Getting Kodo: Network Coding for the ns-3 Simulator

Néstor J. Hernández M.
Steinwurf ApS, Aalborg University
Aalborg, Denmark
nestor@steinwurf.com

Morten V. Pedersen
Steinwurf ApS
Aalborg, Denmark
morten@steinwurf.com

Péter Vingelmann
Steinwurf ApS
Dunaújváros, Hungary
peter@steinwurf.com

Janus Heide
Steinwurf ApS
Aalborg, Denmark
janus@steinwurf.com

Daniel E. Lucani
Aalborg University
Aalborg, Denmark
del@es.aau.dk

Frank H. P. Fitzek
Techn. Universität Dresden
Dresden, Germany
frank.fitzek@tu-dresden.de

ABSTRACT

Network Coding (NC) has been shown to improve current and upcoming communication systems in terms of throughput, energy consumption and delay reduction. However, today's evaluations on network coding solutions rely on home-grown simulators that might not accurately model realistic systems. In this work, we present for the first time the steps to use Kodo, a C++11 network coding library into the ns-3 simulator and show its potential with basic examples. Our purpose is to allow ns-3 users to use a flexible and reliable set of network coding functionalities together with the technologies simulated in ns-3. Therefore, in this paper we (i) show how to set up the Kodo library with ns-3, (ii) present the underlying design of the library examples, and (iii) verify the performance of key examples with known theoretical results.

CCS Concepts

•Networks → Network simulations; *Packet-switching networks;* **•Mathematics of computing → Coding theory; •Computing methodologies → Simulation tools; •Software and its engineering → Software libraries and repositories;**

Keywords

Network Coding, C++, ns-3, simulator

1. INTRODUCTION

Since its inception, network coding [14] has been a disruptive technology that allows intermediate network nodes to combine packets, instead of just routing them, resulting in increased throughput, reliability, and lower delay. NC implementations have also corroborated these promised gains under specific scenarios [15, 23, 24, 27, 29, 30].

In most previous implementations, the Kodo C++11 network coding library [31] was used. Kodo is intended to make network coding implementations available to both researchers and commercial entities, in particular those developing protocols. Kodo provides fast implementations of finite field arithmetics and the encoding, decoding and recoding fnctionalities for a variety of network codes, including Random Linear Network Coding (RLNC) [22], Perpetual [21] and Fulcrum network codes [28]. The library is continuously tested to support a large number of operating systems, compilers and architectures with hardware acceleration (SIMD) [6]. Hence, Kodo has been designed to ensure performance, testability and flexibility.

An important part of the evaluation process for these protocols is the simulation stage that aids developers to verify analytical results, rethink the modeling process by including unobserved system effects or proceed with a given design. Through the research community, the ns-3 project [8] aims to develop and establish an open network simulation environment for research. Among the project's goals are: simulation of standard technologies, simple usage and debugging, code testing and documentation that caters to the needs of the simulation workflow. Although there has been various initiatives to develop simulations tools in the network coding environment, [1, 9, 12, 18], most of these simulators: (i) may not be continuously maintained and tested, (ii) may rely on former functionalities of its components and (iii) are hard to integrate with standard technologies. Thus, to date there are no accurate network coding libraries that are well-tested and maintained to interact with deployable network simulation environments. Hence, in this work we provide for the first time, a set of examples compliant with ns-3 using Kodo as an external library for network coding where we verify know and expected results from the NC literature.

Our work is organized in the following way: Section 2 provides the theoretical aspects regarding the encoding, decoding and recoding of RLNC packets indicating some application scenarios. Section 3 shows how to get the Kodo library for ns-3 in an easy and rapid fashion. Section 4 describes the design and implementation details of our examples. Section 5 provides known verifiable results in the NC literature using several ns-3 simulations to validate the examples. Final conclusions of our work are drawn in Section 6.

WNS3, June 15-16, 2016, Seattle, WA, USA
© 2016 ACM. ISBN 978-1-4503-4216-2/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2915371.2915389

2. NETWORK CODING BASICS

Kodo implements core functionalities of intra-session NC (i.e., where data packets from a single flow are combined with each other). In this type of network coding, the original data $P_j, j \in [1, g]$, each of B bits, is used to create coded packets. In the following subsections, we describe the basic functionalities of RLNC [22], namely encoding, decoding and recoding. Later, we mention applications that could potentially benefit from including RLNC as a coding scheme. More complex code variants available in Kodo are described in more detail in [3].

2.1 Encoding

With RLNC, each coded packet is a random linear combination of the original set of packets. Hence, a linearly independent (l.i.) set of g coded packets, $C_i,\ i \in [1, g]$ is required in order to get the original information. Each original packet is considered as a concatenation of elements from a Galois Field (GF) of a given size q, which we denote $GF(q)$. To create a coded packet, a coding coefficient $v_{i,j}$, is chosen at random from $GF(q)$ for every packet P_j and multiplied and added following the respective GF arithmetics. In this way, a coded packet is:

$$C_i = \bigoplus_{j=1}^{g} v_{i,j} \otimes P_j,\ \forall i \in [1, g] \tag{1}$$

To indicate which packets were used to generate a coded packet, one form is to append its coding coefficients. In this case, the overhead included for $C_i,\ \forall i \in [1, g]$ by the coding coefficients is given by:

$$|v_i| = \sum_{j=1}^{g} |v_{i,j}| = g \times \lceil \log_2(q) \rceil \text{ [bits]} \tag{2}$$

2.2 Decoding

To perform decoding, we define $\mathbf{C} = [C_1 \dots C_g]^T$ and $\mathbf{P} = [P_1 \dots P_g]^T$. Then, decoding reduces to solve the linear system $\mathbf{C} = \mathbf{V} \cdot \mathbf{P}$ using Gaussian elimination [19]. Here, the coding matrix $\mathbf{V}$ contains any set of g linearly independent packets C_i as rows as follows:

$$\mathbf{V} = \begin{bmatrix} v_1 \\ \hline \vdots \\ \hline v_g \end{bmatrix} = \begin{bmatrix} v_{1,1} & \dots & v_{1,g} \\ \vdots & \ddots & \vdots \\ v_{g,1} & \dots & v_{g,g} \end{bmatrix} \tag{3}$$

The decoder begins to compute and remove the contributions from each of the pivot elements, e.g. leftmost elements in the main diagonal of (3), to reduce $\mathbf{V}$ to reduced echelon form. In this way, it is possible to recover the original set of packets.

2.3 Recoding

Network coding allows intermediate nodes in a network to recombine (or recode) packets from their sources whether they are coded or not. In general, a recoded packet should be indistinguishable from a coded one. Thus, we define a recoded packet as R_i and its corresponding encoding vector as w_i with coding coefficients $[w_{i,1} \dots w_{i,g}]$, as follows:

$$R_i = \bigoplus_{j=1}^{g} w_{i,j} \otimes C_j,\ \forall i \in [1, g] \tag{4}$$

$$\tag{5}$$

In (4), $w_{i,j}$ is the coding coefficient that multiplies C_j, uniformly and randomly chosen from $GF(q)$. Any decoder that collects $R_i, i \in [1, g]$ linearly independent coded packets, with their respective w_i, will be able to decode the data as mentioned before.

2.4 Network Coding Applications

There are numerous situations where NC provides benefits over conventional routing schemes. Basic gain descriptions and practical use cases for network coding can be found in [19, Sections 3,4] covering various areas. Among different benefits for communications, network coding can achieve the capacity for networks with multicast flows [26], improve content distribution in peer-to-peer networks [20] or enhance throughput in conventional Transmission Control Protocol (TCP) protocols for reliable communication [25]. For distributed storage systems, network coding has found applications in scenarios where it could incur less redundancy for data protection than simple replication [16].

3. GETTING KODO FOR NS-3

In this section, we explain how to get the Kodo up and running. The procedure helps to quickly add new coding functionalities in ns-3. The project with the examples is available in [4] under a GPLv2 license and it tracks the latest stable revision of the ns-3 development repository, `ns-3-dev`, to get the most recent changes. For research purposes, Kodo uses a free research license detailed in [10].

A more detailed setup guide can also be found at [4]. A descriptive tutorial for the project is available at [5]. We strongly encourage any developer to follow the setup guide. As a reference for this guide, we assume that the ns-3 project is in the `~/ns-3-dev` folder on the developer's system.

1. To get access to Kodo, it is necessary to submit a request at [11] for a research license.
2. Build the local ns-3 repository with its examples since the Kodo examples need the ns-3 binaries in order to build itself. Execute in the local `ns-3-dev` folder:
   (a) `python waf configure --enable_examples`
   (b) `python waf build`
3. Go to ~ and clone the `kodo-ns3-examples` git repository. At this point, a confirmed license is necessary to get the Kodo dependencies.
4. Go to the new `kodo-ns3-examples` folder and configure with `python waf configure` (Kodo also uses the `waf` [13] build system) to set and compile the project and its dependencies.
5. Build the `kodo-ns3-examples` and install all the needed files for ns-3 in the `~/ns-3-dev/examples/kodo` folder with `python waf build install --ns3_path="PATH"`. In this case, `"PATH"` would be `~/ns-3-dev`.

6. Get back to ns-3 folder and build the local ns-3 project with `python waf build`. At this point, the examples should be available to run as any ns-3 simulation.

4. KODO EXAMPLES FOR NS-3

In this section we describe our design implementation and criteria for creating the examples, an overview of what do the examples simulate and the design of two helpers that provide the coding operations for the system represented in the examples. The helpers function is to serve as an interface between ns-3 and the Kodo C++ bindings [2]. These are high level wrappers for the core functionalities of Kodo.

4.1 Examples Implementation

To create our examples, we consider an approach where we perform intra-session network coding, between the application and transport layer of the User Datagram Protocol (UDP) / Internet Protocol (IP) model as shown in Fig. 1. Although other approaches apply coding between the transport and Medium Access Control (MAC) layer [15,24,27,32], we implement it below the application layer to not alter other layers within the protocol stack and keep the implementation simple.

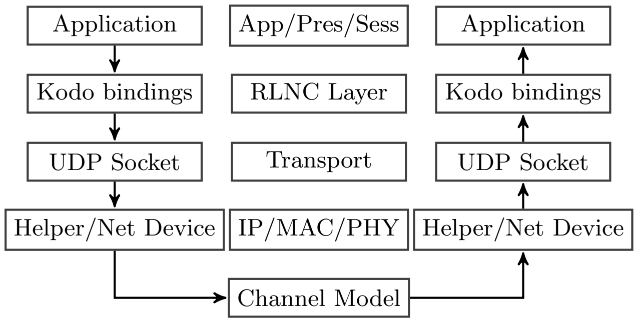

Figure 1: ns-3 + Kodo Implementation Protocol Stack based in a simple UDP/IP model.

In Fig. 1, we consider an application that generates a batch of g packets of some content. For practical reasons, we consider Hyper Text Transfer Protocol (HTTP) traffic and represent it by sending RLNC coded packets through port 80 of a UDP socket from the `UdpSocketFactory`. To encode, recode (if necessary) and decode NC packets, we employ the Application Programming Interface (API) provided by the bindings. We employ UDP datagrams because we consider best effort traffic. For the IP layer, we employ IPv4. For address assignment and routing tables, we use the `InternetStackHelper`, the `Ipv4AddressHelper` and the `Ipv4GlobalRoutingHelper` from ns-3. The details of the MAC, Physical Layer (PHY) and channel models depend on the considered example as we will see.

4.2 Examples Description

With a defined protocol stack, we describe the networks implemented in the examples to evaluate NC performance providing the details for the layers not described previously.

4.2.1 *kodo-wifi-broadcast*

This example, shown in Fig. 2, simulates a source broadcasting a generation of RLNC packets with generation size g and field size q to N sinks with an IEEE 802.11b WiFi ad-hoc channel. For the MAC we regard it without Quality of Service (QoS) implemented through `NqosWifiMacHelper`. We pick a WiFi MAC without QoS since in principle we are simulating connectionless best-effort traffic. Thus, the ns-3 net devices are constructed through the `WifiHelper`. Also, we turn off unnecessary MAC parameters, namely: frame fragmentation for frames larger than 2200 bytes and RTS / CTS frame collision protocol for the less than 2200 bytes. Although not required within the example, these parameters need to be included in order for the WiFi MAC to work.

For the PHY of this example, we use the `YansWifiPhyHelper`. The considered PHY includes a channel model that accounts for channel delay, path loss and receiver signal strength in dBm. We employ the `FixedRssLossModel` where the receiver signal (`rss`) is set to a fixed value. We set the broadcast data rate to be the same as unicast for the given `phyMode`. As a transmission policy, the sender keeps transmitting coded packets until all the receivers have g l.i. coded packets, even if some receivers are able to decode the whole generation.

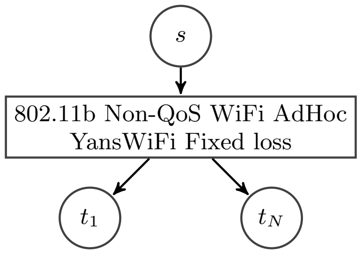

Figure 2: kodo-wifi-broadcast example network.

4.2.2 *kodo-wired-broadcast*

The example shown in Fig. 3, is similar as in Fig. 2 but instead, we evaluate a basic time-slotted wired system where a node either transmits or receives a single packet in a given time slot with the aid of the `PointToPointHelper`. To model a network with erasures, we consider the `RateErrorModel` for the PHY and channel model. In this case, packets sent from the transmitter could be lost or useless before arriving at the receiver. To control the amount of losses, an `ErrorRate` attribute is included at the `ReceiveErrorModel` attribute of the `RateErrorModel` to indicate the frequency of erasures within a given channel. The resulting topology is a basic representation for packet erasure networks which is akin for network coding applications. The transmission policy is the same as before. For simplicity, all devices are assumed to have the packet erasure rate, $0 \leq \epsilon < 1$. The erasure rate can be introduced as a command-line argument to set the `ErrorRate` attribute from the wired topology as we will see.

4.2.3 *kodo-recoders*

This example shows the gain of RLNC with recoding in a 2-hop line wired network consisting of a source, N recoders and a sink with different erasure rates. All the links between the sender and the recoders have the same packet erasure rate, $0 \leq \epsilon_{S \to R} < 1$. Equivalently, the packet erasure rate for the links between the recoders and the receivers is the

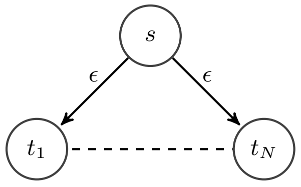

Figure 3: kodo-wired-broadcast example network.

same, $0 \leq \epsilon_{R\rightarrow D} < 1$. Again, both recoding and the erasure rates can be modified by command-line parsing. The transmission policy for this case, is as follows: First, packets are sent to each of the recoders. The transmitter stops if the decoder or all the recoders are full rank, e.g. have g l.i. coded packets. Second, a recoder retransmits packets in another scheduled time slot if l.i. packets to transmit and it stops only if the decoder is full rank.

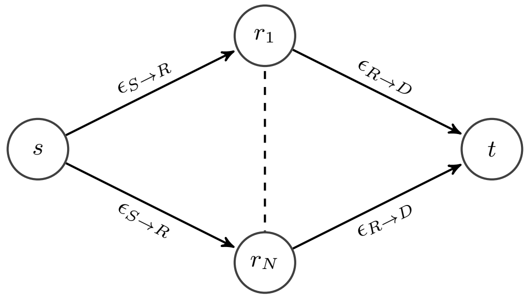
r_1
$\epsilon_{S\rightarrow R}$
$\epsilon_{R\rightarrow D}$
s
t
$\epsilon_{S\rightarrow R}$
$\epsilon_{R\rightarrow D}$
r_N

Figure 4: kodo-recoders example network.

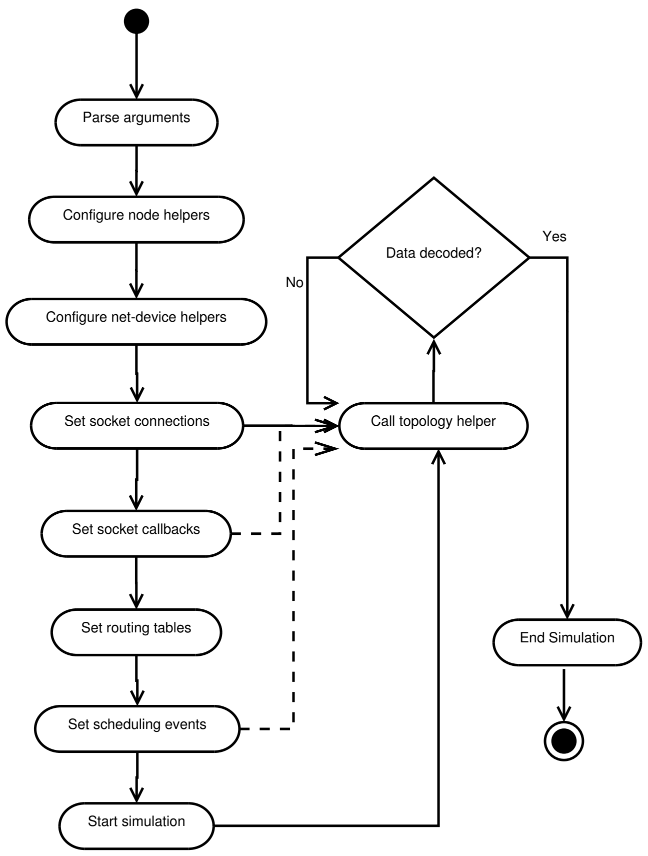

Figure 5: Examples Simulations workflow.

4.3 Simulation Workflow and Helpers

In Fig. 5, we show the workflow of the example's simulation source program. This workflow is standard for ns-3 simulations and consists in defining the network (nodes, net-devices) with ns-3 helpers according the required layer functionality described in Section 4.1. Once the socket connections are defined, we call the topology helper which provides the application and coding layers. A receive callback is set to trigger an action whenever a packet is received in a decoder socket. When an encoding or decoding action has been performed, a new event is scheduled through the `ns3::Simulator` class. Events are scheduled until a generation originated in the source is decoded by the sink(s) in the evaluated example.

At the core of each example implementation resides a topology helper which contains all the encoding, recoding and decoding parameters and functionalities of the RLNC layer, the transmission policy and eases the socket connections made in each source file. The helpers are classes that serve as interfaces between the bindings and ns-3. To accomplish this, the helpers are included in ns-3, but its basic elements are objects from the Kodo C++ bindings. For our case, we use two helpers. For `kodo-wifi-broadcast` and `kodo-wired-broadcast`, we use the `Broadcast` topology helper . For the `kodo-recoders` example, we utilize the `Recoders` topology helper. In this section we present the API of these helpers in order to show the interface between Kodo and ns-3. To do so, we elaborate an Unified Modeling Language (UML) class diagram to visualize the relationships between our bindings and the helpers. We make this review only for the `Broadcast` topology helper, since the analysis for the `Recoders` topology would be similar.

Fig. 6 shows the UML class diagram for the `Broadcast` topology. We have indicated the most important classes that have a type of dependency with the bindings. Also, we employ the UML package notation to indicate the namespace where all the bindings reside. We describe the topology members where the links with `kodocpp` occur. Later, we give an overview of other members whose type are natively contained in the C++ standard library or ns-3. We list the members with dependency on `kodocpp` according to their functionality.

4.3.1 *Code Parameter Members*

First, `m_codeType` stands for the type of erasure correcting codes utilized. In our implementation, an instance of `kodocpp::codec` is passed to the source program. The available codecs in the bindings are: `full_vector`, `on_the_fly`, `sliding_window`, `sparse_full_vector`, `seed`, `sparse_seed`, `perpetual`, `fulcrum` and `reed_solomon`. A complete description of each codec can be found in the overview section of the Kodo documentation [3]. Second, `m_field` indicates the finite field of the coding scheme. An instance of `kodocpp::field` is passed to the source program. For the available fields: `binary`, `binary4` and `binary8` represent $GF(2)$, $GF(2^4)$ and $GF(2^8)$ respectively.

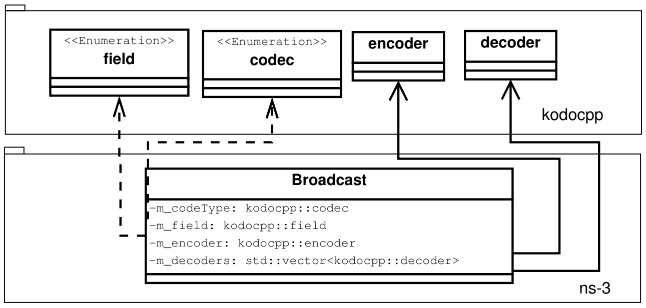

Figure 6: UML class diagram for the Broadcast topology helper interface.

4.3.2 *Encoder / Decoder Object Members*

The encoder data type is `kodocpp::encoder`, which is provided by the bindings. However, the encoder is not aware of the topology on its own, thus the uni-directional association link to indicate this in Fig. 6. The encoder class is a child class of the more general `kodocpp::coder` abstract class. In this way, the encoder class contains both own and general functionalities, inherited from `kodocpp::coder`, to configure its basic parameters and generate coded data. Similarly, we employ `std::vector<kodocpp::decoder>` to get a local decoder instance for each sink socket. As before, it contains functionalities to configure itself, read coded data and signal when to stop transmissions.

4.3.3 *Sockets and Transmission-State Members*

For packet transmissions and receptions, we use the native `ns3::Ptr<ns3::Socket>` class. Only the policies for packet transmissions/receptions are implemented through the methods `SendPacket` and `ReceivePacket`. Both of them receive the intended socket for tranmission or reception. In case of the transmitter, the packet interval time (`ns3::Time pktTime`) is also given because this will indicate the transmitter the scheduling time for next transmissions. Finally, other members like the number of users to serve, generation, packet sizes and storage buffers are considered too as standard types.

5. SIMULATIONS

To verify the accuracy of the results provided by the examples, we execute a set of ns-3 simulations to observe the behavior of RLNC in well-known scenarios. For the simulations, we compute the distribution of the number of transmissions required to decode a set of g packets with RLNC.

We only consider this metric since, typically, the time cost for the encoding and decoding operations is much lower when compared to the time spent in conveying the information from a transmitter to a receiver. Still, information regarding encoding and decoding speeds for RLNC can be easily obtained by running the benchmarks in [7] for a given platform. Similar benchmarks exist for other codes as well in their respective repositories. In our scenarios, we consider that an ideal feedback scheme is employed, where the source is aware when any destination has acknowledged all its required coded packets. To get this information, we simply call the bindings API required functions in the topology helpers.

We obtain the distribution in two scenarios. First, we consider the case of one transmitter-receiver pair. Second, we review the scenario of single-hop broadcast for N receivers. We examine these scenarios under two conditions, without packet erasures and with packet erasures. Hence, for the broadcast case, we regard the packet erasure distribution of receiver $j \in [1, N]$ as $Bernoulli(1 - \epsilon_j)$ where ϵ_j is the packet erasure probability. For evaluation purposes, we compute the distribution under a homogeneous packet erasure for all the receivers, $\epsilon_j = \epsilon \ \forall j$.

To accomplish this, we run the `kodo-wired-broadcast` example and get the number of transmissions required to decode the data in 10^4 runs. To get independent runs, the pseudo-random number generator is set to use the default seed and the `RngRun` parameter is changed in the `RngSeedManager` class by command line parsing.

For the single transmitter and receiver, we use the following parameters: `users` $= 1$, `generationSize` $= 30$, `errorRate` $= 0, 0.1$, and `field` $=$ `binary`, `binary8`. For the broadcast case, we evaluate with `users` $= 10$, `generationSize` $= 50$, `errorRate` $= 0.1, 0.2$, and `field` $=$ `binary`, `binary8`. To verify our simulations, we compare the practical results with analytical ones. To do so, we compute the Probability Mass Function (pmf) as [33, Eqs. 11-12] for the single receiver and [17, Eq. 3, Sec. III-B] for the broadcast case. Then, we plot the pmf of the analytical distributions against the simulation results.

5.1 RLNC Probability Mass Function

Fig. 7 shows the result fors the pmf of the single receiver for the evaluated parameters. We present the results for $g = 30$, $\epsilon = [0,\ 0.1]$ with $GF(2)$ and $GF(2^8)$ to observe the effect of linear independence in packets transmissions. We also evaluate the consequences of packet erasures in the number of transmissions required for decoding. In all the results, it can be clearly seen that the analytical calculations matches the simulations obtained from ns-3. For the case of no erasures, employing RLNC with $GF(2)$ requires more transmissions compared with $GF(2^8)$ since the possibilities for selecting the coding coefficients are much reduced for the last packets. For the erasure case, the transmissions alos increase given that packets might be lost regardless of linear dependency, but still are less that when employing a higher field size.

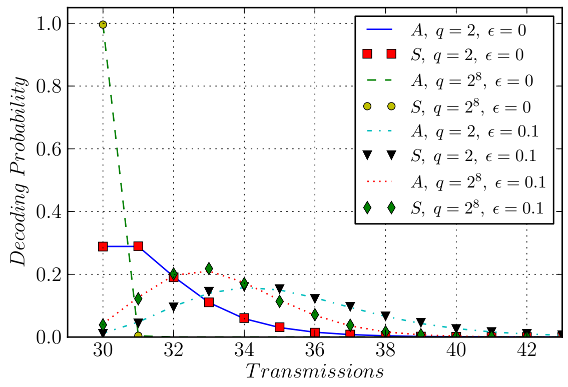

Figure 7: Analytical (A) vs. Simulation (S) for Unicast with 1 receiver and 30 packets.

5.2 RLNC Broadcast Probability Mass Function

Fig. 8 shows the result for the pmf of broadcast with RLNC for the case of 10 receivers and the evaluated parameters. In this scenario, $g = 50$. The selected fields are $GF(2)$ and $GF(2^8)$. We present the results for two erasure rates in all the links, $\epsilon = [0.1, 0.2]$.

Again, we observe the theoretical computations fit the simulations results. A difference that can be noticed with the single receiver case is the number of transmissions required to decode increases much more. Excluding the field and the erasure effects, the difference arises from all the receivers being required to get g l.i. coded packets in order to be able to decode. This is the main reason why the pmfs do not start to show a significant non-zero probability of decoding at g transmissions and shortly afterwards.

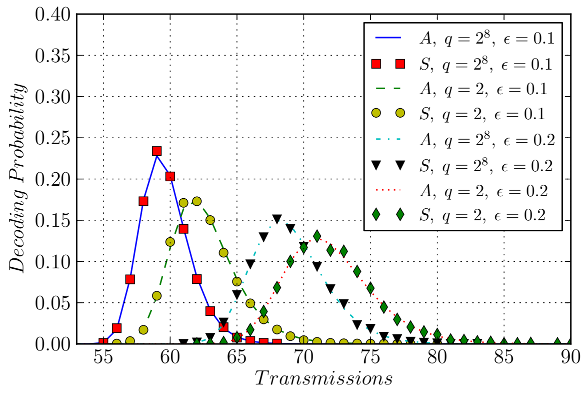

Figure 8: Analytical (A) vs. Simulation (S) for Broadcast with 10 receivers and 50 packets.

6. CONCLUSIONS

Given the increasing amount of NC applications from both academia and industry, we introduced a framework for using the Kodo library with ns-3. We hope that our contribution helps to cover the need for NC simulation capabilities in ns-3. With a set of examples where NC provides known gains, we show that our library complies with the expected results. Although the examples are made for particular topologies, the deployment of different topologies or scenarios could be easily extended by the user as detailed in [4]. Future work will be to simulate RLNC with other technologies, such as Long Term Evolution Advanced (LTE-A) within ns-3.

7. ACKNOWLEDGMENTS

This research has been financed by the CROSSFIRE MITN Marie Curie project (317126) from the European Comission FP7 framework, the Green Mobile Cloud project (Grant No. DFF - 0602 - 01372B) and the TuneSCode project (Grant No. DFF - 1335-00125) both granted by the Danish Council for Independent Research.

8. REFERENCES

[1] Inter-session network coding simulator for matlab. http://www.mathworks.com/matlabcentral/fileexchange/53750-network-coding-simulator.
[2] Kodo c++ bindings git repository. https://github.com/steinwurf/kodo-cpp.
[3] Kodo documentation read-the-docs codecs overview. http://kodo-docs.steinwurf.com/en/latest/overview.html.
[4] Kodo examples for the ns-3 simulator git repository. https://github.com/steinwurf/kodo-ns3-examples.
[5] Kodo-ns3-examples documentation read-the-docs tutorial. http://kodo-ns3-examples.readthedocs.org/en/latest.
[6] Kodo platform support. http://steinwurf.com/kodo-specifications.
[7] Kodo-rlnc git repository. https://github.com/steinwurf/kodo-rlnc.
[8] ns-3 website. https://www.nsnam.org.
[9] Software related to network coding. http://www.ifp.illinois.edu/∼koetter/NWC/Software.html.
[10] Steinwurf research license. http://steinwurf.com/research-license.
[11] Steinwurf research license webpage. http://steinwurf.com/license.
[12] Universidad de cantabria network coding implementation on ns-3.13. https://github.com/dgomezunican/network-coding-ns3.
[13] Waf. the metabuild system webpage. https://waf.io.
[14] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.
[15] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading structure for randomness in wireless opportunistic routing. *SIGCOMM Comput. Commun. Rev.*, 37(4):169–180, 2007.
[16] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Inf. Theor.*, 56(9):4539–4551, 2010.
[17] A. Eryilmaz, A. Ozdaglar, M. Médard, and E. Ahmed. On the delay and throughput gains of coding in unreliable networks. *Information Theory, IEEE Transactions on*, 54(12):5511–5524, 2008.
[18] D. Ferreira, L. Lima, and J. Barros. Neco: Network coding simulator. ICST, 5 2010.
[19] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review*, 36(1):63–68, 2006.
[20] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *IEEE INFOCOM*, number MSR-TR-2004-80, page 12, 2005.
[21] J. Heide, M. V. Pedersen, F. H. P. Fitzek, and M. Médard. Perpetual codes for network coding. *CoRR*, abs/1509.04492, 2015.
[22] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *Information Theory, IEEE Transactions on*, 52(10):4413–4430,

2006.

[23] M. Hundebøll, J. Leddet-Pedersen, J. Heide, M. Pedersen, S. Rein, and F. Fitzek. *CATWOMAN: Implementation and Performance Evaluation of IEEE 802.11 based Multi-Hop Networks using Network Coding*, pages 1–5. IEEE Press, 9 2012.

[24] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. Xors in the air: Practical wireless network coding. *IEEE/ACM Trans. Netw.*, 16(3):497–510, 2008.

[25] M. Kim, T. Klein, E. Soljanin, J. Barros, and M. Médard. Modeling network coded tcp: Analysis of throughput and energy cost. *Mobile Networks and Applications*, 19(6):790 – 803, December 2014.

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

[27] J. Krigslund, J. Hansen, M. Hundebøll, D. Lucani, and F. Fitzek. *CORE: COPE with MORE in Wireless Meshed Networks*, pages 1–6. IEEE, United States, 2013.

[28] D. E. Lucani, M. V. Pedersen, J. Heide, and F. H. P. Fitzek. Fulcrum network codes: A code for fluid allocation of complexity. *CoRR*, abs/1404.6620, 2014.

[29] A. Paramanathan, P. Pahlevani, S. Thorsteinsson, M. Hundebøll, D. Lucani, and F. Fitzek. Sharing the pi: Testbed description and performance evaluation of network coding on the raspberry pi. In *2014 IEEE 79th Vehicular Technology Conference*, 2014.

[30] A. Paramanathan, M. Pedersen, D. Lucani, F. Fitzek, and M. Katz. Lean and mean: Network coding for commercial devices. *IEEE Wireless Communications Magazine*, 20(5):54 – 61, 2013.

[31] M. Pedersen, J. Heide, and F. Fitzek. Kodo: An open and research oriented network coding library. In *Networking 2011 Workshops*, volume 6827 of *Lecture Notes in Computer Science*, pages 145–152. Valencia, Spain, 2011.

[32] H. Seferoglu, A. Markopoulou, and K. K. Ramakrishnan. I2nc: Intra- and inter-session network coding for unicast flows in wireless networks. In *INFOCOM*, pages 1035–1043. IEEE, 2011.

[33] O. Trullols-Cruces, J. M. Barcelo-Ordinas, and M. Fiore. Exact decoding probability under random linear network coding. *Communications Letters, IEEE*, 15(1):67–69, 2011.