



UNIVERSITÀ
di **VERONA**
Dipartimento
di **INFORMATICA**

UNIVERSITY OF VERONA

A.A 2018/2019

Laboratory of Networked Embedded Systems

Lesson 1

SystemC Network Simulation Library (SCNSL)

Enrico Fraccaroli

April 9, 2019

Contents

1	Introduction	3
1.1	Network Simulation	3
1.2	SystemC Network Simulation Library	3
1.3	SCNSL components	3
1.3.1	Task	3
1.3.2	Task Proxy	4
1.3.3	Communicator	4
1.3.4	Node	4
1.3.5	Channel	5
1.3.6	Environment	5
2	Installation and Setup	6
2.1	Requirements	6
2.2	Directory structure	6
2.3	SystemC Installation (If Missing)	7
2.4	SCNSL Installation	8
2.4.1	If LaTeX and Doxygen are Required	9
3	Network Scenario Creation	10
3.1	Creation Steps	10
3.1.1	Instantiate the SCNSL Simulator	10
3.1.2	Instantiate the Environment	11
3.1.3	Instantiate the physical Nodes	11
3.1.4	Instantiate the physical Channels	11
3.1.5	Bind nodes to channels, and set node's properties	11
3.1.6	Instantiate the Tasks	12
3.1.7	Instantiate communicators (optional)	12
3.1.8	Bind tasks, communicators (optional) and channels	12
3.1.9	Set tracing features	12
3.1.10	Creating a custom tasks	13
3.1.11	Example of Binding	14

4	Exercises	15
4.1	Exercises Setup	15
4.1.1	Compile the exercises	15
4.1.2	Execute the exercises	15
4.2	Exercise 1: Two Nodes	16
4.3	Exercise 2: Three Nodes with Router	16
4.4	Exercise 3: Temperature Monitoring for Building Automation	17
4.4.1	Version 2	18
4.4.2	Version 3	18

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

The application 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 implementation is crucial and many operations are connected to its modeling (*i.e.*, change of

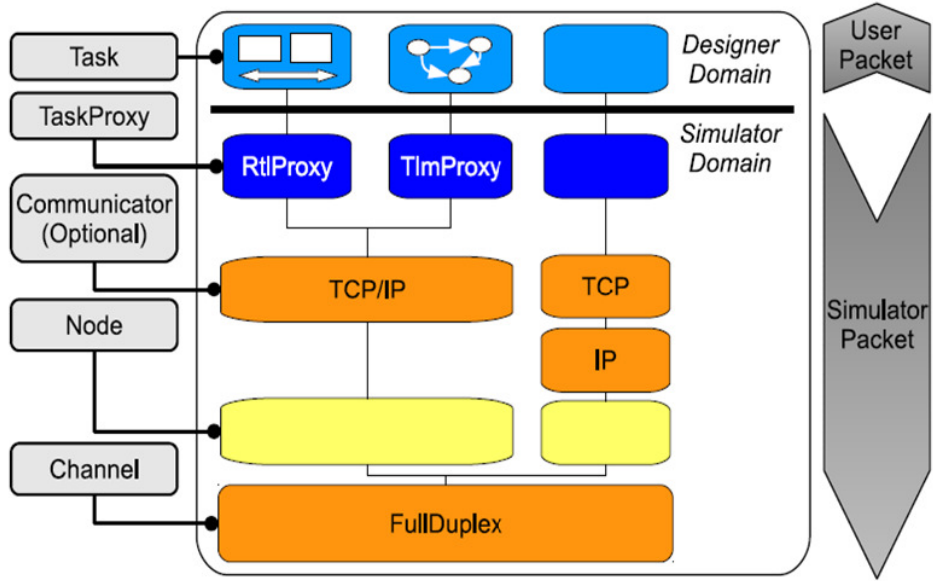


Figure 1.1: SCNSL components

abstraction level, validation, fault injection, HW/SW partitioning, mapping to an available platform, synthesis and so forth).

1.3.2 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 primitives for network communication.

1.3.3 Communicator

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

1.3.4 Node

Abstraction of physical devices. Tasks are hosted on 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

Models the physical transmission channel. For example wired and wireless are available.

1.3.6 Environment

Models some properties of the surrounding environment, also providing functions to get informations related to the transmissions of packets (*e.g.*, delay, error rate, *etc.*).

Chapter 2

Installation and Setup

2.1 Requirements

The lecture requires the following items

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

2.2 Directory structure

Let us assume the following directory structure:

```
HOME
|-- Software
\-- Source
```

let us export the following environment variables:

- Export the root directory of SystemC.

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

- Add its include directory to the PATH variable

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

- Add the path to the library directory to PATH

```
export PATH=${PATH}:${SYSTEMC_HOME}/lib-linux64
```

- Add it also to the variable LD_LIBRARY_PATH

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${SYSTEMC_HOME}/lib-linux64
```

- Create the directory where the scns1 library will be installed

```
mkdir -p ${HOME}/Software/scns1
```

- Export the root directory where the scns1 library will be installed

```
export SCNSL_HOME=${HOME}/Software/scns1
```

- 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 (If Missing)

SystemC can be downloaded at

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

Then, download the last release. For the rest of the procedure I will use the names and version of tools up to the date of April 9, 2019.

You should see the version **SystemC 2.3.1 (Includes TLM)**, and the link **Core SystemC Language and Examples**. This will download a compressed file which contains the source codes of SystemC. Then, unpack the compressed file:

```
tar xvf systemc-2.3.1a.tar.gz
```

Move inside the folder:

```
cd systemc-2.3.1a
```

And then run the configuration file by passing the place where the software should be installed, for instance:

```
./configure --prefix=/home/<username>/Software/systemc
```

In this case replace <username> with yours and be sure that both Software and its sub-directory systemc exists.

Check if the prompted paths at the end of the configuration process are correct.

Afterwards, compile and install it:

```
make install
```

2.4 SCNSL Installation

SCNSL is available to download at:

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

If you have the version control system **Bazaar** (similar to the most well-known **Git**), you can get the most updated version of the library, directly from the repository, by using bzz client.

```
bzz checkout bzz://scnsl.bzz.sourceforge.net/bzzroot/scnsl/trunk scnsl
```

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 ..
ccmake .
```

Compile the library

```
make install
```

Then compile all the tests

```
make tests
```

Copy the compiled library inside the Software folder on your root

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

In order to use the scnsl library, remember to add to the LD_LIBRARY_PATH environment variable the directory where the SCNSL library is located. You also have to add the SCNSL include directory to the PATH environment variable.

2.4.1 If LaTeX and Doxygen are Required

From inside the build directory open with gedit a specific CMake script executed during the generation of the Makefile, in particular:

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

Find the following lines and delete them

```
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. Instantiate the **SCNSL Simulator**.
2. Instantiate the **Environment**.
3. Instantiate the physical **Nodes**.
4. Instantiate the physical **Channels**.
5. **Bind** nodes to channels, and set node's properties.
6. Instantiate the **Tasks**.
7. Instantiate communicators (optional).
8. Bind tasks, communicators (optional) and channels.
9. Set tracing features.
10. Creating a custom tasks.

3.1.1 Instantiate the SCNSL Simulator

It is important, first of all, to create an instance of 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 Instantiate 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 Instantiate the physical Nodes

A node can be created with the following code:

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

3.1.4 Instantiate 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 Bind nodes to channels, and set node's 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` will be used by the reference task to set the `TaskProxy` specific of the destination task. Then, each node has to be bound which 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 bind a given node and a channel:

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

3.1.6 Instantiate 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 Instantiate 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 Bind tasks, communicators (optional) and channels

Communicators are required to bind tasks to 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;
- For each TaskProxy related to a Task, there must be the corresponding binding Task/Channel/(Communicator).

3.1.9 Set tracing features

SCNSL provides tracing capabilities 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 a custom tasks

SCNSL can be used also to test network application. In this case, users will provide the application code into a custom task.

Define the structure of the task class inside a 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 };

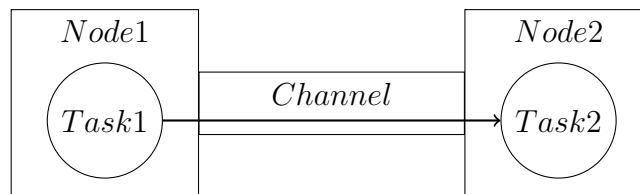
```

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 the application could be the following, 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

Download the file containing the exercises from the course page of

Networked embedded systems (2016/2017)

This should download a compressed file. Uncompress it and then move inside the “exercises_lesson1” directory.

```
cd source
```

Create the “build/” folder and move inside it.

```
mkdir build  
cd build
```

Execute cmake and pass directly the library

```
cmake -DLIB_SCNSL=${SCNSL_HOME}/lib/libscnsl.so ..
```

Compile the code

```
make
```

4.1.2 Execute the exercises

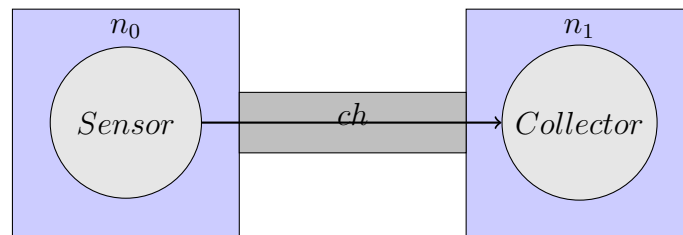
Execute each exercise and redirect its output to a text file

```
./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 root) 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
```

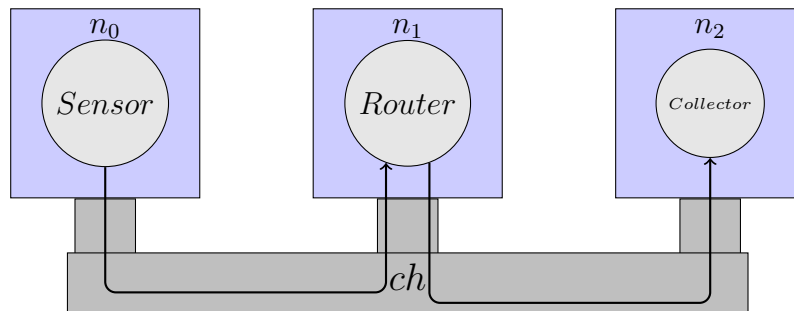
4.2 Exercise 1: Two Nodes



Calculate the minimum transmitting power of the sensor node n_0 . Maintain 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

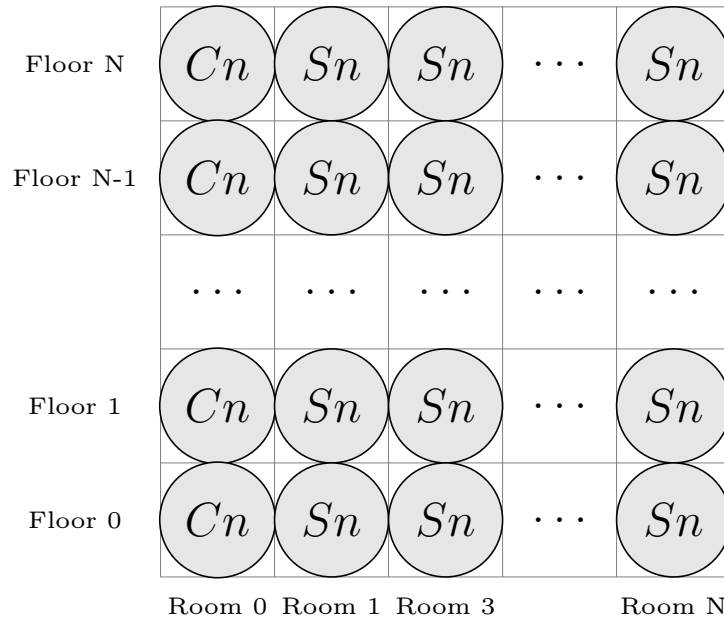


1. Calculate the delay:
 - Sensor-to-Router
 - Router-to-Collector
 - Sensor-to-Collector
2. Calculate the Packet Loss Rate (PLR).

3. Calculate the minimum transmitting power, both for sensor node n_0 and router node n_1 , maintaining unchanged the distances between nodes.

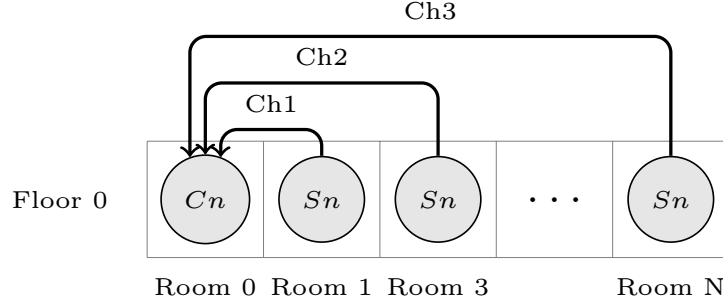
Hint: *help you with Exercise 1 to calculate the delay.*

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 Version 2

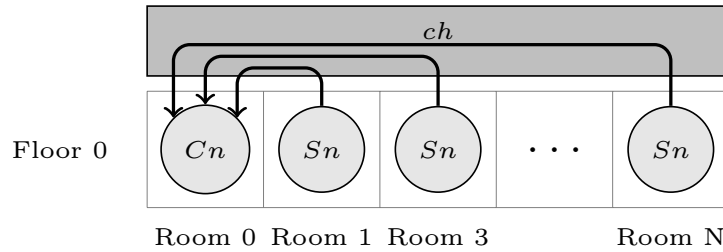


For this exercise we consider the first floor. The idea is that the network scenario can be seen as a $1 \times N$ matrix:

- Node in the first column (Room 0) works as a collector node (RX only)
- Nodes in the other columns (Room 0 – Room N-1) work as sensor nodes (TX only)

In each sensor node (n_i , $1 \leq i \leq N-1$) the corresponding sensor task (s_i , $0 \leq i \leq N-2$) sends data to the controller task (c) through a separate point-to-point channel (ch_i , $0 \leq i \leq N-2$).

4.4.2 Version 3



1. Set the number of rooms (i.e., the number of nodes) to 5 and calculate the Packet Loss Rate (PLR).
 - How can a communication like this be realized in a real scenario, for instance, in a Wireless Sensor Network?
2. Increase the number of rooms (i.e., the number of nodes).
 - How is the new PLR in respect to the node's distance?

3. Change the controller data collection from sensors, from the current point-to-point transmission to a shared one.
4. Set the number of rooms (i.e., the number of nodes) to 5 and calculate the Packet Loss Rate (PLR).
 - What can you say about the new PLR compared to the one of the point-to-point transmission?
 - Is the minimum transmitting power affected by the change to a shared communication?
5. Increase the number of rooms (i.e., the number of nodes).
 - How is the new PLR?
 - Does the increasing of sensor nodes affects the PLR?

That's all folks