UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

University of Verona

A.A 2017/2018

Laboratory of Networked Embedded Systems

Lesson 1

SystemC Network Simulation Library (SCNSL)

Davide Quaglia, Valentina Ceoletta, Enrico Fraccaroli

May 7, 2018

Contents

Chapter 1

Introduction

1.1 Network Simulation

Network simulation allows to reproduce the behavior of both computational and communication aspects of a network, modeling packet-based networks such as Ethernet, wireless LAN and field bus.

1.2 SystemC Network Simulation Library

SystemC Network Simulation Library (SCNSL) is an extension of SystemC to allow modeling packet-based networks such as wireless networks, Ethernet, and fieldbus. As done by basic SystemC for signals on the bus, SCNSL provides primitives to model packet transmission, reception, contention on the channel and wireless path loss. The use of SCNSL together with SystemC allows the easy and complete modeling of distributed applications of networked embedded systems such as wireless sensor networks, routers, and distributed plant controllers.

1.3 SCNSL components

1.3.1 Task

Tasks contain the application blocks interacting with the network, that is the system functionality which is under development. Tasks shall be implemented by designers either at RTL or TLM level. From the point of view of the network simulator, a task is just the producer or consumer of packets and therefore its implementation is not important. For the system designer, task

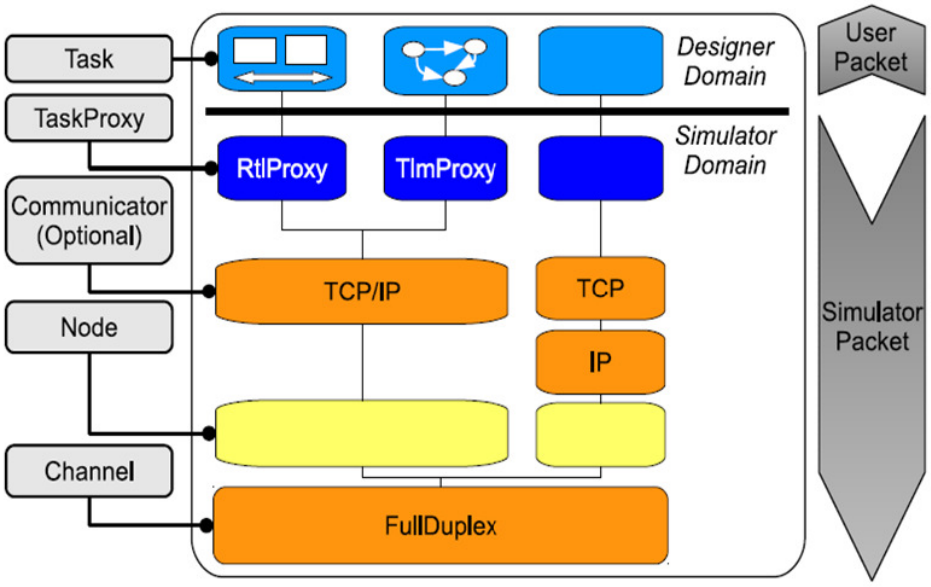


Figure 1.1: SCNSL components

implementation is crucial and many operations are connected to its modeling (*i.e.*, change of abstraction level, validation, fault injection, HW/SW partitioning, mapping to an available platform, synthesis and so forth).

1.3.2 Task Proxy

Task proxy acts as an intermediate layer between designer's domain and simulator domain. Each Task instance is connected to one or more TaskProxy instances and, from the perspective of the network simulation kernel, the TaskProxy instance is the alter-ego of the task. Viceversa, from the point of view of the application, each TaskProxy can represent a sort of socket interface, since it provides the interfaces for network communication.

1.3.3 Communicator

Communicator instances are created by SCNSL developers to modify simulation behavior. For example, they can be used to implement queues and protocols. Their presence is not mandatory.

1.3.4 Node

Nodes are abstraction of physical devices. Tasks are hosted by Nodes. Tasks deployed on different nodes shall communicate by using the API provided by SCNSL for the network communication, while tasks deployed on the same node shall communicate by using standard SystemC communication primitives.

1.3.5 Channel

Channels are models the physical transmission medium. Point-to-point and shared (multi-point) channels are available.

1.3.6 Environment

Environment represents some properties of the environment that contains nodes. It provides functions to get informations related to the transmissions of packets (*e.g.*, delay, attenuation, error rate, *etc.*).

Chapter 2

Installation and Setup

2.1 Requirements

The installation process requires the following items

- Linux 32/64bit
- SystemC 2.3.2 with C++17 option
- cmake
- A C++ compiler and a linker
- Doxygen, for the documentation
- Latex, for the documentation

2.1.1 General important remarks

This document will help you to install SystemC 2.3.2, SCNSL and exercises, in that exact temporal sequence. If something goes wrong during each of the three steps, please delete and re-create the corresponding "build/" folder to avoid that partial wrong configuration files are used.

Furthermore, you have to compile SystemC 2.3.2 with C++17 option which is not the default setup for SystemC. Therefore, it is recommended to

- recompile and re-install it with the following instructions
- remove previous SystemC (and SCNSL of course) info in INCLUDE, PATH and LD_LIBRARY_PATH environment variables

2.2 Directory structure

Let us assume that you are in your HOME directory which has the following directory structure:

```
HOME
  |-- Software
  \-- systemc-2.3.2.tar.gz
```

Create a bash script to export the environment variables in your current shell environment:

```
gedit setup-env.sh
chmod +x setup-env.sh
```

and add the following lines:

- Export the root directory of SystemC.

```
export SYSTEMC_HOME=${HOME}/Software/systemc
```

- Add its include directory to the PATH variable

```
export INCLUDE=${SYSTEMC_HOME}/include
```

- Add the path to the library directory to PATH

```
export PATH=${PATH}:${SYSTEMC_HOME}/lib
```

- Add it also to the variable `LD_LIBRARY_PATH`

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${SYSTEMC_HOME}/lib
```

- Export the root directory where the scnsl library will be installed

```
export SCNSL_HOME=${HOME}/Software/scnsl-stable-linux-x86_64
```

- Add to PATH its include directory

```
export PATH=${PATH}:${SCNSL_HOME}/include
```

- Add to `LD_LIBRARY_PATH` its library directory

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${SCNSL_HOME}/lib
```

2.3 SystemC Installation

SystemC can be downloaded at

```
http://www.accellera.org/downloads/standards/systemc
```

Then, download the last release. For the rest of the procedure we will use the names and version of tools up to the date of May 7, 2018. You should see the version **SystemC 2.3.2 (Includes TLM)**, and the link **Core SystemC Language and Examples**. This will download a compressed file which contains the source codes of SystemC. Then in the Downloads folder, unpack the compressed file:

```
tar xvf systemc-2.3.2.tar.gz
```

Move inside the folder:

```
cd systemc-2.3.2
```

Create and move inside a build directory:

```
mkdir build
cd build
```

Run cmake in order to generate the Makefile:

```
cmake -DCMAKE_CXX_STANDARD=17 -DCMAKE_INSTALL_PREFIX=<path> ..
```

In this case replace `<path>` with the path where you want to install SystemC:

```
<path> = $HOME/Software/systemc
```

Afterwards, compile and install it:

```
make install
```

2.4 SCNSL Installation

SCNSL is available to be downloaded at:

```
https://sourceforge.net/projects/scnsl/
```

If you have installed the **Git** version control system, you can get the most updated version of the library, directly from the repository:

```
cd $HOME
git clone git://git.code.sf.net/p/scnsl/git-code scnsl
```

At this point, run the script written before in order to export the environment variables (please, note the dot and space before the path of the script):

```
. ./setup-env.sh
```

Move inside the scnsl directory:

```
cd scnsl
```

Create and move inside a build directory:

```
mkdir build
cd build
```

Run cmake in order to generate the Makefile

```
cmake ..
```

Compile the library

```
make
```

Then compile all the tests

```
make tests
```

Install the library

```
make install
```

Copy the compiled library inside the Software folder on your root

```
mv scnsl-stable-linux-x86_64/ ${HOME}/Software/
```

2.4.1 LaTeX and Doxygen fix

If LaTex and Doxygen and not installed and you do not want to use them, you can eliminate CMake-level check by opening a specific CMake script executed during the generation of the Makefile, that is

```
gedit ../scripts/FindScnsl.cmake
```

and removing

```
find_package(EdalabLatex    REQUIRED)
find_package(EdalabDoxygen  REQUIRED)
```

Chapter 3

Network Scenario Creation

3.1 Creation Steps

The steps required to create a network scenario with SCNSL are the following:

1. Instantiation of the **SCNSL Simulator**.
2. Instantiation of the **Environment**.
3. Instantiation of the physical **Nodes**.
4. Instantiation of the physical **Channels**.
5. **Binding** of nodes to channels, and setting of nodes' properties.
6. Instantiation of the **Tasks**.
7. Instantiation of communicators (optional).
8. Binding of tasks, communicators (optional) and channels.
9. Setting of tracing features.
10. Creation of custom tasks.

3.1.1 Instantiation of the SCNSL Simulator

It is important, first of all, to create an instance of the SCNSL Simulator; the instance is a singleton and provides the methods for creating the scenario components. Instantiate the simulator as follows:

```
Scnsl::Setup::Scnsl_t * sim = Scnsl::Setup::Scnsl_t::get_instance();
```

3.1.2 Instantiation of the Environment

This object can be used to model, manage and get some properties related to the environment.

```
Scnsl::Utils::DefaultEnvironment_t::createInstance(ALPHA_VALUE);
```

3.1.3 Instantiation of the physical Nodes

A node can be created with the following code:

```
Scnsl::Core::Node_t * NODE_NAME = sim->createNode();
```

3.1.4 Instantiation of the physical Channels

A channel can be created and set as follows:

```
CoreChannelSetup_t CHANNEL_SETUP;

CHANNEL_SETUP.name = "full_duplex_channel";
CHANNEL_SETUP.extensionId = "core";
CHANNEL_SETUP.channel_type(CoreChannelSetup_t::FULL_DUPLEX);
CHANNEL_SETUP.capacity = 1000;
CHANNEL_SETUP.capacity2 = 1000;
CHANNEL_SETUP.delay = sc_core::sc_time(1, sc_core::SC_MS);

Scnsl::Core::Channel_if_t * CHANNEL_NAME = sim->createChannel(CHANNEL_SETUP);
```

3.1.5 Binding of nodes to channels, and setting of nodes' properties

First, for each transmission between pairs of tasks must be defined a unique bindIdentifier as follows:

```
BindSetup_base_t BIND_SETUP;
BIND_SETUP.extensionId = "core";
BIND_SETUP.bindIdentifier = "bind_id";
BIND_SETUP.destinationNode = DESTINATION_NAME;
BIND_SETUP.node_binding.bitrate = Scnsl::Protocols::YOUR_PROTOCOL::BITRATE;
BIND_SETUP.node_binding.transmission_power = 100;
BIND_SETUP.node_binding.receiving_threshold = 10;
BIND_SETUP.node_binding.x = 1;
BIND_SETUP.node_binding.y = 1;
BIND_SETUP.node_binding.z = 1;
```

The `bindIdentifier` plays the role of network interface and is used by the task to set the TaskProxy specific of the destination task. Then, each node has to be bound to each channel to which it is connected. The BindSetup object (`BIND_SETUP_NAME`) is used to set some node's properties, in addition

to the bindIdentifier.
Then, the structure is used to bing a given node and a channel:

```
sim->bind(NODE_NAME, CHANNEL_NAME, BIND_SETUP);
```

3.1.6 Instantiation of the Tasks

In order to instantiate a task use the following code:

```
MYTASK_T * TASK_NAME("task_name", TASK_ID, NODE_NAME, PROXIES);
```

3.1.7 Instantiation of communicators (optional)

```
CoreCommunicatorSetup_t COMMUNICATOR_SETUP;

COMMUNICATOR_SETUP.extensionId = "core";
COMMUNICATOR_SETUP.name = "the_communicator_name";
COMMUNICATOR_SETUP.type =
CoreCommunicatorSetup_t::MAC_802_15_4;
COMMUNICATOR_SETUP.node = NODE_OF_THE_COMMUNICATOR;

// Eventually set here other properties...

Scnsl::Core::Communicator_if_t *
    REFERENCE_PROTOCOL_COMMUNICATOR =
        sim->createCommunicator(COMMUNICATOR_SETUP);
```

3.1.8 Binding of tasks, communicators (optional) and channels

Communicators can be put between tasks and channels:

```
sim->bind(REFERENCE_TASK_NAME,
          DESTINATION_TASK_NAME,
          REFERENCE_CHANNEL_NAME,
          BIND_SETUP,
          REFERENCE_COMMUNICATOR_NAME);
```

- The destination task can be NULL for broadcast transmission or if the reference task is a receiver task;
- Each binding of Task/Channel/(Communicator) generates a TaskProxy instance.

3.1.9 Setting tracing features

SCNSL provides tracing capabilityes via an object named `Tracer`. A tracer object combines two utility objects: a `Filter` and a `Formatter`.

```
CoreTracingSetup_t SETUP;

SETUP.extensionId = "core";
SETUP.filterExtensionId = "core";
SETUP.formatterExtensionId = "core";
SETUP.filterName = "base_filter";
SETUP.formatterName = "base_formatter";
SETUP.print_trace_type = true;
SETUP.info = 5;
SETUP.debug = 0;
SETUP.log = 5;
SETUP.error = 0;
SETUP.warning = 0;
SETUP.fatal = 0;

Scnsl_t::Tracer_t * TRACER = scnsl->createTracer(SETUP);
```

3.1.10 Creating custom tasks

SCNSL is used to simulate network application. Therefore, users will provide application code into custom tasks.
The following example defines the structure of the task class inside the file `Hello_t.hh`:

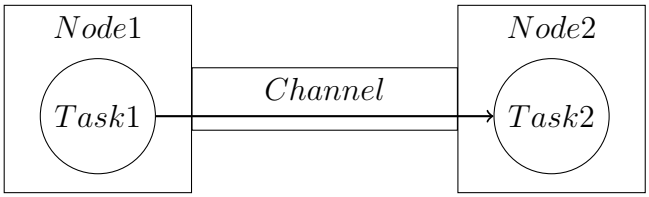
```
1  #include <systemc>
2  #include <scnsl.hh>
3
4  class Hello_t :
5  public Scnsl::Tlm::TlmTask_if_t
6  {
7      ...
8      SC_HAS_PROCESS( Hello_t );
9      ...
10     /// @brief Constructor.
11     /// @param name This module name.
12     /// @param id This module unique ID.
13     /// @param n The node on which this task is placed.
14     /// @param proxies The number of connected task proxies.
15     /// @param is_sender Switches this task behavior.
16     /// @throw std::invalid_argument If proxies is zero.
17     Hello_t(sc_core::sc_module_name name,
18             const task_id_t id,
19             Scnsl::Core::Node_t * n,
20             const size_t proxies,
21             const bool is_sender) throw (std::invalid_argument);
22     ...
23     // The standard TLM blocking transport, used to receiving:
24     virtual void b_transport(tlm::tlm_generic_payload & p,
25                              sc_core::sc_time & t);
26     ...
27     // The routine sending the message.
28     void sendingRoutine();
```

```
29  };
```

The constructor accepts a flag, `is_sender`, which will be used by the application to have a sender or receiver behavior. All the other constructor parameters are required by the parent.
The implementation of such task could be the following, contained in a file named `Hello_t.cc`:

```
1  #include "Hello_t.hh"
2  Hello_t::Hello_t(sc_core::sc_module_name name,
3                   const task_id_t id,
4                   Scnsl::Core::Node_t * n,
5                   const size_t proxies,
6                   const bool is_sender) throw (std::invalid_argument):
7                   Scnsl::Tlm::TlmTask_if_t( name, id, n, proxies )
8  {
9      if (is_sender)
10     {
11         SC_THREAD(sendingRoutine);
12     }
13 }
```

3.1.11 Example of Binding




```
1  MyTask * Task1("Task1", 0, Node1, 1);
2  MyTask * Task2("Task2", 1, Node2, 1);
3  ...
4  bsb0.bindIdentifier = "Task1_Task2";
5  bsb1.bindIdentifier = "Task2_Task1";
6  ...
7  sim->bind(Node1, Channel, bsb0);
8  sim->bind(Node2, Channel, bsb1);
9  ...
10 sim->bind(&Task1, &Task2, Channel, bsb0, NULL);
11 sim->bind(&Task2, NULL,   Channel, bsb1, NULL);
```

Chapter 4

Exercises

4.1 Exercises Setup

4.1.1 Compile the exercises

Recover and enter the "source/" folder provided with this tutorial.

```
cd source
```

Create the "build/" folder and move inside it.

```
mkdir build
cd build
```

Execute cmake and pass directly the library

```
cmake -DCMAKE_CXX_STANDARD=17 -DLIB_SCNSL=<lib path> ..
```

Where `<lib path>` is the path of SCNSL lib:

```
<lib path> = ${HOME}/Software/scnsl-stable-linux-x86_64/lib/libscnsl.so
```

Compile the code

```
make
```

4.1.2 Execute the exercises

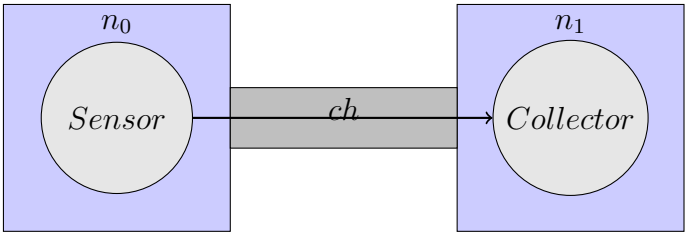
Execute each exercise and redirect its output to a text file

```
./Two_Nodes > Two_Nodes.log
./Three_Nodes_with_Router > Three_Nodes_with_Router.log
./Temperature_Monitoring 15 > Temperature_Monitoring.log
```

Use the script **calculatePLR.sh** (inside the "source/" folder) to calculate the Packet Loss Rate (PLR). The script takes as only parameter a text file containing the simulation traces.

```
../calculatePLR.sh Two_Nodes.log
../calculatePLR.sh Three_Nodes_with_Router.log
../calculatePLR.sh Temperature_Monitoring.log
```

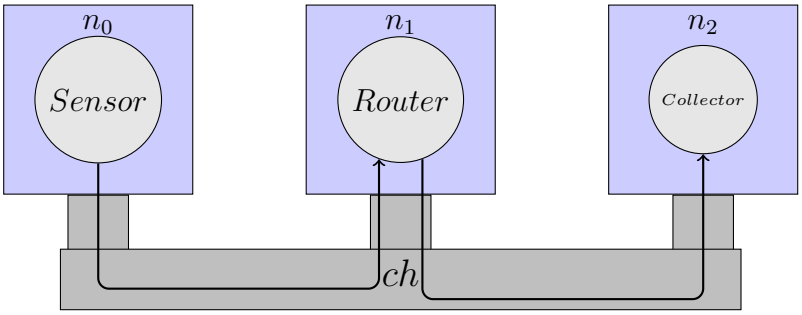
4.2 Exercise 1: Two Nodes



Calculate the minimum transmitting power of the sensor node n_0. Mantain unchanged the distance between nodes.

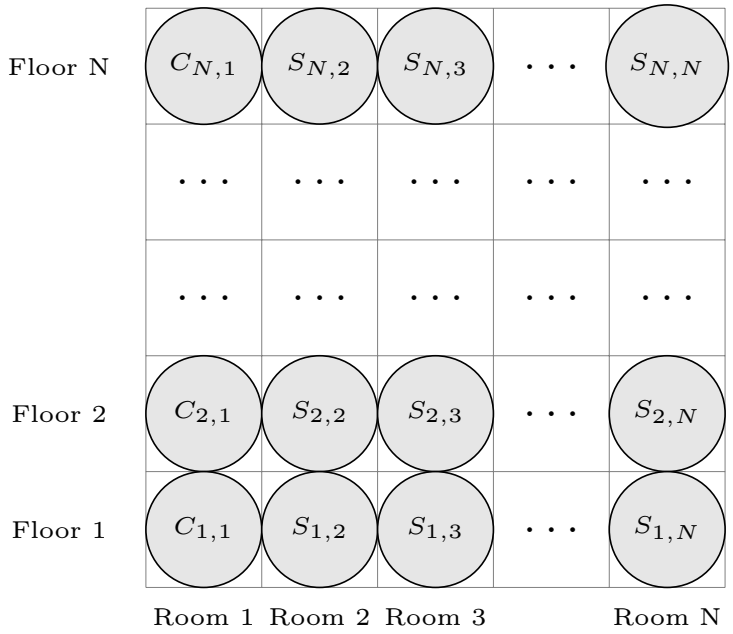
Hint: *if the transmitting power is lower than the minimum transmitting power, no packets will arrive to the receiver, i.e., Packet Loss Rate (PLR)=100%.*

4.3 Exercise 2: Three Nodes with Router



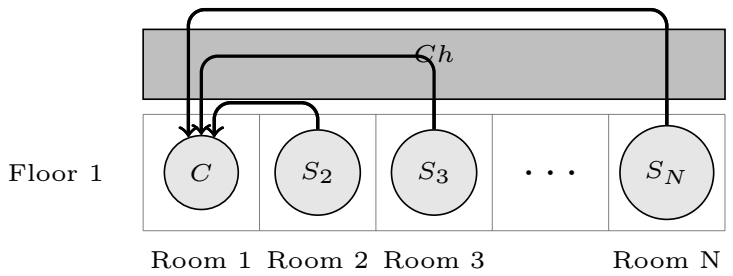
1. Calculate the delay:
   - Sensor-to-Router
   - Router-to-Collector
   - Sensor-to-Collector
2. Calculate the Packet Loss Rate (PLR).
3. Calculate the minimum transmitting power, both for sensor node n_0 and router node n_1, mantaining unchanged the distances between nodes.

4.4 Exercise 3: Temperature Monitoring for Building Automation



- N floors
- N rooms for each floor
- 1 controller for each floor
- 1 sensor for each room (#sensors > 0)
- Each sensor sends the detected temperature to the controller of its floor

4.4.1 Exercise 3.1



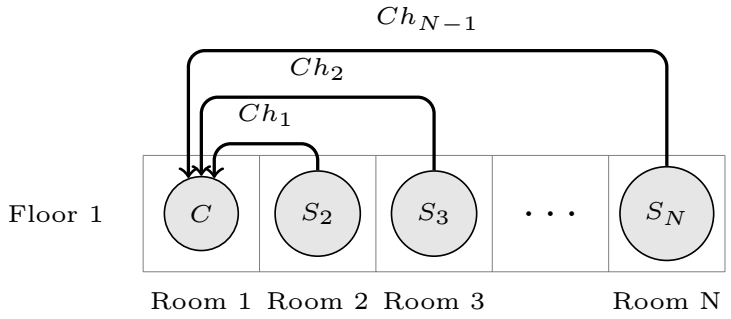
For this exercise we focus on the first floor. The idea is that the network scenario can be seen as a 1xN matrix:

- Node in the first column (Room 1) works as a collector node (RX only)
- Nodes in the other columns (Room 2 – Room N) work as sensor nodes (TX only)

In each sensor node (n_i , 2≤i≤N) the corresponding sensor task (S_i , 2≤i≤N) sends data to the controller task (C) through a shared multi-point channel.

1. Set the number of rooms (i.e., the number of nodes) to 5 and calculate the Packet Loss Rate (PLR).
2. Increase the number of rooms.
   - What about PLR as a function of the number of sensors?
3. Set the number of rooms to 3 (i.e., 2 sensors).
   - What about PLR?
4. Set the transmission power of each sensor to the minimum required to receive packets (use knowledge from the previous exercise).
   - Does PLR change?

4.4.2 Exercise 3.2



For this exercise, in each sensor node (n_i , 2≤i≤N) the corresponding sensor task (S_i , 2≤i≤N) sends data to the controller task (C) through a dedicated point-to-point channel (ch_i , 1≤i≤N-1).

1. Set the number of rooms (i.e., the number of nodes) to 5 and calculate the Packet Loss Rate (PLR).
    - What about PLR with respect to the previous exercise?
2. Increase the number of rooms.
    - What about PLR as a function of the number of sensors?
    - What about PLR with respect to the previous exercise?
3. How can a communication like this be realized in a real scenario?

That's all folks