Thursday, 13 December 2018

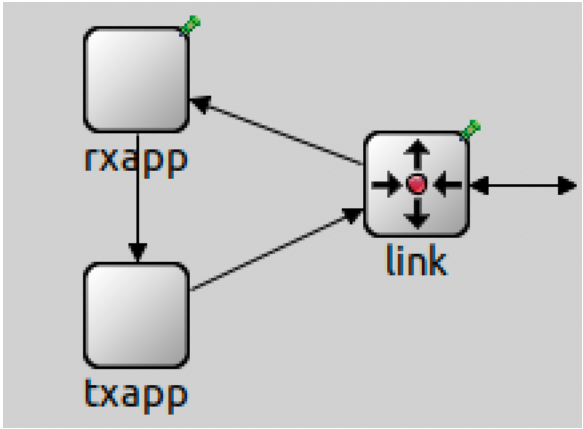
Network Simulation

Laboratory 2 - “Adding layers”

NOTE: It appears that a bug in OMNET++ prevents the correct use of arrays of pointers to events (e.g., timeout events), which may be used to complete this lab. You may not have run into this issue if you chose a different implementation method. However, an alternate version of Lab Task 2.2 is shown below. In your report state which version you implemented.

DEADLINE: JANUARY 7, 2019

In this lab, you will extend the simulation framework you worked on in Lab 1 and implement a two-layer architecture to distinguish between the task of routing messages from the task of generating (at the source) and receiving messages (at the destination). Each node will thus be changed from a single simple module into a compound module including three simple modules:



1. *linkLayer* : its task is simply to determine the next hop that the message has to take on its way to the destination, following whatever routing policy (random or rows/columns) you have implemented in Lab 1. It receives new messages from *txapp* and it delivers messages that have reached their destination to *rxapp.*
2. *rxapp* : it is handed by *linkLayer* the messages that have reached their final destinations. It destroys the messages and, after a time T_{ia} , it tells *txapp* to generate a new message
3. *txapp* : it generates a new message every time it is “woken up” by a message from rxapp.

LAB TASK 2.1: Delivery Delay with Row/column routing

Repeat the same topology and the same routing of Lab Task 1.3 and collect the same statistics, setting T_{ia}=0. You should obtain very similar results. This is a sanity check to make sure you have split the functionalities in a correct fashion.

LAB TASK 2.2: A stop-and-wait protocol

Set to 0.1 the probability of discarding a message at the *linkLayer* module. Implement a stop-and-wait protocol (using ACKs) to recover a lost message and retransmit it (to the same destination, obviously). Additionally, as in previous Tasks, a new message will also be generated by the node (by its txApp) that has just received the message. Since it is possible that a Node has multiple outstanding (i.e., without ACKs) messages at the same time, you need to assign a sequence number to each message to be able to identify which message is being ACKed.
Computing the delivery delay as the time needed to receive a message *including* the time taken up by its retransmission(s), collect the same statistics as Lab Task 2.1 (still using T_{ia}=0 for now).

LAB TASK 2.2a: A stop-and-wait protocol (Alternate version)

Set to 0.1 the probability of discarding a message at the *linkLayer* module. Implement a stop-and-wait protocol (using ACKs) to recover a lost message and retransmit it (to the same destination, obviously).

Implement the behaviour as follows:

- *txAPP*: the first message will be generated by txApp of every node at the beginning of the simulation, after a random time T_{ia}, negative exponentially distributed with parameter 0.1s. It will be sent to a random destination and a timeout will be set. If a message "ACK received" arrives from the rxApp in the same node, the timeout is cancelled and the transmission of a new message will be scheduled after another random time T_{ia}, negative exponentially distributed with parameter 0.1s.
- *rxApp*: If it receives a data message destined to the node, it will return a data message back to the sender as an ACK. If it receives an ACK message for the node, it will discard the ACK message and send an "ACK Received" message to the txApp in the same node.

Let the simulation end when each node has generated 10000*G/10 messages.

Computing the delivery delay as the time needed to receive a message *including* the time taken up by its retransmission(s), collect the same statistics as Lab Task 2.1 (still using T_{ia}=0 for now).

HINTS

- Use `getParentModule()` to access parameters and data of the compound module (such as its index number) from the c++ code of one of its simple modules.
- To manage the handling of different types of messages, when creating a new message, assign it a "Kind" as in Section 5.2.1 of the Omnet++ Manual. Then use `getKind()` to determine the kind of message when you receive it.

By January 7, submit a written report with:

- The answer to Tasks 2.1 and 2.2
- The .ini, .ned and C++ files you wrote for each of the tasks. Be sure to exhaustively comment your code.