ESD Electronic Systems Design

Synthesis Verification Testing Power Communication

NES Simulation with SCNSL

Alex Malfatti, Davide Quaglia

A.A. 2014/2015

Outline

- Introduction
- Installation & Setup
  - SCNSL
  - Examples
- Network scenario creation
- Exercises

Introduction

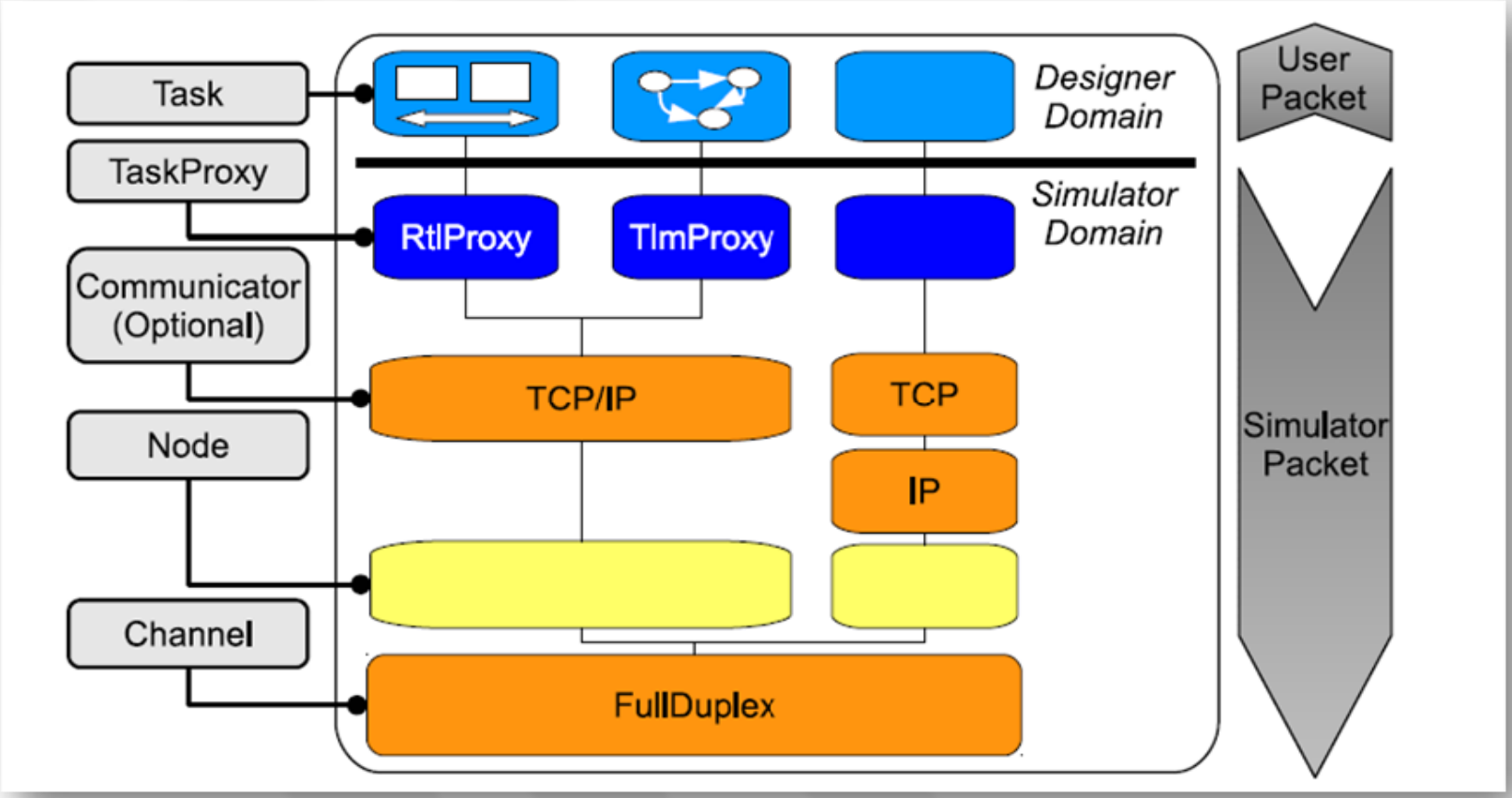
Network Simulation

- 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.

SystemC Network Simulation Library (SCNSL)

- 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.

SCNSL components (1)



SCNSL components (2)

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

SCNSL components (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.
- 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.
- Channel:
  - Models the pysichal transmission channel. For example wired and wireless are available.
- 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.).

Installation & Setup

Requirements

- Linux operating system
- SystemC library, version 2.2 or newer
- TLM library 2.0 or newer
- Cmake
- A C++ compiler and a linker
- Doxygen, for the documentation
- Latex, for the documentation

Install SCNSL (1)

- Untar the «*scnsl.tar.gz*» in your home directory (or wherever you prefer).

```
1$ tar -xzvf scnsl.tar.gz
```

  - 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 bzr client as follows:

```
1$ bzr checkout
   bzr://scnsl.bzr.sourceforge.net/bzrroot/scnsl/trunk
```

Install SCNSL (2)

- Move to the "*trunk/*" directory, inside the unpacked SCNSL directory, create a temporary directory (e.g., "*obj/*"), and move into it.

```
1$ cd scnsl
2$ cd trunk
2$ mkdir obj
3$ cd obj
```

- Export in the environment the paths to installed SystemC library, SystemC includes and TLM headers. A script, namely env-setup.sh, is provided to help this task: edit it (e.g., emacs, gedit, etc.) and then source it.

```
1$ emacs ./../scripts/env-setup.sh
2$ source ./../scripts/env-setup.sh 64
```

  - Source the script passing as argument your architecture word width (e.g. 32 or 64). Default is 32.

Install SCNSL (3)

- Run CMake, and optionally, run ccmake to configure other parameters.

```
1$ cmake ..
2$ ccmake ..
```

- Compile, running generated support files, and optionally generate the documentation.

```
1$ make doc
2$ make install
```

Setup Examples

- The previously procedure do not compile the examples, but only the SCNSL library.
- In order to compile also the examples, go to the "*tests/*" directory and repeat this procedure from the beginning.
  - Output libraries will be placed into the "*lib/*" directory
  - Relative headers into "*include/*" directory
  - Example executables will be placed into the "*bin/*" directory.
    - Remember to add to the *LD_LIBRARY_PATH* environment variable the directory into which the SCNSL library is located in order to run the examples.
  - Generated documentation will be installed into the "*doc/*" directory.

Network scenario creation

Custom scenario (1)

- In general the steps to follow when creating a network scenario in SCNSL are:
  - Instanciate the SCNSL Simulator.
  - Instanciate the environment.
  - Instanciate nodes.
  - Instanciate channels.
  - Bind nodes to channels, and set node's properties.
  - Instanciate tasks.
  - Instanciate communicators (optional).
  - Bind tasks, communicators (optional) and channels.
  - Set tracing features.

Custom scenario (2)

- SCNSL Simulator creation:

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

  - 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.

- Environment creation:

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

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

Custom scenario (3)

- Node creation:

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

- Channel setup and creation:

```
CoreChannelSetup_t NAME_OF_SETUP;

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

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

Custom scenario (4)

- Node's properties setup:

```
BindSetup_base_t BIND_SETUP_NAME;

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

- Node to channel binding:

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

Custom scenario (5)

- Task creation:

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

- Communicator creation (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 );
```

Custom scenario (6)

- Task to channel binding, using a communicator:

```
sim->bind( REFERENCE_TASK_NAME , DESTINATION_TASK_NAME ,
       REFERENCE_CHANNEL_NAME , BIND_SETUP_NAME ,
       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).

Binding mechanism (1)

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

```
BIND_SETUP_NAME.bindIdentifier = "bind_id";
```

  - This identifier 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.

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

  - The BindSetup object (*BIND_SETUP_NAME*) is used to set some node's properties, in addition to the *bindIdentifier*.

Binding mechanism (2)

- Finally, for each task and for each proxy defined on its creation …

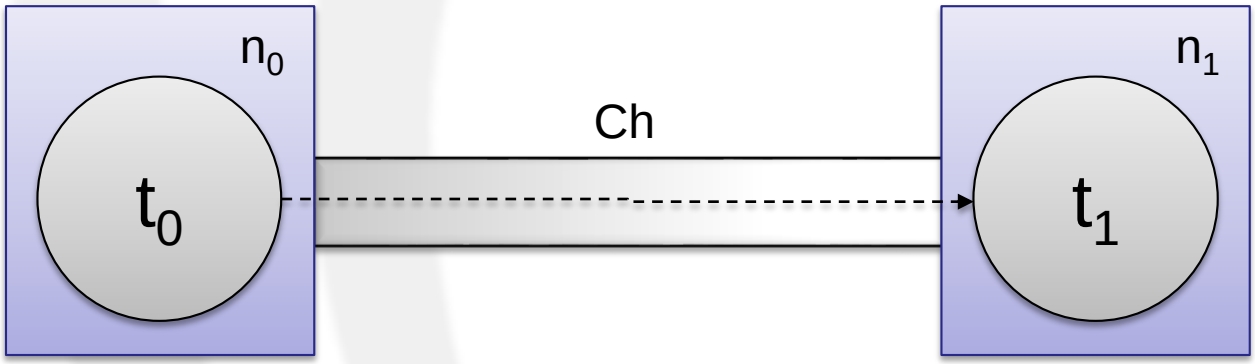
```
MYTASK_T * TASK_NAME(
        "task_name" , TASK_ID , NODE_NAME , PROXIES );
```

… the task must be bound with:

- Destination task, if it is a sender task in a one to one transmission; otherwise NULL;
- Channel hosting the transmission;
- BindSetup object;
- Communicator object (optional)

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

Binding mechanism (3)




```
MyTask * t0( "t0" , 0 , n0 , 1 );
MyTask * t1( "t1" , 1 , n1 , 1 );
...
bsb0.bindIdentifier = "t0_t1";
bsb1.bindIdentifier = "t1_t0";
...
sim->bind( n0 , Ch , bsb0);
sim->bind( n1 , Ch , bsb1);
...
sim->bind( & t0 , & t1 , Ch , bsb0 , NULL );
sim->bind( & t1 , NULL , Ch , bsb1 , NULL );
```

Exercises

Setup Exercises

- In order to compile the exercises, go to the "*trunk/*" directory and untar the «*exercises_nesLab2.tar.gz*».

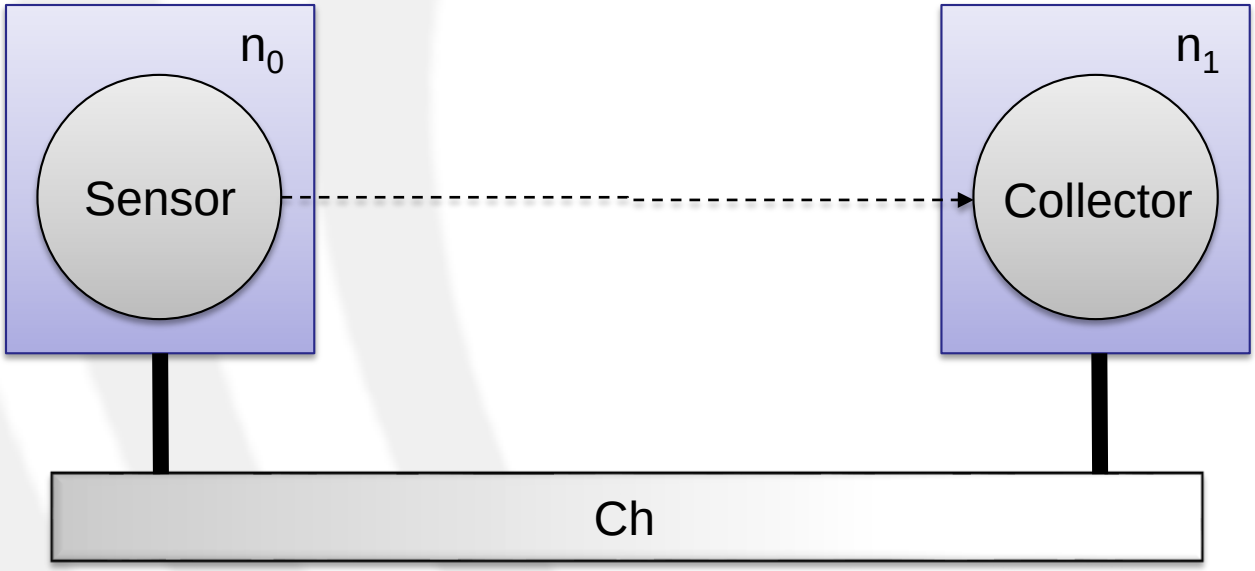
```
1$ tar -xzvf exercises_nesLab2.tar.gz
```

- Repeat from the beginning the same procedure used to compile and run the scnsl tests.
- A script to calculate the Packet Loss Rate (PLR) is provided ("*calculatePLR.sh*").

```
1$ ./calculatePLR.sh sim-traces.txt
```

  - The script takes as only parameter a text file containing the simulation traces.

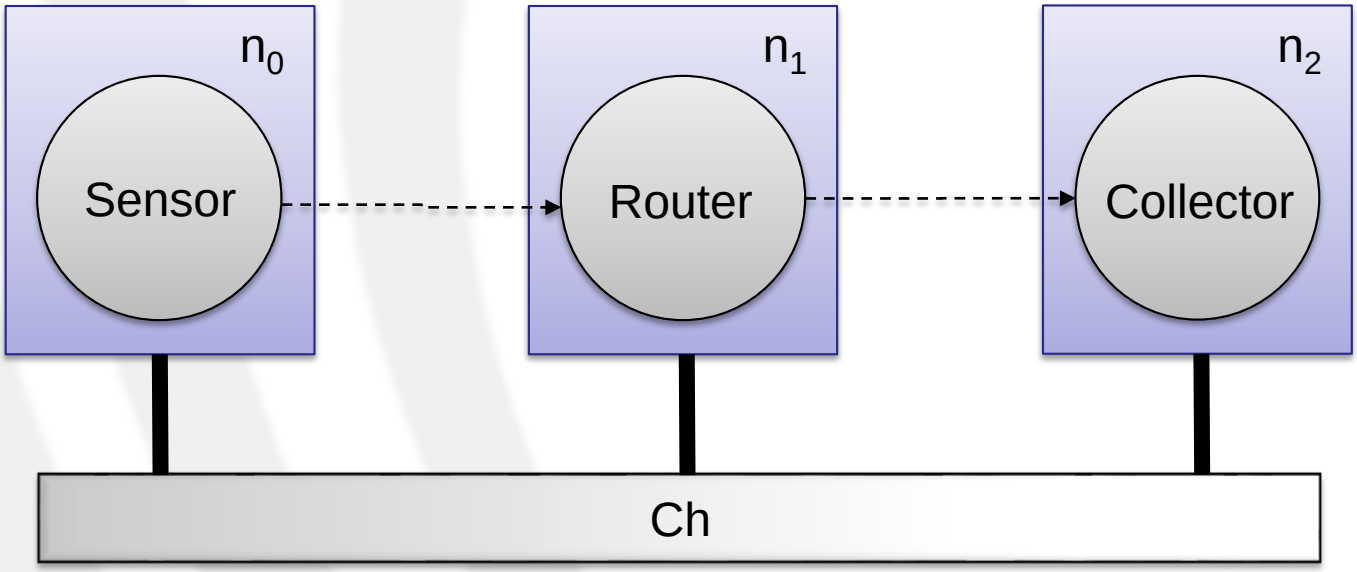
Exercise 1: Two Nodes



1. Calculate the minimum transmitting power of the sensor node n_0, mantaining 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%.*

Exercise 2: Three Nodes with Router (2)

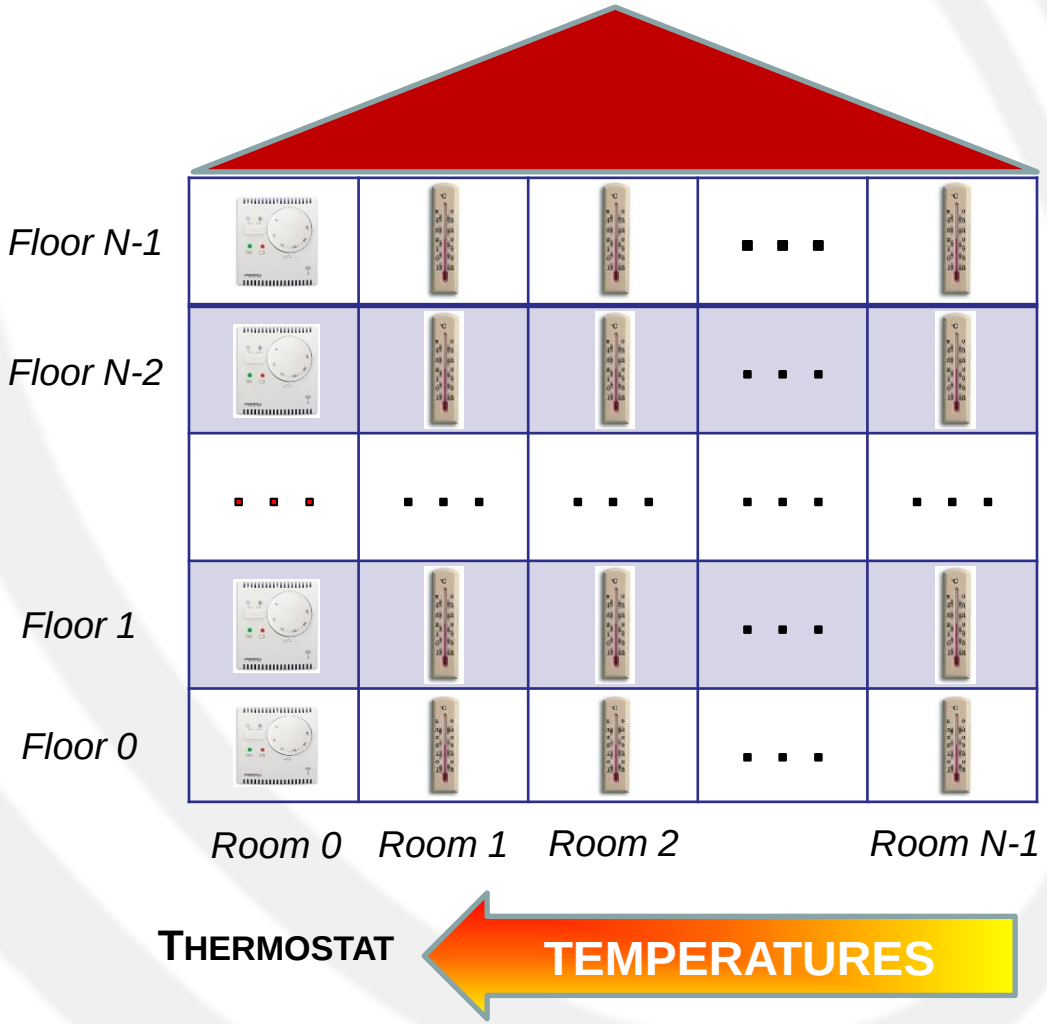


1. Calculate the delay:
   - Sensor-to-Router
   - Router-to-Collector
   - Sensor-to-Collector
2. Calculate the Packet Loss Rate (PLR).
3. Calculate the minimum transmitting power, both for sensor node n_0 and router node n_1, mantaining unchanged the distances between nodes.

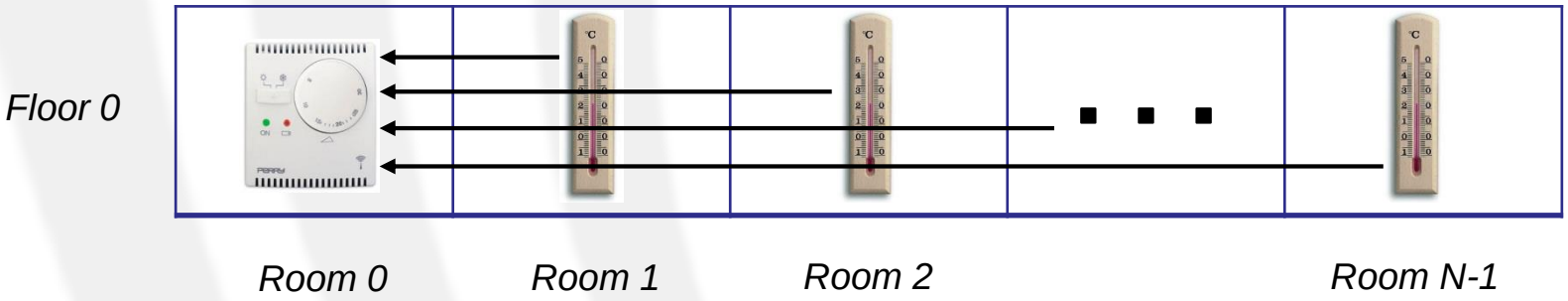
Hint: help you with Exercise 1 to calculate the delay.

Exercise 3: Temperature monitoring for Building Automation (1)

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

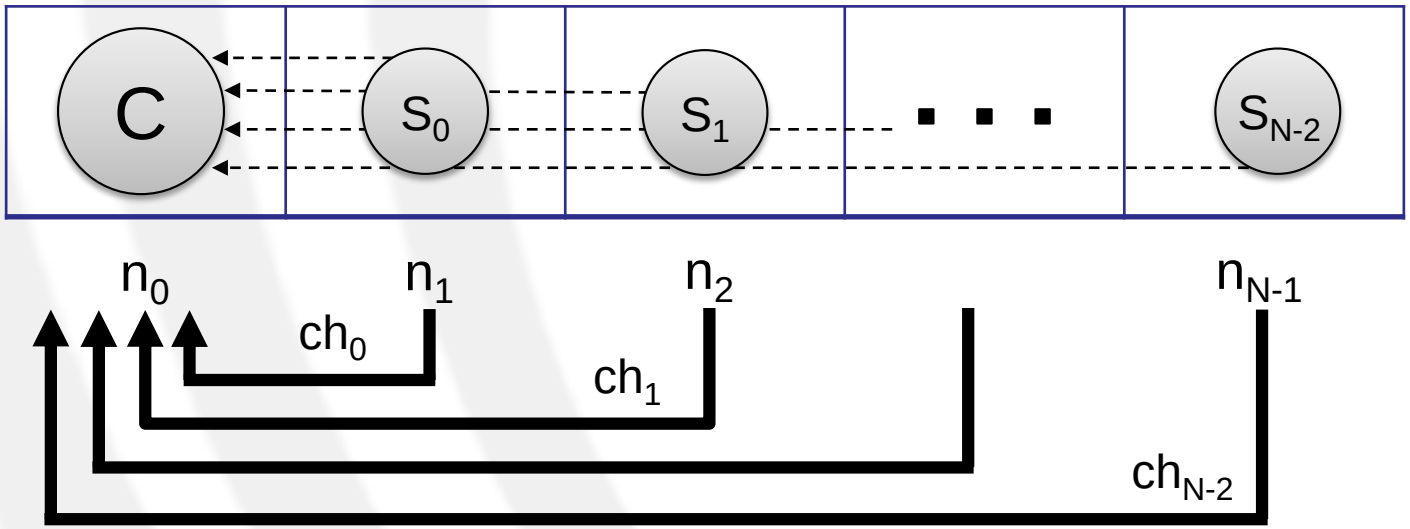


Exercise 3: Temperature monitoring for Building Automation (2)



- For this exercise we consider the first floor.
- The idea is that the network scenario can be seen as a 1xN 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)

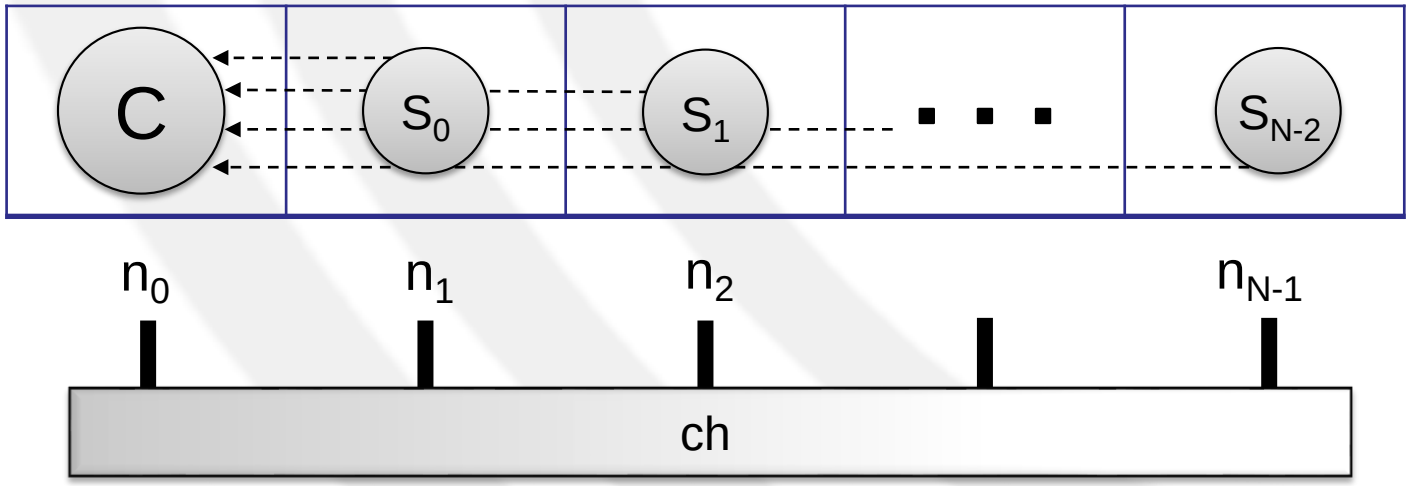
Exercise 3: Temperature monitoring for Building Automation (3)



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

Exercise 3: Temperature monitoring for Building Automation (4)

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.



Exercise 3: Temperature monitoring for Building Automation (5)

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?