application layer of the node. Therefore, each node can implement a different application layer or all the nodes can have the same processing routines. Note that this processing layer does not only correspond to tracking algorithms as other distributed algorithms can be implemented.

The selection of the application layer is done in the **`omnet.ini`** file and its configurations depends on the defined parameters for the layer. Figure 16 shows an example for the *WiseCameraICF* application.

3.4.1 Types

The processing in WiSE-MNet can be performed via two mechanisms:

- On demand via **`fromNetworkLayer`** function. This mechanism correspond to replies to received messages from other network nodes.
- Periodically via timers using the **`timerFiredCallback`** function. This function corresponds to repetitive tasks that the node has to perform (e.g. grab a video frame and analyze its content every second). The timer type and alarm period have to be defined.

Figure 17 provides an example of these two functionalities.

3.4.2 Application Layer classes

The application module contains the algorithm of the distributed application. The user should typically provide its own application module to implement a new distributed algorithm. The application module, derived from *WiseBaseApplication*, interacts with a *WiseBaseSensorManager* and the Castalia communication module in order to realize the logic of the distributed algorithm. In the current distribution of WiSE-MNet , we provided some application-layer classes according to the hierarchy shown in Figure 18.

```
/*! \class WiseCameraSimplePeriodicTracker
 *  \brief This class implements the template for periodic distributed target tracking.
 *
 *  More detailed description to be provided...
 */
class WiseCameraSimplePeriodicTracker : public WiseCameraApplication {

private:
        ...

protected:
        void startup();
        void finishSpecific();
        void fromNetworkLayer(WiseApplicationPacket *, const char *, double, double);
        void handleSensorReading(WiseCameraMessage *);
        void timerFiredCallback(int index);

        void send_message(WiseApplicationPacket*);
        void send_message(WiseApplicationPacket*, const std::string&);

        int send_messageNeighboursCOM(WiseApplicationPacket*);
        int send_messageNeighboursFOV(WiseApplicationPacket*);
```

Figure 15: Functions for graph-based communication in *WiseCameraSimplePeriodicTracker*.

```
# ============================================
# Applications
# ============================================
SN.node[**].ApplicationName = "WiseCameraICF"
SN.node[**].Application.collectTraceInfo = true
SN.node[**].Application.showCamImage = false
SN.node[**].Application.sampling_time = 10

# if set to 'false', the processing starts at time=0 (as discovering FOV neighbours is done instantly)
# if set to 'true', there is delay to discover the Comms neighbours (different for each node)
SN.node[**].Application.neighbourDiscover = false
SN.node[**].Application.neighbourDiscoverFOV = true
SN.node[**].Application.procNoiseCov = 10
#SN.node[**].Application.processNoiseCov = 5
SN.node[**].Application.measNoiseCov = 10
#SN.node[**].Application.alpha = 0.005
SN.node[**].Application.iter_max = 5
#SN.node[**].Application.iter_max = ${0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
SN.node[**].Application.alpha = 0.65
```

Figure 16: Configuration of the application layer for *WiseCameraICF* (`omnetpp.ini` file).

```
void WiseAppTest::fromNetworkLayer(WiseApplicationPacket * rcvPacket,
                                   const char *src, double rssi, double lqi)
{
        // Function called when a packet is received from the network
        // layer of the communication module
        LOGGER << "WiseAppTest::fromNetworkLayer() called" << endl;

        // Print some packet info: sender ID, RSSI, LQI, payload(hex)
        LOGGER << "\tRx from" << string(src) << " with rssi=" << rssi <<
                " lqi=" << lqi << endl << "\tPayload[] = " << hex;
        for (unsigned c = 0; c < 100; c++)
                logger << (unsigned int) rcvPacket->getPayload(c) << " ";
        logger << dec << endl;
        // Calculate application-to-application communication delay and print it
        ApplicationInteractionControl_type ctl = rcvPacket->getApplicationInteractionControl();
        double l = 1000 * SIMTIME_DBL(simTime() - ctl.timestamp);
        LOGGER << "\t app2app delay = " << l << endl;
}
```

(a)

```
void WiseAppTest::timerFiredCallback(int index)
{
        // Called when an alarm expires:
        LOGGER << "WiseAppTest::timerFiredCallback() called";
        switch (index) {
        case ALARM_SENSOR_SAMPLE:// alarm was for sensor reading:
                // query the sensor manager a new sample (image)
                requestSensorReading();//call the sensor reading function
                break;
        case ALARM_SEND_PACKET:// alarm was a send packet: create a simple packet of 19200
                // bytes, put some payload and broadcast it.
                WiseApplicationPacket * pkt = new WiseApplicationPacket("Test Pkt",APPLICATION_PACKET);
                // set packet details
                // ...
                toNetworkLayer(pkt, BROADCAST_NETWORK_ADDRESS); //send a message to network
                break;
        default:
                // unexpected alarm ID: generate and error
                opp_error("WiseAppTest::timerFiredCallback(): BAD index");
        }
}
```

(b)

Figure 17: Processing via (a) `fromNetworkLayer` function for network response and (b) `timerFiredCallback` function for periodic processing.

WiseBaseApplication This is the base class for any application module in WiSE-MNet . The class provides a mechanism to automatically calculate the set of radio neighbor nodes (a set of nodes that can be reached by a give node with a single-hop wireless communication). The application *WiseAppTest* shows an example of a class derived directly from this class.

WiseCameraApplication This is a base class to derive from when we are interested in creating an application module that uses a *WiseCameraManager*. During the startup phase this class will query the *WiseCameraManager* to collect information about the camera (e.g. FOV) and to create a list of overlapping-FOV neighbor nodes. The node's and other nodes' camera information are available as protected member respectively called `camera_info` and `overlapping_fov_cameras`. The application *WiseCameraAppTest* shows an example of a class derived from *WiseCameraApplication*.

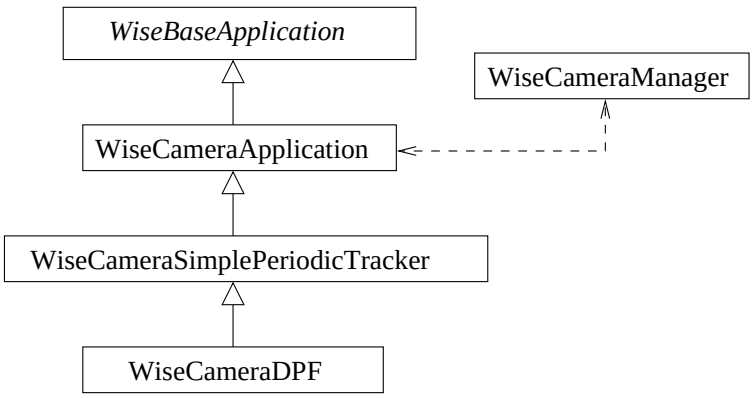


Figure 18: Application class hierarchy

WiseCameraSimplePeriodicTracker This class is derived from *WiseCameraApplication* and is meant to be used as a base class for periodic tracking algorithms based on *WiseCameraApplication*. This class defines a set of callback-like functions that will be called at different steps of any periodic tracking algorithm. The class defines some functions that will be called at startup (for initializations) and other functions that will be periodically called when a new image is available. The application *WiseCameraTrackerTest* is a basic example derived from *WiseCameraSimplePeriodicTracker*.

WiseCameraDPF It is a *WiseCameraSimplePeriodicTracker* that implements a distributed particle filter algorithm. The algorithm uses a sequential aggregation mechanism, exchanging the partial posterior approximated with Gaussian Mixture Models. For more details the reader can refer to [3].

WiseCameraKCF It is a WiseCameraSimplePeriodicTracker that implements a distributed Kalman filter algorithm. The algorithm uses a consensus mechanism, exchanging the final state among camera neighbors. For more details the reader can refer to [4]

WiseCameraICF It is a WiseCameraSimplePeriodicTracker that implements a distributed Kalman filter algorithm via its equivalent information matrix formulation. The algorithm uses a consensus mechanism, exchanging the weighted final state among camera neighbors (information vector and matrix). For more details the reader can refer to [1]

WiseCameraICF-NN It is a WiseCameraSimplePeriodicTracker that extends **WiseCameraICF** for multiple targets. For the association stage, the algorithm uses a nearest-neighbor approach so tracks at one time-step are linked with following one. For more details the reader can refer to [2]

3.5 Visualization

3.5.1 Simple GUI

We included a simple GUI that can be useful for testing and evaluation of distributed algorithm for WMSNs. We currently used the GUI for a simple representation of a 2D-world (ground plane) where targets and sensor-cameras can be displayed during the simulation (see Figure 19). The GUI could be further used to evaluate distributed algorithms involving computer-vision processing.

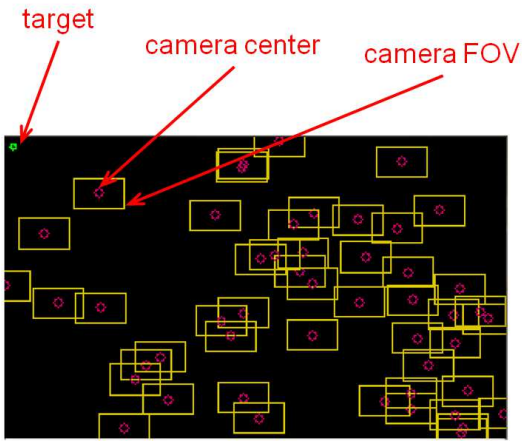


Figure 19: Simple GUI for 2D-world scenarios

4 Application Examples

4.1 Test applications

4.1.1 WiseAppTest

The module is derived directly from the *WiseBaseApplication* base class. This is the simplest application example and shows how to use the three basic elements of any distributed application for WMSNs: sensor reading, network communication and time-triggered actions.

Source files: `WiseAppTest.ned`
`WiseAppTest.h`
`WiseAppTest.cc`

Run the example To try this example the simulation setup `wise/Simulations/WiseSimpleApp_test/omentpp.ini` should be used. We assume to work in the home directory (type '`cd ~`' to enter it) in a *bash* shell and to have WiSE-MNet properly installed.

1. Enter the simulation directory:

```
$ cd WiSE-MNet-v1.1/wise/Simulations/WiseSimpleApp_test/
```

2. Run the simulation using (WiSE-MNet extended) Castalia:

```
$ wise-mnet -c General
    Running configuration 1/1
```

A `myLog.txt` will be created containing the application printouts.

4.1.2 WiseCameraAppTest

The module is derived from the WiseCameraApplication class. This example module is similar to the WiseAppTest one. It shows a basic interaction with the WiseCameraManager (a camera-based sensor manager that produces target detections). The example shows also a custom application message exchanged among the nodes.

Source files: `WiseCameraAppTest.ned`
`WiseCameraAppTest.h`
`WiseCameraAppTest.cc`
`WiseCameraAppTestPacket.msg`

Run the example

To try this example the simulation setup `wise/Simulations/WiseCamera_test/omentpp.ini` should be used. We assume to work in the home directory (type '`cd ~`' to enter it) in a *bash* shell and to have WiSE-MNet properly installed.

1. Enter the simulation directory:

```
$ cd WiSE-MNet-v1.1/wise/Simulations/WiseCamera_test/
```

2. Run the simulation using (WiSE-MNet extended) Castalia:

```
$ wise-mnet -c General
    Running configuration 1/1
```

   A "WORLD" window will pop-up showing a simple 2D representation of the ground-plane world, the cameras and the targets.

3. Press a button (on the WORLD window) to start the simulation. This will show an animation of the three targets moving with different types of motion on the ground plane.

4. When the simulation is over, a `myLog.txt` will be created containing the application printouts.

4.1.3 WiseCameraTrackerTest

The module is derived from the WiseCameraSimplePeriodicTracker class. This example shows how the skeleton of a distributed target tracking application looks like when using the WiseCameraManager (producing target detections) and adopting a classic periodic tracker approach.

Source files: `WiseCameraTrackerTest.ned`
`WiseCameraTrackerTest.h`
`WiseCameraTrackerTest.cc`
`WiseCameraTrackerTestMessage.msg`

Run the example

To try this example the simulation setup `wise/Simulations/WiseTracker_test/omentpp.ini` should be used. We assume to work in the home directory (type '`cd ~`' to enter it) in a *bash* shell and to have WiSE-MNet properly installed.

1. Enter the simulation directory:

```
$ cd WiSE-MNet-v1.1/wise/Simulations/WiseTracker_test/
```

2. Run the simulation using (WiSE-MNet extended) Castalia:

```
$ wise-mnet -c General
    Running configuration 1/1
```

   A "WORLD" window will pop-up showing a simple 2D representation of the ground-plane world, the cameras and the targets.

3. Press a button (on the WORLD window) to start the simulation. This will show an animation of two targets moving with different types of motion on the ground plane with 4 camera nodes (3 with partially overlapping FOV).

4. When the simulation is over, a `myLog.txt` will be created containing the application printouts.

4.2 Single target tracking

4.2.1 WiseCameraKCF

This module implements a Distributed Kalman Filter (KCF) tracker based on a consensus mechanism to exchange the Posterior (see Section 3).

Source files:
- `WiseCameraKCF.ned`
- `WiseCameraKCF.h`
- `WiseCameraKCF.cc`
- `WiseCameraKCF_utils.h`
- `WiseCameraKCFMsg.msg`

Run the example(s) A single simulation setups is provided to test this algorithm.

- `wise/Simulations/WiseCameraKCF/omnetpp.ini`:
  shows an example with various camera nodes with partially overlapping FOV and two targets moving inside a fully-overlapping region (all the cameras observing the target).

4.2.2 WiseCameraICF

This module implements a Distributed Kalman Filter (ICF) tracker based on a consensus mechanism to exchange the Posterior and Observed data (see Section 3).

Source files:
- `WiseCameraICF.ned`
- `WiseCameraICF.h`
- `WiseCameraICF.cc`
- `WiseCameraICF_utils.h`
- `WiseCameraICFMsg.msg`

Run the example(s) A single simulation setups is provided to test this algorithm.

- `wise/Simulations/WiseCameraICF/omnetpp.ini`:
  shows an example with various camera nodes with partially overlapping FOV and two targets moving inside a fully-overlapping region (all the cameras observing the target).

4.2.3 WiseCameraDPF

This module implements a Distributed Particle Filter (DPF) tracker based on a sequential aggregation mechanism to exchange the (Partial) Posterior (see Section 3).

Source files:
- `WiseCameraDPF.ned`
- `WiseCameraDPF.h`
- `WiseCameraDPF.cc`
- `WiseCameraDPFMessage.msg`
- `WiseCameraDPFMessage_custom.h`

Run the example(s) Three different simulation setups have been provided to test this algorithm.

- `wise/Simulations/WiseCameraDPF_example1/omnetpp.ini`:
  shows an example with 4 camera nodes with partially overlapping FOV and two targets moving inside a fully-overlapping region (all the cameras observing the target).
- `wise/Simulations/WiseCameraDPF_example2/omnetpp.ini`:
  4 cameras with partially- and non-overlapping FOV and a single target moving inside and outside the FOVs.

- `wise/Simulations/WiseCameraDPF_example3/omnetpp.ini`:
  20 cameras and a single moving target.

This simulation folders contain a `Makefile` to run and clean the simulation output. To run the simulation enter the setup directory and type '`make`'. Several files will be created after the simulation (with GUI animation). The files '`dpf_results.txt`' and '`dpf_part_results.txt`' contain information respectively about the tracking output and the intermediate tracking steps. To clean-up the simulation folder, type '`make clean`'.

4.3 Multiple target tracking

4.3.1 WiseCameraICF-NN

This module implements a Distributed Kalman Filter (ICF-NN) tracker for multiple targets based on a consensus mechanism to exchange the Posterior and Observed data (see Section 3). Multi-target association is done via Nearest Neighbor algorithm.

Source files:
`WiseCameraICF-NN.ned`
`WiseCameraICF-NN.h`
`WiseCameraICF-NN.cc`
`WiseCameraICF-NN_utils.h`
`WiseCameraICF-NNMsg.msg`

Run the example(s) A single simulation setups is provided to test this algorithm.

- `wise/Simulations/WiseCameraICF-NN/omnetpp.ini`:
  shows an example with various camera nodes with partially overlapping FOV and two targets moving inside a fully-overlapping region (all the cameras observing the target).

5 Developing your own application

For creating a new application, the *WiseCameraSimplePeriodicTracker* base class is provided which extends WiseBaseApplication. This class contains basic functions to initialize resources, send/receive messages from/to network (or direct node-to-node communication) and handling of control messages. Prior to process packets, this class discovers the communication graph for each node (i.e. neighbors nodes using network communication).

The new distributed algorithm has to extend the *WiseCameraSimplePeriodicTracker* in order to use the provided functionality. Please check the already defined classes *WiseCameraICF* or *WiseCameraKCF* for examples of applications developed based on *WiseCameraSimplePeriodicTracker*.

5.1 Required files

At least, one file of the following types is required:

Source files:	`WiseCameraXXXX.ned`	Description of your application using NED language
	`WiseCameraXXXX.h`	Include file with the header of the application
	`WiseCameraXXXX.cc`	Source file with the code of the application
	`WiseCameraXXXMsg.msg`	Packet definition to exchange among nodes
Simulation files	`omnetXXX.ini`	Configuration of the simulation for the new application

5.2 Steps

The steps for developing a new application are:

```
#include "WiseCameraSimplePeriodicTracker.h"
#include "WiseCameraICFMsg_m.h"
#include "WiseDefinitionsTracking.h" //include for definitions of states and measurements
#include "WiseCameraICF_utils.h" //include specific-structures for single-target tracking of ICF

#define MAX_SIZE_BUFFER 10

/*! \class WiseCameraICF
 *  \brief This class implements distributed Single-target tracking based on ICF
 */
class WiseCameraICF : public WiseCameraSimplePeriodicTracker
{

private:
    // Define variables
    // ...

protected:
    // Functions to be implemented from WiseCameraSimplePeriodicTracker class
    virtual void at_startup();                      //!< Init internal variables.
    virtual void at_timer_fired(int index) {} ;     //!< Response to alarms generated by specific tracker.
    virtual void at_tracker_init();                 //!< Init resources.
    virtual void at_tracker_first_sample();        //!< Operations at 1st example.
    virtual void at_tracker_end_first_sample();   //!< Operations at the end of 1st example.
    virtual void at_tracker_sample();               //!< Operations at the >1st example.
    virtual void at_tracker_end_sample();          //!< Operations at the end of >1st example.
```

Functions to implement from tracking template

Figure 20: Functions to be implement for developing a new application.

1. Define the new application as a new class (extending *WiseBaseApplication* for generic processing or *WiseCameraSimplePeriodicTracker* for a distributed tracker).
2. Create the structures and classes to be used within your new application.
3. Implement the functions `startup`sand `finishSpecific` for the new application requirements.
4. Implement the function `handleSensorReading` to handle the data provided by the *SensorManager*.
5. Implement the functions `fromNetworkLayer` and `handleDirectApplicationMessage` to handle received packets from, respectively the network and direct node-to-node communication.
6. Functions `handleNetworkControlMessage`, `handleMacControlMessage` and `handleRadioControlMessage` are optional.
7. Implement the logic of your application via the processing functions (see Figure 20).

Note that the development of new application layers do not need to modify the sensing (*WiseMovingTarget*, *WiseVideoFile* and *WiseCameraManager*) and communication (*WiseDummyWirelessChannel* and *WirelessChannel*) modules.

References

[1] J. Farrell A. Kamal and A. Roy-Chowdhury. Information weighted consensus filters and their application in distributed camera networks. *IEEE Transactions on Automatic Control*, 58(12):3112–3125, Dec 2013.

[2] A. Kamal, J. Farrell, and A. Roy-chowdhury. Information consensus for distributed multi-target tracking. In *Proc. of the IEEE Int. Conf. on Computer Vision and Pattern Recognition*, pages 2403–2410, Portland (USA), 25-27 Jun. 2013.

[3] C. Nastasi and A. Cavallaro. Distributed target tracking under realistic network conditions. In *Proc. of Sensor Signal Processing for Defence (SSPD)*, pages 1–5, London (UK), 28-29 Sept. 2011.

[4] R. Olfati-Saber. Distributed kalman filtering for sensor networks. In *Proc. of the IEEE Int. Conf. on Decision and Control*, pages 5492–5498, San Diego (USA), 12-15 Dec. 2007.