NECO: NEtwork COding simulator

Diogo Ferreira    José Serra    Luísa Lima    Rui Prior    João Barros

Abstract

NECO is a simple high-performance simulation framework dedicated to the evaluation of Network Coding based protocols. Its main features include (1) definition of graphs representing the topology (which can be randomly generated by NECO or given through a standard representation), (2) the modular specification of network coding protocols, (3) visualization of network operations and seamless statistics module. The simulator is entirely written in Python and is easily extensible to account for extra modules.

Index Terms

simulation, network coding, random graph, python

I. INTRODUCTION

Since the seminal paper of Ahlswede, Li, Cai and Yeung [1], where it is proved that the max-flow min-cut capacity of a general multicast network can only be achieved by allowing intermediate nodes to mix different data flows, a surge in network coding research (e.g. [2], [3], [4]) has uncovered its potential to provide higher throughput and robustness, particularly where highly volatile networks such as mobile ad-hoc networks, sensor networks and peer-to-peer networks are concerned.

The basic idea behind network coding is illustrated in Figure 1. Suppose that node 1 aims at sending bits $a$ and $b$ simultaneously (i.e. multicast) to sinks 6 and 7. It is not difficult to see that the link between nodes 4 and 5 results in a bottleneck in the sense that either bit $a$ is forwarded (in which case node 6 does not receive bit $b$), or bit $b$ is sent (in which case node 7 will receive incomplete information). It follows that although the capacity of the network is 2 bits per transmission (because

D. Ferreira (dferreira@dcc.fc.up.pt), J. Serra (c0616024@alunos.dcc.fc.up.pt), L. Lima (luisalima@dcc.fc.up.pt), R. Prior (rprior@dcc.fc.up.pt) are with the Instituto de Telecomunicações (IT) and the Department of Computer Science, Faculdade de Ciências da Universidade do Porto, Portugal. J. Barros (jbarros@fe.up.pt) is with the Instituto de Telecomunicações (IT) and the Department of Informatics Engineering, Faculdade de Engenharia da Universidade do Porto, Portugal. Part of this work was carried out with assistance of financial support from the European Community under grant FP7-INFOSO-ICT-215252 (N-Crave Project). This work was partly supported by the Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) under grant SFRH/BD/24718/2005.
Fig. 1. Canonical network coding example: node 1 multicasts bits $a$ and $b$ to nodes 6 and 7. If node 4 did not perform a simple encoding operation on the incoming bits, the maximum network capacity would be 1 instead of 2.

the min-cut to each destination equals 2), this capacity cannot be achieved unless node 4 jointly encodes $a$ and $b$, for example, through an XOR operation that allows perfect recovery at the sinks.

Research in network coding has been heavily based on toy models and analytical tools until now. However, with the advent of practical protocols and applications, there is increasing need for a common simulation platform for network coding in complex networks.

Although implementations in NS [5], [6], Opnet [7] and other general network simulators have the significant advantages of using well-known frameworks and accessing a wide array of available libraries, there are also some underlying disadvantages. In particular, until now, there is no standard network coding library for these platforms – development tends to be scattered among different research groups and no common code basis has been set. These generic frameworks can also be deemed as feature heavy for an area of research which is in its beginning and for which the protocol stack has to be revisited or even rebuilt from scratch. A rather important disadvantage of this approach is that since network coding is most beneficial for unreliable and large networks, a crucial feature of such a simulator is to be able to simulate complex networks. However, heavily loaded with features, these frameworks must compromise performance, and thus, simulating complex networks can be a significant challenge.

Other related work in the simulation and emulation of network coding includes SlimSim and Lava. SlimSim [8] is a bare-bones simulator for wireless network coding. Although it has the advantage of having a small code-base and incorporating basic event-driven wireless simulation capabilities, it has the disadvantages of being a single-person effort, thus only implementing very basic features targeted for very specific research needs, and of having been written in C++, which can have a long development time. Lava [9] is an experimental testbed which uses a cluster of servers to implement realistic and controllable experiments using large volumes of traffic, with the goal of performing a
reality check of the use of network coding for live peer to peer multimedia streaming.

In this paper we present NECO (Network Coding Simulator), as a first step towards a common core for a high-performance open-source simulator for the network coding scientific community. NECO is entirely written in Python and allows for the evaluation of network coding based protocols. Its main features include (1) definition of graphs representing the topology (which can be randomly generated by NECO or given using a standard representation), (2) easy definition of network coding protocols, (3) visualization of the network operation and (4) seamless statistical analysis.

The remainder of the paper is organized as follows. Section II provides an overview of the usage of NECO by the means of use cases and examples. Abstraction and representation details are explained in Section III, followed by an in-depth overview of the simulator structure and important implementation details in Section V. Section VI provides an overview of the mechanisms for extension of modules. The paper is concluded in Section VII.

II. Usage

A typical usage of the simulator, both in graphical and text modes, consists of:

1) Generation of a Random Graph, or importing a static graph by using the Pygraphviz format;
2) Selecting sink(s) and source, at random or not;
3) Determination of the routing algorithm (for example, “directed diffusion”);
4) Determination the Network Coding Protocol (for example, “Random Linear Network Coding”);
5) Visualization of the network operation in real-time by using either the graphical user interface or the text output given in the terminal, or processing the statistics by interpreting the statistics file that is seamlessly generated.

In the following subsections, we give an overview of the usage of NECO.

A. Using and installing NECO

To use NECO, please visit [10]. There are two ways of using the simulator. The first and most straightforward is by installing VirtualBox [11], a virtualization software package, which is available for most operating systems. The virtual image provided in the website includes Ubuntu as a guest operating system and everything needed to run NECO is included in the image. The second requires the installation of the packages that NECO uses (see Section IV). An installation script is provided for Linux-based systems.

B. Graphical User Interface

NECO includes a graphical user interface for easy debugging, visualization of graphs and verification of protocol steps. An overview of its usage is provided here.
The program can be started from the command line with "python neco.py". The main window of the interface will appear. The necessary tabs will be highlighted as the simulation parameters are entered. On the graph tab, the type of the graph can be selected. The necessary running options will, again, be enabled accordingly. After selecting the options, the user should click on Generate graph. This step is shown in Figure 2.

![Fig. 2. Screenshot of the graph tab.](image)

On the protocol tab, the protocol used in the simulation can be chosen. Once again, the necessary options will be enabled. After selecting the options, the user should click on Next. The protocol tab is shown in Figure 3.

![Fig. 3. Screenshot of the protocol tab.](image)

On the simulation tab, the following options can be selected: simulation limit time, simulation
seed for the random number generator and number of destination nodes. If 0 is selected on the simulation limit time, the simulation will end only when all destination nodes have received the data that the sources sent. Both the source and sink nodes are selected uniformly at random. This is shown in Figure 4.

Fig. 4. Screenshot of the simulation tab.

Finally, on the shell tab, the user can start the simulation by pressing the Start button. In order for an user to follow the evolution of the communication of the time at a slower pace, a Slow Simulation mode is provided in Options. The simulation can be stopped at any time before it ends. While the simulation is running, an histogram of the number of packets received by the nodes in the network is updated every two seconds, if this option is enable in the Options tab, as shown in Figure 5. The shell, on the left, shows the nodes that have received the full data set. This is shown in Figure 6.

NECO includes some basic graph visualization options for convenience, which are enabled whenever the simulation is not running and the mouse pointer is located inside of the graph box. Nodes can be relocated by selecting them with the left button of the mouse. By scrolling the mouse or typing the "+" and "-" keys on the keyboard, the graph can be zoomed in or out, respectively. If the shift key is pressed and the user clicks on the graph by pressing the right button of the mouse, the entire graph will move.

Options for saving images of the graph and of the histogram are included in the File menu.

C. Command Line

For more complex graphs and simulations, it is recommended to use the command line option. By enabling this option, a full simulation can be performed, without using processing time on the
Fig. 5. Screenshot of the histogram evolution.

update of the user interface, so the simulation runs faster. The command line interaction is shown in Figure 7.

To start the command line options the program should be called with the "-c" flag. The command line options are shown in Table I, along with the graphs that require the respective options. The choice of the graph type in the command line is given by a number, which is given by the Id column in the table.

As for the protocols available and the options needed for each, they are as follows:

1) Flooding – generation size;
2) Random Linear Network Coding – generation size and logarithm of the size of the finite field.

It is mandatory to indicate the type of the graph with the corresponding options, the protocol
with the corresponding options, the simulation limit time, the simulation seed and the number of destination nodes.

D. XML file for loading and saving parameters

As of RC2, Neco can perform the setup of simulation parameters described in an XML file.

To create the XML file, the user can activate the options "Save simulation parameters" if the graphical user interface is being used, or use the flag $--save-sim-parameters$ if the command line is used. This will save the simulation parameters to the XML file when a simulation is performed. An XML can be created from scratch, by following a simple grammar which is exemplified in Figure 8.

```
<?xml version="1.0" ?>
<neco>
  <graph graph-seed="0543" type-graph="12" number-k="4"
        number-nodes="30" prob-edges-creation="0.16"/>
  <simulation sim-limit-time="500" number-dst-nodes="7"
              sim-seed="3"/>
  <nodes type="1">
    <protocol generation-size="6" index="2" size-of-field="8"/>
    <routing type="1"/>
  </nodes>
</neco>
```

To load the XML file, the user should select "Open simulation parameters file" if the graphical user interface is being used, or use the flag $--sim-parameters=filename.xml$ if the command line is being used. This will setup all the parameters needed to perform a simulation. The grammar options have a direct correspondence to the command line options and are as follows:

- The **graph** element
### TABLE I

**Command-line options for NECO.**

<table>
<thead>
<tr>
<th>Id</th>
<th>Graph</th>
<th>Mandatory options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Complete graph</td>
<td>--number-nodes</td>
</tr>
<tr>
<td>5</td>
<td>Path graph</td>
<td>--number-nodes</td>
</tr>
<tr>
<td>6</td>
<td>Star graph</td>
<td>--number-nodes</td>
</tr>
<tr>
<td>7</td>
<td>Wheel graph</td>
<td>--number-nodes</td>
</tr>
<tr>
<td>8</td>
<td>Random geometric graph</td>
<td>--number-nodes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--radius</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--graph-seed</td>
</tr>
<tr>
<td>9</td>
<td>Dual radio graph</td>
<td>--number-nodes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--radius1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--radius2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--prob</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--graph-seed</td>
</tr>
<tr>
<td>10</td>
<td>Fast gnp graph</td>
<td>--number-nodes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--prob-edges-creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--graph-seed</td>
</tr>
<tr>
<td>11</td>
<td>Gnp graph</td>
<td>--number-nodes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--prob-edges-creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--graph-seed</td>
</tr>
<tr>
<td>12</td>
<td>Newman watts strogatz graph</td>
<td>--number-nodes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--prob-edges-creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--graph-seed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--number-k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>--h, --help</td>
<td>display command-line options</td>
</tr>
<tr>
<td>--c, --command-line</td>
<td>run simulation in shell mode</td>
</tr>
<tr>
<td>--g, --gui</td>
<td>load graphical user interface</td>
</tr>
<tr>
<td>--sim-parameters</td>
<td>load simulation parameters from xml file</td>
</tr>
<tr>
<td>--x, --save-sim-parameters</td>
<td>save simulation parameters in xml file</td>
</tr>
<tr>
<td>--type-graph</td>
<td>type of the graph</td>
</tr>
<tr>
<td>--number-nodes</td>
<td>number of nodes</td>
</tr>
<tr>
<td>--number-lines</td>
<td>number of lines</td>
</tr>
<tr>
<td>--number-columns</td>
<td>number of columns</td>
</tr>
<tr>
<td>--dimension-n</td>
<td>n dimension</td>
</tr>
<tr>
<td>--dimension-m</td>
<td>m dimension</td>
</tr>
<tr>
<td>--hyper-dimension</td>
<td>hypercube dimension</td>
</tr>
<tr>
<td>--radius</td>
<td>radius</td>
</tr>
<tr>
<td>--prob-edges-creation</td>
<td>probability of creation of edges</td>
</tr>
<tr>
<td>--graph-seed</td>
<td>seed for the creation of the graph</td>
</tr>
<tr>
<td>--number-k</td>
<td>clustering coefficient k</td>
</tr>
<tr>
<td>--protocol</td>
<td>protocol</td>
</tr>
<tr>
<td>--protocol-seed</td>
<td>protocol seed</td>
</tr>
<tr>
<td>--generation-size</td>
<td>packet generation size</td>
</tr>
<tr>
<td>--size-of-field</td>
<td>logarithm of the size of the finite field</td>
</tr>
<tr>
<td>--link-capacity</td>
<td>capacity for the links</td>
</tr>
<tr>
<td>--sim-limit-time</td>
<td>simulation limit time</td>
</tr>
<tr>
<td>--sim-seed</td>
<td>simulation seed</td>
</tr>
<tr>
<td>--number-dist-nodes</td>
<td>number of destination nodes</td>
</tr>
<tr>
<td>--stats-filename</td>
<td>name of the file where to save statistics</td>
</tr>
</tbody>
</table>

- The mandatory attribute for the `graph` element is `type-graph`, which refers to the index corresponding to the type of the graph, as shown in Table I.

- The other mandatory attributes depend on the type of the graph, as shown in Table I. All other attributes are optional.

- **The `simulation` element**
  - All attributes for the `simulation` element are mandatory, as shown in Table I.

- **The `nodes` element**
  - The mandatory attribute for the `nodes` element is `type`. 

DRAFT
- The nodes element includes mandatorily the protocol and routing elements that are going to be ran on those nodes.

- The protocol element
  * The mandatory attributes for the protocol element are the ones shown in Table I.

- The routing element
  * The mandatory attributes for the routing element are the ones shown in Table I.

If there is more than one type of nodes, there should be more than one nodes element, each corresponding to the type, protocol and routing that is done on those nodes.

E. Scripting file

As of V1.0, NECO includes a simple scripting language for loading simulation parameters in a convenient way. To load the script, the user can write `neco -l sim.py` on the shell.

```python
g = NewmanWattsStrogatzGraph(0.5, 2015, 4, 20)
lst = []
for i in xrange(20):
    n = Node()
    if i%2 == 0:
        p = RLNCInnovative(8,2)
        p.setGenerationsize(5)
    else:
        p = RLNC(8,2)
        p.setGenerationsize(0)
    r = FloodingRouting()
    l = Link(i)
    n.setProtocol(p)
    n.setRouting(r)
    n.setLink(l)
    lst.append(n)
g.setNodes(lst)
simulationTime = 0
simulationSeed = 321
destinationNodes = 5
simulationId = "test"
```

Fig. 9. Scripting language for loading simulation parameters with NECO.

The script is interpreted using Python, thus any Python code that is included in the script is executed. The simulation is built, similarly to the remaining simulation execution methods, by creating a graph, using the constructors specified in the initialization.xml file (as shown in Figure 9. Then, nodes can be created by using the Node() constructor and assigning a protocol, routing scheme and mac layer type (that is, the "link" object) to each node. The nodes are then assigned to the graph. The
simulation time, seed, and number of destination nodes are specified afterwards. New constructors can be specified for new elements of the simulation.

*Table II* specifies the available constructors as of V1.0.

<table>
<thead>
<tr>
<th>TABLE II</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONSTRUCTORS FOR SCRIPT FILE SUPPORTED BY NECO</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graphs</th>
<th>CompleteGraph(int number of nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathGraph(int number of nodes)</td>
<td></td>
</tr>
<tr>
<td>StarGraph(int number of nodes)</td>
<td></td>
</tr>
<tr>
<td>WheelGraph(int number of nodes)</td>
<td></td>
</tr>
<tr>
<td>RandomGeometricGraph(int number of nodes, float radius, int seed)</td>
<td></td>
</tr>
<tr>
<td>DualRadioGraph(int number of nodes, float radius1, float radius2, int seed, float probability)</td>
<td></td>
</tr>
<tr>
<td>FastGnpRandomGraph(int number of nodes, int seed, float probability)</td>
<td></td>
</tr>
<tr>
<td>GnpRandomGraph(int number of nodes, int seed, float probability)</td>
<td></td>
</tr>
<tr>
<td>NewmanWattsStrogatzGraph(int number of nodes, int seed, float probability, int k)</td>
<td></td>
</tr>
<tr>
<td>RandomGeometricGraphTorus(int number of nodes, float radius, int seed)</td>
<td></td>
</tr>
<tr>
<td>DualRadioGraphTorus(int number of nodes, float radius1, float radius2, int seed, float probability)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Flooding(int seed); Flooding.setGenerationSize()</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLNC(int seed, int generation size); RLNC.setGenerationSize()</td>
<td></td>
</tr>
<tr>
<td>RLNCInnovative(int seed, int generation size); RLNCInnovative.setGenerationSize()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Links</th>
<th>Standard(int capacity)</th>
</tr>
</thead>
</table>

| Nodes | Standard; Standard.setProtocol(), Standard.setRouting(), Standard.setLink() |

**F. Statistics and Simulation Output**

The statistics are saved in a *python hash*, a flexible data structure which is then stored in a *python cPickle* file, and can be interpreted according to its structure. For more details please refer to Section III.

**III. SIMULATION ENGINES AND REPRESENTATION**

In what follows, we present an overview of the internal representation of the main data structures of the simulator, as well as of the abstractions made.

**A. Underlying graph**

The network is represented by a graph, which can be a representation of a real topology in Graphviz format [12], [13], or as a random graph generated on the spot using NetworkX algorithms.
Random graphs are widely accepted [14] as representative models of typical topologies of several types of networks. Table III shows the types of random graphs that are supported by the random graph package in NECO.

**TABLE III**

**RANDOM GRAPHS SUPPORTED BY NECO**

<table>
<thead>
<tr>
<th>Regular Graphs</th>
<th>Complete graph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Grid 2d graph</td>
</tr>
<tr>
<td></td>
<td>Grid graph</td>
</tr>
<tr>
<td></td>
<td>Hypercube graph</td>
</tr>
<tr>
<td></td>
<td>Path graph</td>
</tr>
<tr>
<td>Generic Models</td>
<td>Érdos-Rényi Model (gnp, fast gnp)</td>
</tr>
<tr>
<td></td>
<td>Small-World Networks (Newman-watts algorithm)</td>
</tr>
<tr>
<td>Wireless Models</td>
<td>Random Geometric Graph</td>
</tr>
<tr>
<td></td>
<td>Dual Radio Networks</td>
</tr>
<tr>
<td>Other Models</td>
<td>Star graph</td>
</tr>
<tr>
<td></td>
<td>Wheel graph</td>
</tr>
</tbody>
</table>

**B. Network and protocol abstractions**

Since our main focus is on the simulation of protocols for complex networks, we abstract from the network stack and implement a simplified version produced specifically for network coding protocols, which is shown in Figure 10. The incoming and outgoing links are represented by two buffers – the `inBuffer` and the `outBuffer`, respectively. Network coding protocol implementations simply check the `inBuffer` for received packets; the packets undergo processing in the main NC components, that is, coding and path selection, on their `outBuffer`.

In order to save processing time in network coding protocols, since all quantities of interest can be either directly measured or computed from the encoding matrices present at each node, network coding is performed solely on the encoding vectors [2], [3] of each packet, that is, the payload of the packets is not included in the simulation.

**C. Metrics and Statistics**

NECO contains methods for seamless saving statistics, which are in the `Statistics` class. The `Statistics` class contains two main methods: `writeConstant` and `writeTimeDependent`. The first one can be used to save constant information, such as constant graph parameters or simulation seeds. The second one can be used to save variables that change over time such as, for example, the number of
Fig. 10. A simplified stack for network coding protocol testing.

packets that each node received on each simulation step. This setup allows the user to save statistics in a seamless way.

IV. DEVELOPMENT CHOICES

In this section, we present an overview of the development choices regarding language and chosen libraries.

A. Language and Language Implementation

We chose the Python programming language [15] for several reasons. The first is the high legibility of the language, which leads to a reduced development time and high productivity, as well to improved program maintenance and extensibility. The second main reason is the platform universality: Python is available for the main operating systems with no changes in code needed. Python is also used extensively for scientific applications, and in particular, two excellent mathematics and engineering libraries – SAGE [16] and Pylab [17] are freely available. Seamless integration is Finally, NS-3 is expected to implement python bindings, which can be a plus when developing more complex protocols.

B. Other libraries

All the libraries used in the development of the simulator are according to the GPL licence (GNU General Public Licence) and are open source. In particular, for the creation, manipulation and study of
random graphs, the NetworkX library version 0.35.1 [19] is used. The SAGE library version 3.0.5 [16] is used for its finite fields implementation and algorithms, as well as other mathematics algorithms. The PyQt library version 4.3.3, with Sip version 4.3.3 and OpenGL libraries version 3.7, are used for the implementation of the graphical user interface.

C. Licence and possibilities for extension

The NECO simulator is subdivided into core modules, which are the modules that provide it with the minimal feature set (basic protocols such as flooding and random linear network coding, flooding routing, graph generation, scheduler, graphical user interface and command line), and extension modules, which extend its basic functionality, for example, to account for more elaborate protocols. This division is illustrated in Figure 11. In principle, to implement extension modules, there should be no need to interfere with the core modules.

We release the core modules of NECO under the GNU General Public Licence [20]. The licence of the extension modules is left to the choice of the institutions that implement them.

![Fig. 11. Main modules of NECO. The core includes the barebones of communication and protocols, graph generation, graphical user interface, and scheduler. The extensions (shown in blue with dashed border) can include other modules or extensions of the core modules, as well as possible bindings to external simulators or libraries.](image)

V. Simulator Structure and Implementation

We now give a brief overview of the simulator structure and some implementation details. The NECO code is divided into two main modules: core and gui.

The core module includes the control of all the simulation steps and other modules, as well as the basic components for building and using networks and protocols. The UML class diagram for this module is shown in Figure 12. Its main classes are: classes related to the simulation core, concurrency
and scheduling (that is, NecoCore, Scheduler, SimulatorThread, NodeThread), graph related classes such as Graph, Node, Link and protocol related classes such as Protocol and Routing. A brief overview of the operation of the core is as follows.

The generation of the graph is of the responsibility of the object PrepareGenerationGraph which runs on a separate thread, and then calls methods from the class GraphGenerator to generate the graph using Networkx [19].

NecoCore is responsible for the communication with the graphical user interface, since no options have been chosen yet when the program is started. NecoCore is also responsible for starting the simulation, which is done by calling the methods in the Simulator class. This class includes methods to communicate between the core and the simulation, control the simulation time, and hence runs on a separate system thread from the simulation. However, it is also responsible for starting the simulation thread, which is specified in the SimulationThread class.

The SimulationThread is also responsible for randomly assigning the source and destination nodes, to control the scheduler and update the statistics. For each node of the graph, an object called NodeThread is created, which represents a node in the simulation. This includes methods to instantiate
the buffers as well as the protocol that the nodes in the network are running.

The Scheduler class acts as a queue sorted by time. Within each timestep, the tasks are ordered by priorities. Each task is represented by an object of the Event class. This class includes a pointer to the method that is to be performed.

A Protocol object is created for each node of the simulation. The Protocol main method runs as a cycle with the following steps: (1) check whether there are packets for processing in the input buffer, (2) execute the intermediate node behavior, (3) execute the sink node behavior, (4) check whether there are packets in the output buffer to be sent to other nodes. The Protocol object then communicates with the Routing class to determine the next hop of each packet transmitted.

The NecoParser class generates virtual classes corresponding to each element of the simulation (that is, protocols, links, routing, nodes and graphs) from the initialization.xml file. It then parses the scripting file that is provided by the user and feeds it to the simulation engine.

Two “dummy” classes, SharedSimInfo and UpdateUIInfo, are present for improved readability and extensibility of the code. The SharedSimInfo class is responsible for the variables that are shared for all the nodes, like the instance of the random class, the generationSize, the number of nodes on the simulation, the instance of the SAGE class for performing computation on a finite field, the basis of the finite field, the matrix whose elements belong to the chosen finite field and the logarithm of the size of field.

The UpdateUIInfo class is responsible for the methods to update the user interface information, methods like, shellAppend, that send the string to the parent to be shown to the user, if there is a user interface send to it, if not just print the string and updateConnsGL, that update the links that are drawn The complete documentation for the NECO source code is available at the NECO wiki, at [10].

The graphical user interface is divided in four classes. The first one, uiApplication, includes

Fig. 13. UML class diagram for GUI.
the methods that are responsible for the creation and management of the graphical user interface by creating the objects, placing them and controlling the gui events. In order to draw the graph, *uiApplication* uses the *glWidget* class methods, which render the graph using opengl. On the other side, to draw the histogram, *uiApllication* uses the *Histogram* class methods, which implement the histogram calling. The *BarCurve* class includes methods to draw the bars of the histogram. The UML class diagram for the *gui* is shown in Figure 13.

VI. IMPLEMENTATION OF EXTENSION MODULES

In the following subsections, we describe how to implement extensions for NECO in terms of new protocols, routing protocols, graph and network properties.

A. XML file for loading extension modules

As of RC3, Neco can load extension modules through a description present in an XML file, *initialization.xml*. This XML file follows the grammar which is exemplified in Figure 14.

```xml
<xml version="1.0" ?>
<neco>
  <superclasses>
    <superclass grouponm: protocols" superclassName="AbstractProtocol" />
    <superclass grouponm: routing" superclassName="AbstractRouting" />
    <superclass grouponm: links" superclassName="AbstractLink" />
    <superclass grouponm: nodes" superclassName="AbstractNode" />
    <superclass grouponm: graphs" superclassName="AbstractGraph" />
  </superclasses>
  <newattribute id="0" name="Logarithm of size of field" variableName="rLnC" maxValue="16" minvalue="1" defaultValue="0" commandLineParameter="size-of-field" type="int" />
  <newattribute id="1" name="Protocol seed" variableName="protocolSeed" maxValue="999999" minValue="1" defaultValue="1" commandLineParameter="protocol-seed" type="int" />
  <newattribute id="3" name="Capacity" variableName="capacity" maxValue="999" minValue="1" defaultValue="1" commandLineParameter="link-capacity" type="int" />
  <newattribute id="14" name="Routing protocol seed" variableName="routingSeed" maxValue="999999" minValue="1" defaultValue="2" commandLineParameter="routing-seed" type="int" />
  <newattribute id="0" name="Complete graph" src="completeGraph" className="CompleteGraph" attribute="0" param="n" />
  <newattribute id="1" name="list" setter="true" constructor="false" />
  <newattribute id="0" name="Flooding" src="floodingRouting" className="FloodingRouting" attribute="14" param="g" />
  <newattribute id="2" name="FLNC discards non innovative packets" src="FLNCInnovative" className="FLNCInnovative" attribute="0" param="g" />
  <newattribute id="3" param="s" />
  <newattribute id="4" type="int" />
  <newattribute id="4" type="int" setter="true" constructor="false" />
</protocol14
```

Fig. 14. XML grammar for loading and saving simulation parameters with NECO. The *newattribute* element is used to create parameters for extensions, such as *protocol* extensions, *routing* extensions, *node* extensions and others. The *graph*, *protocol*, *routing*, *packet*, *link* and *node* elements contain *attribute* elements, which must have been defined previously, with the mandatory parameters for each. If there is more than one type of protocol, routing protocol, packet type, link type or node type, there should be a corresponding element for each type.
The XML file is formed according to the following rules:

- **The newattribute element**
  - The newattribute element is used to create parameters for extensions, such as protocol extensions, routing extensions, node extensions, etc. It includes the following mandatory attributes:
    * id – the attribute id, used to identify and use it.
    * name – the name of the attribute, which is displayed in the GUI.
    * variableName – the name of the corresponding variable in the code.
    * maxValue – the maximum value allowed for the attribute.
    * minValue – the minimum value allowed for the attribute.
    * defaultValue – the default value of the attribute.
    * commandLineParameter – the parameter for the command line.

- **The graph, protocol, routing, packet, link and node elements**
  - All attributes for these elements are mandatory:
    * id – the protocol id, used to include it in the other elements.
    * name – the name of the protocol, which is displayed in the GUI.
    * src – the name of the module in the source code.
    * className – the name of the class in which the protocol is defined.
  - Then, the necessary attributes, which must have been defined previously, as stated above, follow as elements of the protocol. For example, to use the attribute with id 0, this line should follow: `<attribute id="0"/>`
  - Afterwards, the variables for the constructors of the object are given as param elements, which include its id, type of the variable, and a setter if the class is to have a setVariable method.

If there is more than one type of graph type, protocol, routing protocol, packet type, link type or node type, there should be a corresponding element for each type.

**B. Implementation of new network protocols**

To implement a new protocol, the user should create a new class for the protocol and override five methods, `__init__, sourceBehaviour, intNodeBehaviour, sinkBehaviour, sendOutBuffer` and `seeInBuffer`. This is shown in Figure 15.
C. Implementation of new routing protocols

All routing protocols extend the Routing class. To implement a new routing protocol, the user should create a new class that extends the Routing class. The new routing protocol can override the Routing methods, as well as creating new ones. It’s recommended to only override the method `selectNextHops`. This is shown in Figure 16.

```python
import routing

class NewRouting(routing.Routing):
    def __init__(...):
        ...
    def selectNextHops(...):
        ...
```

D. Implementation of new link types

```python
import link

class NewLink(link.Link):
    def __init__(...):
        ...
    def sendBehaviour(...):
        ...
```

Fig. 16. Implementation of a new routing protocol.

Fig. 17. Implementation of a new link type.

All links extend the Link class. To implement a new link type, the user should create a new class
that extends the Link class and override the sendBehaviour method to specify the new link behavior. This is shown in Figure 17.

E. Implementation of new packet types

```python
import packet

class NewPacket(packet.Packet):
    def __init__(...):
        ...
    def newMethod(...):
        ...
```

Fig. 18. Implementation of a new link type.

All packets extend the Packet class. To implement a new packet type, the user should to create a new class that extends the Packet class and add the desired attributes. This is shown in Figure 18.

F. Implementation of generation of new graph types

All packets extend the GenGraph (Generic Graph) class. To implement a new graph type, the user should to create a new class that extends the GenGraph class and add the mandatory attributes, which are graph and positions. The graph should be a NetworkX Graph type, as shown in Figure 19. Then, the user should add the corresponding lines to the file initialization.xml, similarly to a normal plugin.

```python
import networkx as NX
from genGraph import *

class CompleteGraph(GenGraph):
    def __init__(self, layout, attributes, parent):
        self.parent = parent
        self.graph = NX.complete_graph(attributes['nNodes'][self.selectedValue], create_using=None)
        if layout:
            self.positions = NX.graphviz_layout(self.graph, layout)
```

Fig. 19. Implementation of a new graph type.

G. Implementation of new node behaviours

This feature shall be enabled in future releases.

VII. CONCLUSIONS AND FURTHER WORK

We presented an open-source network coding simulator, with a high performance and easily extensible core.
We present our short-term future releases, with timings, in Table IV. As part of our long-term future work, we are considering the addition of more graph models, including evolving networks for evaluation of distributed storage and peer to peer models and mobility models. We also envision the development of a split-programming model such as the one present in NS-2.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge insightful discussions with João Paulo Vilela, Sérgio Crisóstomo, Pedro Brandão, Paulo F. Oliveira and João Mendes from Universidade do Porto, Portugal, Jay Kumar Sundararajan from Massachusetts Institute of Technology, USA, Alberto Toledo and Steluta Gheorghiu, from Telefonica Espana, Spain, and the remaining partners from the N-CRAVE project.

REFERENCES


