Kodo: An Open and Research Oriented Network Coding Library

Morten V. Pedersen, Janus Heide, and Frank H.P. Fitzek

Aalborg University, Denmark
mvp@es.aau.dk
http://cone-ftp.dk

Abstract. This paper introduces the Kodo network coding library. Kodo is an open source C++ library intended to be used in practical studies of network coding algorithms. The target users for the library are researchers working with or interested in network coding. To provide a research friendly library Kodo provides a number of algorithms and building blocks, with which new and experimental algorithms can be implemented and tested. In this paper we introduce potential users to the goals, the structure, and the use of the library. To demonstrate the use of the library we provide a number of simple programming examples. It is our hope that network coding practitioners will use Kodo as a starting point, and in time contribute by improving and extending the functionality of Kodo.

Keywords: Network Coding, Implementation.

1 Introduction

First introduced by Ahlswede et al. in [1], network coding has in recent years received significant attention from a variety of research communities. However despite the large interest most current contributions are largely based on theoretical, analytical, and simulated studies and only few practical implementations have been developed. Of which even fewer has openly shared their implementations. In this paper we introduce an open source network coding library named *Kodo*. It is the hope that Kodo may serve as a practical starting point for researchers and students working in the area of network coding algorithms and implementations. At the time of writing the initial release of Kodo supports basic network coding algorithms and building blocks, however by releasing Kodo as open source it is the hope that contributions may lead to additional algorithms being added to Kodo. Before describing the library design and functionality we will first give an overview of the motivations and goals behind the library.

Flexibility: as the target users of Kodo are researchers working with network coding, one of the key goals behind the design of Kodo has been to create a flexible and extensible Application Programming Interface (API), while providing already functional components for implementers to re-use. To enable this, the ambition is that Kodo should provide a number of customizable

V. Casares-Giner et al. (Eds.): NETWORKING 2011 Workshops, LNCS 6827, pp. 145–152, 2011.

building blocks rather than only a set of complete algorithms. Creating easy to use extension points in the library is vital to provide researchers the opportunity to use/reuse Kodo for their specific use-case or implementation. To achieve this goal Kodo relies on the generic programming paradigms provided by the C++ template system. Through the use of C++ templates implementers may substitute or customize Kodo components without modifying the existing code. This makes experimental configurations easy to create and lowers the entry level for new users.

Ease of use: to the extend possible it is the intent that Kodo should be usable also for researchers with limited programming experience. To achieve this Kodo attempts to provide a simple and clean API hiding the more complicated implementation details. To further support this Kodo will also provide programming language bindings, which will allow users of programming languages other than C++ to use the library functionality. Currently experimental bindings for the Python programming language are provided for a selected part of the API.

Performance: one of the key challenges in the design of Kodo has been to provide the flexibility and ease of use required for research while not significantly sacrificing performance of the library. One of the most active areas for network coding implementations have been investigating and improving the performance of network coding algorithms, see [5],[6],[2]. For this reason Kodo attempts to find a reasonable middle ground between flexibility and performance. In cases where these two requirements does not align, Kodo will typically aim for flexibility and ease of use over performance. The reason for this is to keep the library agile, and that high performance implementations should easily be derived from Kodo source code.

Testing: to avoid and catch as many problems as early as possible, Kodo is tested using a number of unit tests. At the time of writing most components in Kodo is delivered with accompanying test cases, and it is the goal that all Kodo components should have a matching unit test. The testing framework used in Kodo is Qt test library (QtTestLib). It was chosen due to the good cross-platform support and ease of use provided by the Qt framework. It is important to note that the Kodo library itself does not have any dependencies on Qt, and that Kodo therefore may be used on platforms not supported by the Qt framework.

Portability: it is the intention to keep Kodo as portable and self-contained as possible. In cases where platform specific dependencies are required, Kodo should provide interfaces which enable easy adaptation to that specific platform.

Contributions: it is the hope that researchers from the networking coding community will join in the development of Kodo and contribute new functionality. Kodo will be released under a research friendly license which allows everybody to contribute and use the library in their research.

Benchmarking: since Kodo is created to facilitate research of various Network Coding (NC) implementations, enabling measurements and monitoring the performance of the implemented algorithms are a priority. Having good

methods for benchmarking allows quick comparisons between performance of existing and future algorithms or optimizations. Using the C++ template system users of Kodo may instrument code to collect useful information about algorithms and data structures used. Examples of this are to monitor the number of finite field operations performed by a specific type of encoder or the number of memory access operations performed during decoding. Additionally it allows the maintainers of Kodo to ensure that no modifications or additions to the library results in unwanted performance regressions.

Following the goals and motivation behind the library the following section will introduce the functionality provided by Kodo.

2 Network Coding Support

Due to its promising potential many algorithms and protocols have been suggested to demonstrate and exploit the benefits of network coding. Use cases range over wired to wireless and from the physical to the transport and application layer. In addition to these widely different areas of application, different variants of network coding also exist. Examples of this are inter- and intra-session network coding, where the former variant operates on data from a uniquely identifiable network flow and the latter allows mixing of data from different network flows. One might also differentiate between deterministic and non-deterministic network coding algorithms. These refer to the way a computer node participating in a network coding system operates on the data traversing it. Typically in deterministic algorithms a node performs a "fixed" number of operations on the incoming data, these predetermined operations may be selected based on different information e.g. the network topology. In contrast non-deterministic network coding or random network coding operates on the data in a random or pseudo-random manner. This has certain desirable advantages in e.g. dynamic wireless networks.

The initial release of Kodo supports only a subset of these use-cases, however it is the goal that future versions will eventually support a wide range of different network coding variants. The initial features implemented have been selected to support the creation of digital Random Linear Network Coding (RLNC) systems. RLNC has in recent years received a large amount of interest for use in dynamic networks e.g. wireless mesh networks [3]. Understanding the functionality of a RLNC system will therefore also be helpful to understand the current structure of the Kodo library. In the following we therefore provide a general overview of the supported RLNC components and show the corresponding C++ code used in Kodo. The C++ programming examples will not be explained in detail here, but are there to demonstrate how a RLNC application would look, using Kodo. For details about usage of the library, the programming API and further programming examples we refer the reader to the Kodo project web page.

2.1 Random Linear Network Coding

In RLNC we operate on either finite or infinite streams of binary data. To make the coding feasible we limit the amount of data considered by segmenting it into

manageable sized chunks. Each chunk in turn consists of some number g packets, where each packet has a length of d bytes. In network coding terminology such a chunk of binary data, spanning $g \cdot d$ bytes, is called a *generation*.

One commonly used way for constructing generations from a file is illustrated in Figure 1. In this approach a file is divided into N packets, p_1, p_2, ..., p_N. Packets are then sequentially divided into $M = \frac{N}{g}$ generations, each containing g packets. In Kodo this algorithm is referred to as the *basic partitioning* algorithm.

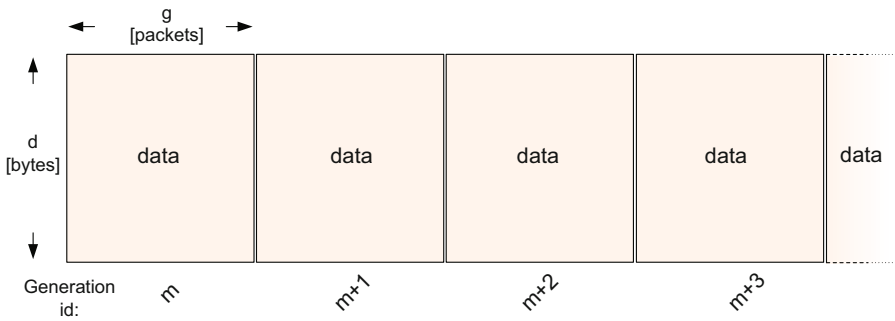

Fig. 1. Basic partitioning algorithm for slicing a file into generations of a certain size

Besides the basic partitioning algorithm Kodo also supports other partitioning schemes, such as the *random annex code* described in [4]. The following code listing shows how the basic partitioning algorithm is used in Kodo to partition a large buffer.

Listing 1.1. Using the basic partitioning algorithm

```
// Selecting general coding parameters
int generationSize = 16;
int packetSize = 1600;

// Size of the buffer
int bufferPackets = 432141;

// The buffer contaning all the packets of the file
UnalignedBuffer fileBuffer(bufferPackets, packetSize);

// Vector to hold the buffers for each generation
std::vector<ThinBuffer> generationBuffers;

// Run the basic partitioning, and add the generation
// buffers to the vector
BasicPartitioning basic(generationSize, packetSize);
basic(fileBuffer.dataBuffer(), std::back_inserter(
    generationBuffers));
```

Listing 1.1 : Initially we specify the target generation size and the desired packet size for each generation. The *fileBuffer* object represents the file, containing a large amount of packets to be distributed over different generations. This is achieved in the last line by passing file buffer to the basic partitioning algorithm. This creates a number of generation buffers and inserts them in the vector. Following this we may operate on a single generation at a time.

In addition to selecting the packet size and generation size parameters, implementers are faced with an additional choice before the coding can start, namely which *finite field* to use. In NC data operations are performed using finite field arithmetics. Performing arithmetic operations in finite fields are informally very similar to normal arithmetic with integers, the main difference being that only a finite number of elements exist, and that all operations are closed i.e. every operation performed on two field elements will result in an element also in the field. A simple example is the binary field $\mathbb{F}_2$, which consists of two elements namely $\{0, 1\}$. In the binary field addition is implemented using the bit-wise XOR and multiplication is implemented using the bit-wise AND operator. For NC applications the choice of finite field represents a trade-off between computational cost and efficiency of the coding, i.e. how fast we may generate coded packets versus how likely it is to generate linear dependent and therefore useless packets. Kodo provides several choices of finite field implementations, and also allow external implementations to be used.

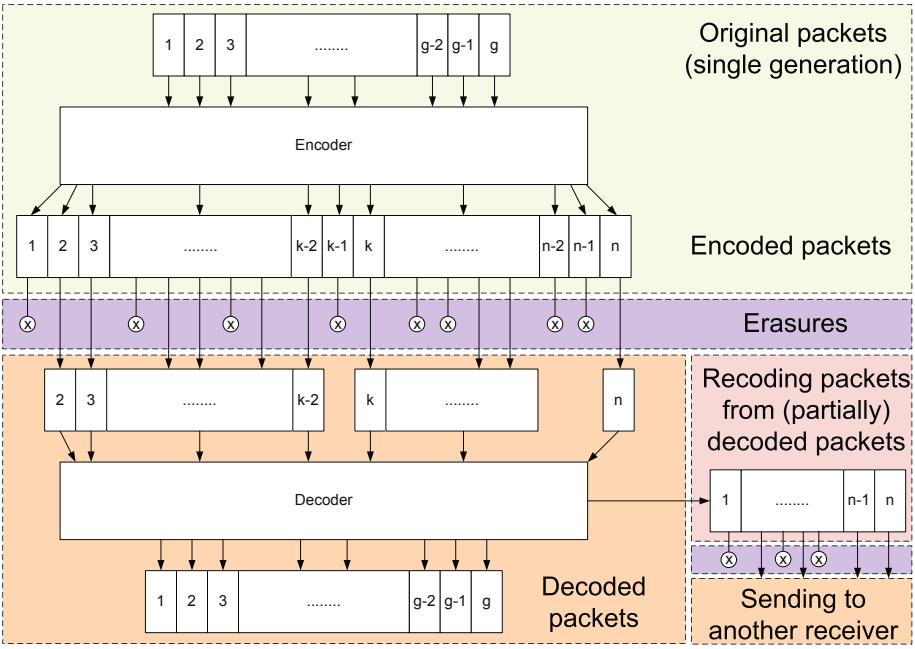

Fig. 2. Network coding system overview

After creating the buffers containing the generation data and selecting the finite field to use, the encoding and decoding processes may start. Figure 2 presents an overview of the basic operations performed for each generation in a typical RLNC system.

At the top of Figure 2 the encoder uses the g original source packets to create k encoded packets, where k is larger or equal to g. The k encoded packets are then transmitted via an unreliable channel, causing a number of packet erasures. The job at the decoder side is to collect enough encoded packets to be able to reconstruct the original data. In general this is possible as long as the decoder receives g or more encoded packets. Note, that in RLNC the number of encoded packets, k, is not fixed which means that the encoder can continuously create new encoded packets if needed. For this reason this type of coding is often referred to as *rate-less*. Also shown in the figure is one of the most significant differences between NC and traditional schemes, namely the re-coding step shown in the bottom right of Figure 2. In contrast to e.g. traditional error correcting codes, NC based codes are not necessarily end-to-end codes, but allow intermediate nodes between a sender and receiver to re-code and forward data, instead of pure forwarding.

In Kodo the encoding and decoding step for a single generation may be performed using the following example code:

Listing 1.2. Encoding and decoding data from a single generation

```
int generationSize = 16;
int packetSize = 1600;

// Create an encoder and decoder object
Encoder encoder(generationSize, packetSize);
Decoder decoder(generationSize, packetSize);

// Fill the encoder data buffer with data
fillbuffer(encoder.packetBuffer().buffer(), encoder.
    packetBuffer().size());

// Construct a vector generator
typedef DensityGenerator<Binary, RandMT> DensityGenerator;
DensityGenerator generator(0.5);

int vectorSize = Binary::sizeNeeded(generationSize);

// Make a buffer for one packet and one vector
UnalignedBuffer packet(1, packetSize);
UnalignedBuffer vector(1, vectorSize);

// Loop as long as we have not decoded
while( !decoder.isComplete() )
{
    memset(packet[0], '\0', packetSize);
```

```
    memset(vector[0], '\0', vectorSize);

    // Fill the encoding vector
    generator.fillVector(vector[0], generationSize);

    // Create an encoded packet according to the vector
    encoder.encode(packet[0], vector[0]);

    // For a simulation this would be where the erasures
    // channel could be implemented

    // Pass the encoded packet to the decoder
    decoder.decode(packet[0], vector[0]);
}
```

Listing 1.2 : The source code is a slightly modified excerpt from one of the unit tests, testing the encoder and decoder implementations of Kodo. Initially the coding parameters are specified and some buffers are prepared. We then setup a *vector generator*. The vector generator is responsible for creating the encoding vectors, in this case we are using a *density generator* which creates encoding vectors according to some specified density (i.e. number of non-zero elements in the vector). Kodo contains other types of vector generators e.g. for creating systematic encoding vectors. You may notice the word, *binary*, present in the vector generator, this denotes the finite field we are using. Binary refers to the $\mathbb{F}_2$ binary field. Kodo contains a number of different finite field implementations for different field sizes. Following this the code loops passing encoded packets from the encoder into the decoder until the decoder reports that decoding has completed.

The two source code examples shown demonstrates the code needed to implement an operational RLNC application in less than 60 lines of code. This completes the overall introduction of the Kodo functionality, to get an in-depth view on the possibilities with Kodo visit our project website (which may be found at `www.cone-ftp.dk`.

3 Conclusion

In this paper we have introduced the Kodo C++ network coding library, the goal of this paper is to provided a general overview of the components of the library and introduce its usage. In the initial release of Kodo a number of goals for the library has been specified, and it is the hope that these goals and introduction of Kodo presented here, may aid researchers in identifying whether Kodo may be useful for them in their network coding research. As shown in this paper Kodo currently support development of basic RLNC schemes and many general NC features are still missing. However, it is the aspiration that by releasing Kodo as open source, researchers will contribute to strengthen the capabilities and usefulness of the project.

Acknowledgments. This work was partially financed by the CONE project (Grant No. 09-066549/FTP) granted by Danish Ministry of Science.

References

1. Ahlswede, R., Cai, N., Li, S.Y.R., Yeung, R.W.: Network information flow. IEEE Transactions on Information Theory 46(4), 1204–1216 (2000), http://dx.doi.org/10.1109/18.850663
2. Heide, J., Pedersen, M.V., Fitzek, F.H., Larsen, T.: Network coding for mobile devices - systematic binary random rateless codes. In: The IEEE International Conference on Communications (ICC), Dresden, Germany, June14-18 (2009)
3. Ho, T., Medard, M., Koetter, R., Karger, D.R., Effros, M., Shi, J., Leong, B.: A Random Linear Network Coding Approach to Multicast. IEEE Transactions on Information Theory 52(10), 4413–4430 (2006), http://dx.doi.org/10.1109/TIT.2006.881746
4. Li, Y., Soljanin, E., Spasojevic, P.: Collecting coded coupons over overlapping generations. CoRR abs/1002.1407 (2010)
5. Shojania, H., Li, B.: Random network coding on the iphone: fact or fiction? In: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV 2009, pp. 37–42. ACM, New York (2009), http://doi.acm.org/10.1145/1542245.1542255
6. Vingelmann, P., Zanaty, P., Fitzek, F.H.P., Charaf, H.: Implementation of random linear network coding on opengl-enabled graphics cards. In: European Wireless, Aalborg, Denmark (May 2009)