NECO: NEtwork COding Simulator

Diogo Ferreira
Instituto de Telecomunicações
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Portugal
dferreira@dcc.fc.up.pt

Luísa Lima
Instituto de Telecomunicações
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Portugal
luisalima@dcc.fc.up.pt

João Barros
Instituto de Telecomunicações
Dep. de Eng. Electrotécnica e de Computadores
Faculdade de Engenharia da Universidade do Porto
Portugal
jbarros@fe.up.pt

ABSTRACT

We present NECO, a 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 generated randomly or pre-defined by means of a standard representation), (2) modular specification of network coding protocols, (3) visualization of the network operation and (4) extraction of key statistics. The simulator is entirely written in Python and can be easily extended to account for extra functionalities.

Categories and Subject Descriptors

C.2.1 [**Computer-Comunication Networks**]: Network Architecture and Design—*Network Communications*; C.2.2 [**Computer-Comunication Networks**]: Network Protocols—*Applications*; I.6.0 [**Simulation and Modeling**]: General

General Terms

Network Coding, Simulation, Networks

Keywords

network coding, simulation, topology, random graph

1. INTRODUCTION

Research in network coding (that is, algebraic mixing of packets in networks) has been so far heavily based on toy models that are amenable to mathematical treatment. However, with the advent of practical protocols and sophisticated network coding applications, the need to characterize their behavior in large networks motivates the development of adequate simulation tools.

The key insights of [12], which 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, has caused a surge in *network coding* research (e.g. [16, 20]) uncovering 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.

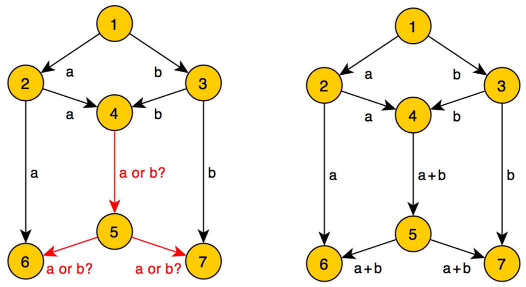


Figure 1: Canonical network coding example: node 1 multicast bits a and b to nodes 6 and 7. If node 4 did not perform a simple encoding operation on the incoming bits, the maximum network capacity of 2 could not be achieved.

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 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.

Random Linear Network Coding (RLNC) [16] is a distributed methodology for performing network coding, in which each node in the network independently and randomly selects a set of coefficients and uses them to form linear combinations of the data symbols it receives. These linear combinations are then sent over the outgoing links of each node. Each symbol or packet is sent along with the global encoding vector [18], which enables the receivers to decode the original data using Gaussian elimination, provided that the received matrix has full rank. It turns out that RLNC is sufficient to reach the multicast capacity of a network [19, 18, 16]. In addition, robustness gains have been reported in packetized networks with lossy links in [20]. Recent successful applications of RLNC include peer-to-peer networks for content distribution [14, 13] and wireless networks [17, 22].

Network coding simulation presents the following significant main challenges in comparison to traditional routing protocols:

- Since network coding is particularly beneficial in unreliable and large networks, the simulator must be capable of reproducing the dynamics of complex networks, that is, networks with a very large number of nodes, in an efficient way;

- Because protocol stacks for network coding are yet to be defined, the simulator should be as generic as possible, such that many features of classical network simulators become excessive and should be avoided.

Although implementations in *NS* [11], *Opnet* [6] and other general network simulators offer the advantage of well-known frameworks and a wide array of available libraries, there are significant disadvantages in using them for network coding. In particular, until now, a standard network coding library for these platforms has not emerged – 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. Frameworks with too many features compromise performance and simulation time, hence simulating the aforementioned complex networks remains a significant challenge. Other related work in the simulation and emulation of network coding includes *SlimSim* [9], which 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, with very basic features targeted for specific research needs.

Our main contribution is NECO (Network Coding Simulator), a first step towards a common core for a high-performance open-source simulator for the network coding scientific community. It is entirely written in Python and allows for the evaluation of network coding based protocols. It is easily extensible and allows for high-performance simulation in complex networks.

The remainder of the paper is organized as follows. *Section 2* provides an overview of the main features of NECO, including use cases. An overview of the simulation engines, including the main abstractions and internal representation of structures, is given in *Section 3*. Development choices such as the language and libraries used, as well as the licence of the simulator, are provided in *Section 4*. *Section 5* provides an in-depth overview of the structure of the simulator, as well as implementation details, followed by possibilities of extension in *Section 6*. *Section 7* provides an usage example, followed by conclusions and further work in *Section 8*.

2. FEATURES AND USAGE

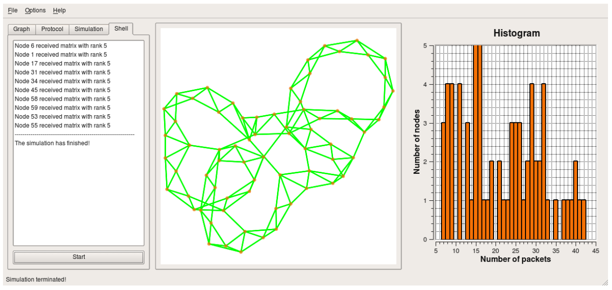
As mentioned above, NECO is aimed at the evaluation of network coding based protocols. The capabilities of NECO can be sub-divided in two groups: (a) pre-implemented features and (b) extensions, such as external plugins. A typical usage of the simulator, both in graphical and text modes, can consist of the generation of a graph and selecting sink(s) and source(s), followed by the determination of the routing and network coding protocol, and the visualization of the network operation in real-time. The latter uses either the graphical user interface or the text output at the terminal. Simulation data can be extracted by interpreting the statistics file that is generated in a seamless fashion.

2.1 User features, interfaces and output

NECO's main pre-implemented features include:

1. Generation of random graphs and optional import of graphs in the standard *graphviz* [2] format;
2. Basic flooding and network coding protocols, among which several versions of the RLNC protocol;
3. Basic routing protocols, flooding and directed diffusion protocols;
4. Seamless saving of key statistics, which are stored in a *python cPickle* file in the form of the flexible data structure *python hash*;
5. Several different user interfaces (discussed below).

As mentioned above, NECO includes three ways of running simulations: (1) a graphical user interface (GUI), (2) a command-line option and an (3) XML file for simple setup of simulation parameters. The GUI can be used for easy debugging, visualization of graphs and verification of protocol steps, as shown in *Figure 2*.



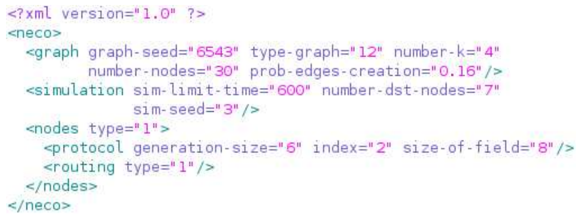
```
Node 6 received matrix with rank 5
Node 1 received matrix with rank 5
Node 17 received matrix with rank 5
Node 31 received matrix with rank 5
Node 34 received matrix with rank 5
Node 45 received matrix with rank 5
Node 58 received matrix with rank 5
Node 59 received matrix with rank 5
Node 53 received matrix with rank 5
Node 55 received matrix with rank 5
------------------------------------------------------------
The simulation has finished!
```

Simulation terminated!

Figure 2: The GUI of NECO is divided in three main components. At the left is the control component, in which the simulation parameters can be controlled beforehand, and the simulation output is updated when the simulation is running. The network visualization part is located in the middle; protocols can be easily followed since the links in usage and the ones in which information has been transmitted are highlighted using different colours. An histogram is located at the left, which shows the influence of the topology in network coding protocols.

The command line option shows the same output as the one shown in the shell shown in *Figure 2*.

The setup of simulation parameters can also be performed through an XML description. This is available both through the GUI and the command line option. It is also possible to save the current simulation parameters to a new XML file. This file can be created by following a simple grammar which is exemplified in *Figure 3*.



```
<?xml version="1.0" ?>
<neco>
  <graph graph-seed="6543" type-graph="12" number-k="4"
         number-nodes="30" prob-edges-creation="0.16"/>
  <simulation sim-limit-time="600" 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>
```

Figure 3: XML grammar for loading and saving simulation parameters with NECO.

2.2 Development of extensions

NECO also accounts for the possibility of extensions, which allow a developer to add functionalities (such as new protocols, routing protocols, types of nodes, etc) to the simulator without having to modify the core of the simulator. The interested user can simply create the classes corresponding to the extensions and add relevant information to an XML file which describes the functionalities of the simulator. The GUI is also automatically updated. For in-depth information, see *Section 6*.

2.3 Statistics

NECO contains methods for saving important statistics, which can be found 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 packets that each node received at each simulation step.

3. SIMULATION ENGINES

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.

3.1 Underlying graph

The network is represented by a graph, which can be a real topology described in *Graphviz* format [2] or a random graph generated on the spot using NetworkX algorithms. Random graphs are widely accepted [21] as representative models of typical topologies in several types of networks.

3.2 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 4*. 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, and proceed to the *outBuffer*.

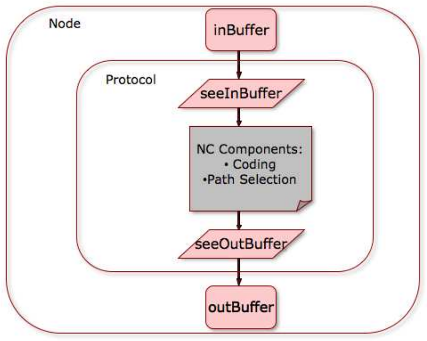


Figure 4: A simplified stack for network coding protocol testing.

In order to save processing time in simulating network coding protocols, we exploit the fact that all quantities of interest can be either directly measured or computed from the encoding matrices present at each node, and focus entirely on the encoding vectors [18, 16] of each packet, that is, the payload of the packets is not included in the simulation.

4. DEVELOPMENT CHOICES

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

4.1 Language

We chose the Python programming language [7] for several reasons. The first argument is tied to the high legibility of the language, which leads to reduced development time and high productivity, as well as improving program maintenance and extensibility. The second main reason is the platform universality: Python is available for the main operating systems without requiring changes in code. Python is also used extensively for scientific applications, and offers two useful mathematics and engineering libraries – SAGE [8] and Pylab [3]. Finally, NS-3 is expected to implement Python bindings, which can be a plus when developing more complex protocols.

In particular, we use an implementation of Python - Python stackless [10], an enhanced version of the Python programming language, which combines the benefits of thread-based programming without the performance and complexity issues of conventional threads. The objectives are more realistic concurrency and mitigating the effects of using a scripting language to provide higher speed in the simulation. The version of Python stackless is 2.5.2.

4.2 Libraries

All the libraries used in the development of the simulator are open source and available under the GPL licence (GNU General Public Licence). In particular, creation, manipulation and study of random graphs, all use the NetworkX library version 0.35.1 [5]. The SAGE library version 3.0.5 [8] 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.

4.3 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 RLNC, flooding, graph generation, scheduler and user interfaces), and *extension modules*, which extend its basic functionality, for example, to account for more elaborate protocols. This division is illustrated in *Figure 5*. It is possible to implement extension modules without interfering with core modules. We release the *core modules* of NECO under the GNU General Public Licence [1]. The licence of the extension modules is left to the choice of the institutions that implement them.

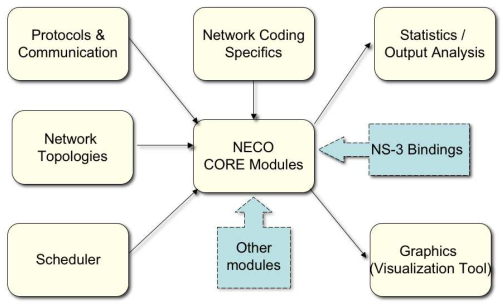


Figure 5: Main modules of NECO. The *core* includes the barebones of communication and protocols, graph generation, user interfaces and scheduler. The extensions (shown in blue with dashed border) can include other modules or extensions of the core modules, as well as possible bindings to external simulators or libraries.

5. SIMULATOR IMPLEMENTATION

The NECO code is divided into two main modules: *core* and *ui*, which are explained in depth in the following subsections. The complete documentation for the NECO source code is available at the NECO wiki, at [4].

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5646
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5646

5.1 Core

The *core* module includes the control of all the simulation steps and of other modules, as well as the basic components for building and using networks and protocols. 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*.

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 [5].

NecoCore is responsible for the communication with the graphical user interface and for starting the simulation, which is done by calling the methods in the *Simulator* class. This class includes methods that communicate to the scheduler and update the statistics. Hence, it is also responsible for starting the simulation thread, which is specified in the *SimulationThread* class and runs on a separate system thread from the simulation.

The *SimulationThread* is also responsible for randomly assigning the source and destination nodes, and for starting the stackless microthreads. 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.

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 transmitted packet.

It is worthwhile to emphasize that, as previously mentioned, the neco *core* uses Python stackless, not just for its speed, but mainly for its use of concurrency. Due to the in-existence of a limited stack, this approach allows for the benefits of thread-based programming without the complexity associated with conventional threads. Stackless provides the use of *microthreads*, which allow python to change context very quickly and still individually manipulate the thread that runs at a given time. This granularity of control is essential for the simulator, since system threads' priorities are controlled by the operating system.

5.2 User Interface

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

Two "dummy" classes, *SharedSimInfo* and *UpdateUIInfo*, are aimed at 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 generation size (that is, the number of packets that are mixed using RLNC), 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, both in the graphical setting and in the command line interface.

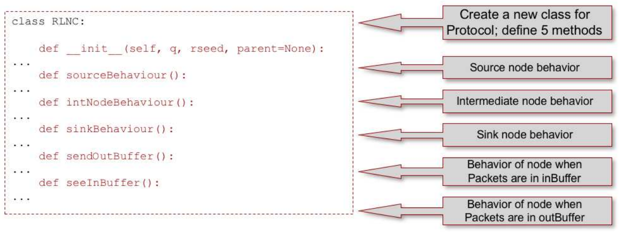
6. EXTENSION PLUGINS

We now briefly describe the implementation of extension modules for NECO. NECO can load extension modules through a description present in an XML file, *initialization.xml*. This XML file follows the grammar which is exemplified and briefly explained in *Figure 6*. Besides extending the core of NECO, by indicating the source file and the class name of each new plugin, the same XML file is used to seamlessly update the GUI.

```
<?xml version="1.0" ?>
<neco>
  <newattribute id="0" name="New attribute" variableName="xpto"
                maxValue="10" minValue="1" defaultValue="5"/>
  <protocol id="0" name="Flooding" src="flooding" className="Flooding">
    <attribute id="0"/>
  </protocol>
</neco>
```

Figure 6: 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 *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 user should then create the corresponding classes with the implementation of the new module, by extending existing classes or creating one from scratch. A brief example is shown for creating a new protocol. The methodology for creating other modules is similar.



```
class RLNC:

    def __init__(self, q, rseed, parent=None):
...
    def sourceBehaviour():
...
    def intNodeBehaviour():
...
    def sinkBehaviour():
...
    def sendOutBuffer():
...
    def seeInBuffer():
...
```

Create a new class for Protocol; define 5 methods
Source node behavior
Intermediate node behavior
Sink node behavior
Behavior of node when Packets are in inBuffer
Behavior of node when Packets are in outBuffer

Figure 7: Implementation of a new network protocol.

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 7*. Regarding the remaining elements, all routing protocols extend the *Routing* class, and override the method *selectNextHops*. Likewise, all new links extend the *Link* class, and override the *sendBehaviour* method to specify the new link behavior. All packets extend the *Packet* class.

7. USE CASE

We now present one simple use case, which demonstrates the capabilities of the simulator. To show the role that topology plays in RLNC, we run the RLNC algorithm for two different graph models and plot throughput metrics.

We represent the network as a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. We choose to simulate two representative random graph models. The first one, the Érdös-Rényi $ER(n, p)$, is formed by creating a link between each of the n nodes in the graph, independently with probability p. The Random Geometric Graph $RGG(n, r)$ can be constructed by placing n nodes uniformly at random onto the surface of a unit square, and connecting all nodes within Euclidean distance r of each other.

We then run the RLNC protocol described in *Algorithm 1*. We evaluate important throughput-related metrics for network coding, which are relevant for evaluating protocols that require a fast dissemination of information over a certain period of time, such as, for example, low-delay and immediate decoding protocols. These are the number of symbols that a node is able to decode, the rank and the number of packets that a node receives until a certain time.

The general methodology used for performing the simulations is summarized in *Algorithm 2*. To average the results and diminish the choice of a favorable (or unfavorable) pick of scenarios, the method of *independent replications* from [15] is used. We generate 5 instances of each simulation set, with different seeds, and then, for each random graph, we run 4 instances with different sources and sinks. For each of these sub-instances, we then run the RLNC algorithm with 2 different seeds. This choice still yields a total of 40 simulation runs for each parameter of each graph. We choose to use the min-cut (M_c) of the graph as the generation size for RLNC, since packet losses would introduce unnecessary confusion in the parameters that we choose to evaluate. We also choose to have only one sink: since the metrics we evaluate are influenced by the distance between the source and the sinks, the only sink to influence the results would be the one at maximum distance from the source. The field size chosen for the RLNC protocol is 2^8, which is used by most protocols.

In order to avoid situations in which the source is not connected to the sink, we choose the parameters for each type of graph so as to obtain a connected graph. We choose the same number of nodes for each graph, 100: while this number is enough to observe the essential properties of each random graph, the key parameter is the density of links in the graph, which is determined, for example, by the ratio p/n in the case of the ER graph, and by the ratio r/n in the case of the RGG graph. We choose $p = \{0.2, 0.4, 0.6, 0.8\}$ for the ER graph and then, in order to be able to perform a direct comparison, we choose the radius for the RGG graph such that the expected number of neighbors of each node is the same in both types of graphs. That is, in the case of the $ER(n,p)$ graph, $E(|(v,i)|, \forall i)_{ER(n,p)} = np$ and in the case of the $RGG(n,r)$ graph, $E(|(v,i)|, \forall i)_{RGG(n,r)} = \pi r^2 n$. This yields a choice of $r = \{0.25, 0.36, 0.44, 0.50\}$, for the RGG graph, without considering torus effects. The simulation results for the described setting are illustrated in *Figure 8*. It is clearly visible that in the RGG graph, there are peaks at which the network is flooded with packets over a short period of time, and, consequently, the average rank and number of packets decoded increases. This is due to the fact that connectivity in RGGs is determined by the distances between nodes, which leads to a higher likelihood of cliques and bursty dissemination. On the other hand, in the ER graph, because links are equiprobable, the distribution of packets and ranks is closer to uniform.

8. CONCLUSIONS AND FURTHER WORK

We presented an open-source network coding simulator, to the best of our knowledge the first of its kind, which features a high performance and easily extensible core. We also showed a simple use case that exemplifies how our simulator can provide insights into the benefits of network coding protocols. As part of our short-term future work, we expect to implement node behavior and perform a complete reformulation of the simulator scheduler. Our long-term plans include the addition of more graph models, such as evolving networks for evaluation of distributed storage, peer to peer models and mobility models.

9. ACKNOWLEDGEMENTS

Part of this work was carried out with assistance of financial support from the European Community under grant FP7-INFSO-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. The authors gratefully acknowledge insightful discussions with João Vilela, Rui Prior, Pedro Brandão and João Mendes from Universidade do Porto, Portugal, Jay K. 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, for their insightful suggestions.

Algorithm 1 RLNC protocol

```
1: Initialization (source node $s$): $s$ forms the message packets $w_1, w_2, ..., w_h$ according to the same rules that the intermediate nodes use. $h$ corresponds to the min-cut of the network.
2: Operation at intermediate node $v$:
3: if packet received then
4:     Gaussian elimination is performed with the packets already in the buffer.
5:     for all outgoing edges $i$ do
6:         Node $v$ chooses all the packets $p_1, p_2, ..., p_L$ that are in his buffer.
7:         Form packet $x_i := \sum_{l=1}^{L} \alpha_l p_l$, where $\alpha_l$ is chosen according to a uniform distribution over the elements of $F_q$. The packet's global encoding vector $\gamma$, which satisfies $x := \sum_{k=1}^{K} \gamma_k w_k$, is placed in its header.
8:         Send packet $x_i$.
9:     end for
10: end if
11: Decoding (sink nodes):
12: if packet received then
13:     Gaussian elimination is performed with the packets already in the buffer.
14:     if inverse of the matrix $M^I$ exists then
15:         The sink node applies the inverse to the packets to obtain $w_1, w_2, ..., w_K$; otherwise, a decoding error occurs.
16:     end if
17: end if
```

Algorithm 2 Simulation Methodology

```
1: for all random graph models do
2:     for all parameters $par \in parameter\_set$ do
3:         for $i$ in range(0,5) do
4:             Generate random graph $G$ with $seed_i$
5:             for $j$ in range(0,4) do
6:                 Select source $s$ with $seed_j$ uniformly at random from $V$ and sink $t$ with $seed_j$ uniformly at random from $V \backslash \{s\}$
7:                 Obtain min-cut $h = M_c(s,t)$ of $G$
8:                 for $l$ in range(0,2) do
9:                     Run original RLNC algorithm with $seed_l$ and generation size $h$
10:                    Stop when sink $t$ has decoded.
11:                end for
12:            end for
13:        end for
14:    end for
15: end for
```

10. REFERENCES

[1] Gnu General Public Licence. *http://www.gnu.org/copyleft/gpl.html*.

[2] Graphviz. *http://www.graphviz.org/*.

[3] Matplotlib and Pylab – Matlab style python plotting. *http://matplotlib.sourceforge.net/*.

[4] NECO: NEtwork COding simulator. *http://neco.dcc.fc.up.pt/*.

[5] NetworkX – High productivity software for complex networks. *https://networkx.lanl.gov/wiki*.

[6] Opnet Simulator. *http://www.opnet.com/*.

[7] Python Programming Language. *http://www.python.org/*.

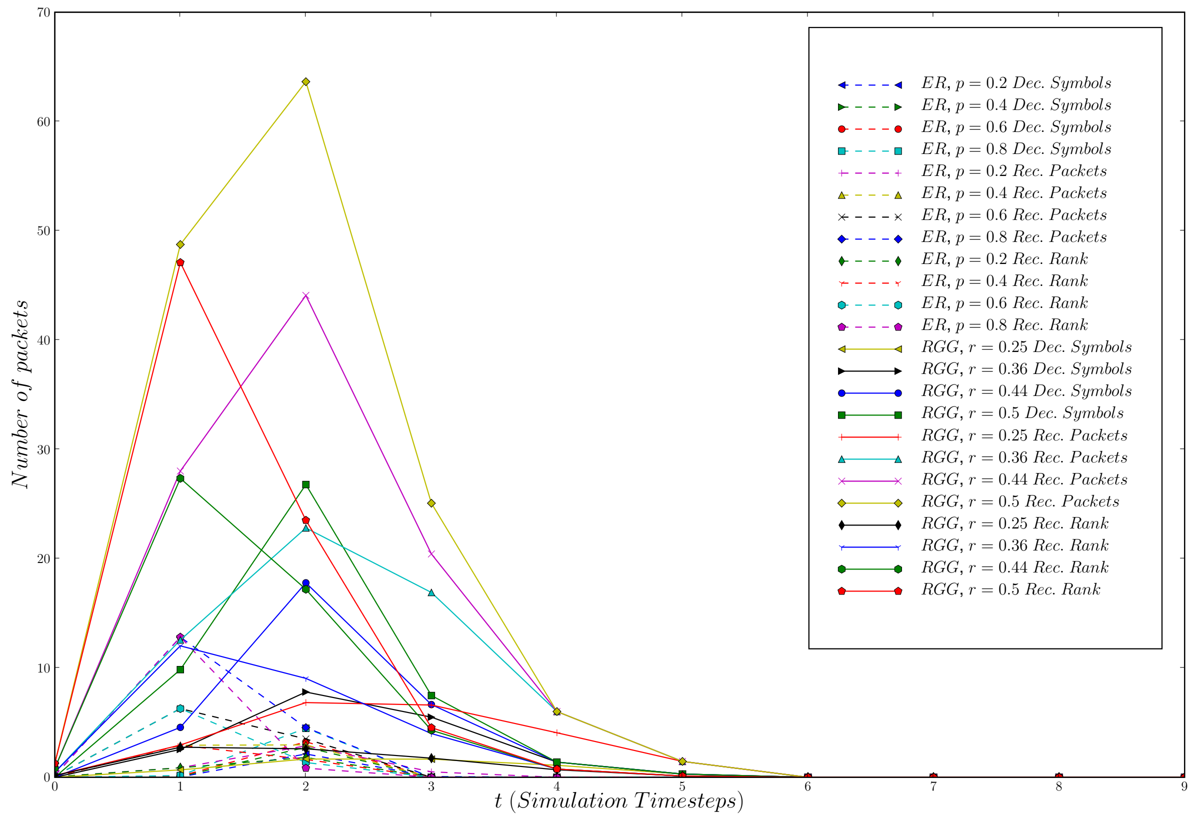


Figure 8: Illustration of the evolution of the number of received packets, rank of the encoding matrix and number of effective symbols decoded, averaged over all nodes in the network, for the ER and RGG graphs. Each quantity is computed by obtaining the difference between the quantity at timestep t and timestep $t - 1$.

[8] Sage: Open Source Mathematics Software. *http://www.sagemath.org/*.

[9] SlimSim - The Wireless Network Coding Network Simulator. *http://cs.anu.edu.au/ aaron/sim.php*.

[10] Stackless Python. *http://www.stackless.com/*.

[11] The Network Simulator - ns-2. *http://www.isi.edu/nsnam/ns/*.

[12] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, July 2000.

[13] A. Dimakis, P. Godfrey, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *26th IEEE International Conference on Computer Communications (INFOCOM 2007)*, pages 2000–2008, May 2007.

[14] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proceedings of IEEE Infocom*, Miami, Florida, May 2005.

[15] D. Goldsman and G. Tokol. Output analysis: output analysis procedures for computer simulations. In *WSC '00: Proceedings of the 32nd conference on Winter simulation*, pages 39–45, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[16] T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, October 2004.

[17] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. Xors in the air: practical wireless network coding. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 243–254, Pisa, Italy, 2006. ACM Press.

[18] R. Koetter and M. Médard. An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, 11(5):782–795, 2003.

[19] S.-Y. R. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[20] D. Lun, M. Médard, R. Koetter, and M. Effros. Further results on coding for reliable communication over packet networks. *IEEE International Symposium on Information Theory (ISIT)*, pages 1848–1852, 2005.

[21] M. Newman. Random graphs as models of networks. *Arxiv preprint cond-mat/0202208*, 2002.

[22] J. Widmer and J.-Y. L. Boudec. Network coding for efficient communication in extreme networks. In *WDTN '05: Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 284–291, Philadelphia, Pennsylvania, USA, 2005. ACM Press.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5646
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5646