Adding Network Coding Capabilities to the WSNet Simulator

Wei Liang Choo, Frédéric Le Mouël, Katia Jaffrès-Runser, Marco Fiore

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Adding Network Coding Capabilities to the WSNet Simulator

Wei Liang Choo — Frédéric Le Mouël — Katia Jaffrès-Runser — Marco Fiore

N° 0405

March 2011

Networks and Telecommunications

Rapport technique

ISSN 0249-0803 ISRN INRIA/RT--0405--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Adding Network Coding Capabilities to the WSNet Simulator

Wei Liang Choo* , Frédéric Le Mouël* † , Katia Jaffrès-Runser* ‡ ,
Marco Fiore* ‡

Theme : Networks and Telecommunications
Équipe-Projet SWING

Rapport technique n° 0405 — March 2011 — 25 pages

Abstract: This technical report presents the implementation of a Network Coding module in WSNet - a Wireless Sensor Network simulator. This implementation provides a generic programming interface to allow an easy specialization of different coding strategies: random, source/destination-oriented, intra/inter-flow, etc.

Key-words: Network Coding, Wireless Sensor Network (WSN), Simulation, WSNet

* University of Lyon, INSA-Lyon, CITI, F-69621, Villeurbanne, France, {firstname.lastname}@insa-lyon.fr
† INRIA, AMAZONES Team
‡ INRIA, SWING Team

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Implantation d'un module de codage de réseaux dans le simulateur WSNet

Résumé : Ce rapport technique présente l'implantation d'un module de codage de réseaux dans le simulateur de capteurs sans-fil WSNet. L'objectif de conception de ce module est d'avoir une interface de programmation générique pour ensuite spécialiser facilement différentes stratégies de codage: aléatoire, orienté-source/destination, intra/inter-flux, etc.

Mots-clés : Codage des réseaux, Réseaux de capteurs sans-fil, Simulation, WSNet

Contents

1 Introduction

Our goal is to study the impact of different Network Coding strategies (NC) on end-to-end service delivery over mobile and wireless Disruption-Tolerant Networks (DTNs). To realize this study, in a first step, we simulate a mobile and wireless DTN environment. This report presents (i) in section 2, the context: DTN, NC and why we have chosen the WSNet simulator, (ii) in section 3, our NC framework provided in a WSNet module: architecture, generic API definition, packet storing, linear independence checking, real encoding/decoding.

2 Context

2.1 Disruption-Tolerant Network

Disruption-Tolerant Networking (DTN) is an approach that seeks to address the non-constant nature of links in networks [2]. This could be caused by mobility of the nodes or interference in the environment.

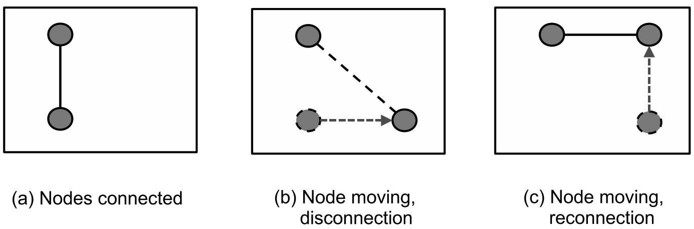

Figure 1: Disrupted Network Example

Figure 1 shows an example of DTN where, as a node moves, the link previously established is broken. The link may be established again once the node moves back into range of communicating with the original node. Conventional routing involves finding a path and forwarding packets from the source to the destination. However because of the link break, storing and then forwarding when the link is re-established may be needed - so introducing a delay. A common approach is to send out replicated packets to many nodes hoping that packets will reach the destination. However, this takes up large amounts of storage and bandwidth.

Strategies involving Network Coding are to be accessed to judge their impacts on delay/tolerance/capacity improvement of a DTN environment [11, 9, 1].

2.2 Network Coding

Network Coding (NC) is a technique where nodes of a network are able to combine together two more or received packets and transmit them [4]. With enough information - enough encoded packets, the original packets can then be decoded at the destination. This is a change from just forwarding packets which can bring about potential throughput improvements and a high degree of robustness.

Figure 2 presents the Butterfly Network Coding multicast example. S1 and S2 are Source nodes, R1 and R2 are Relay nodes, D1 and D2 are Destination

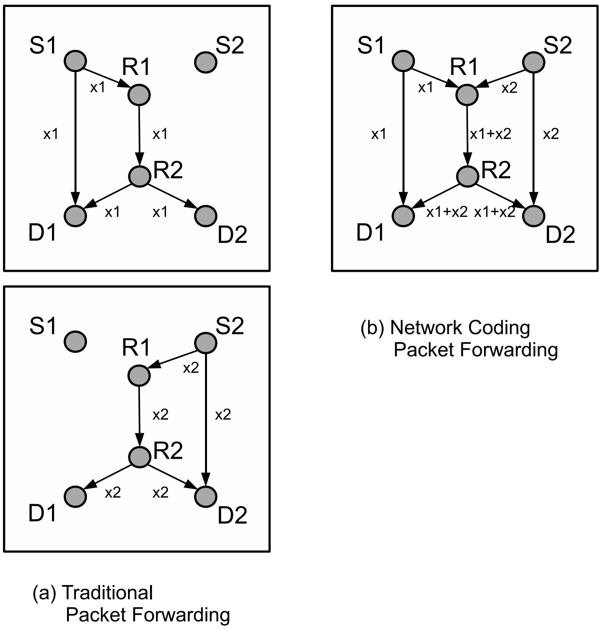

Figure 2: Network Coding Butterfly Example

nodes and x1 and x2 are packets from S1 and S2 respectively. D1 and D2 needs to receive both x1 and x2 packets. In the traditional packet sending method, x1 would be sent to D1 from S1 directly. But due to distance from D2, S1 will send x1 via R1 to R2 and then to D2. The same case is for S2 and D1. R1 forwards the whole packet x1 then x2, to R2 which then forwards it to D1 and D2. Using NC, when R1 receives both x1 and x2, R1 can combine the packets and send only one packet combining both x1 and x2 to R2 which forwards it to D1 and D2. D1 and D2 use this encoded packet to retrieve the other missing packet. In this case, NC thus helps in reducing the sending of a second packet from R1 and R2.

According to the network topologies considered (linear vs non-linear, multicast vs non-multicast, directed vs undirected, cyclic vs acyclic), different NC strategies exist to select and encode packets [7]: random, unique/multi source-oriented, unique/multi destination-oriented, intra-session, inter-session, etc. We plan to develop and test social and service-oriented NC strategies and so we need a realistic simulation environment to compare them in a mobile and wireless DTN.

2.3 WSNet Simulator

WSNet is a simulator for large-scale Wireless Sensor Networks created and developed at the CITI Laboratory [3]. While several simulator exist for DTN [10, 8], WSNet main features - Node Simulation, Environment Simulation, Radio Medium Simulation and Extensibility - are particularly suitable for our NC testing. Radio medium simulation provides realistic radio channel modeling appropriate to test wireless communication in mobile DTN. Node

Simulation allows the integration of the application level, suitable to test social and service-oriented NC strategies.

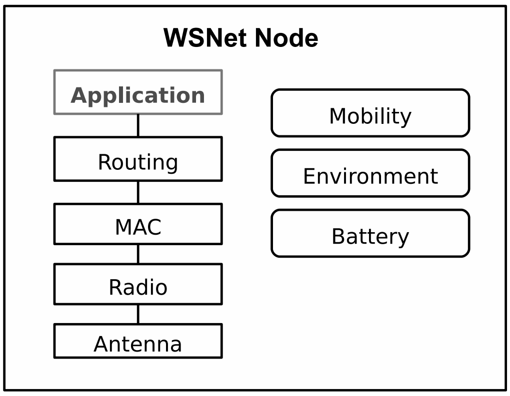

Figure 3: Modular Architecture of a WSNet Node

Figure 3 describes a node architecture that can be created in WSNet. There are already various standing modules that can be used for each part: support for complex nodes architecture (MIMO systems, multiple radio/antenna interface support), support for energy consumption simulation, support for various propagation models, support for propagation delays, etc. Modules are attached on run time and an XML file is used to control the WSNet.

3 A Generic Network Coding Module in WSNet

3.1 Module Configuration

Using the WSNet extensibility feature [5], we have developed an application module that simulates a wireless DTN with NC by storing, selecting/dropping, encoding/decoding IP packets.

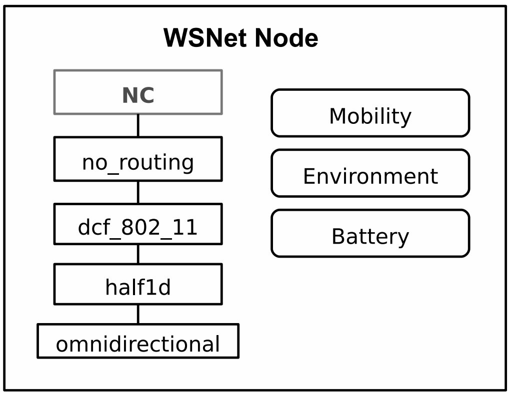

Figure 4: Network-Coding Module in WSNet

We configure and test it with different existing WSNet modules [6] (cf Figure 4 and Listing 1). No routing module has currently been used since a static one, with the Butterfly example, was applied.

```
<!-- == RADIO and ANTENNA ================================ -->
<entity name="omnidirectionnal" library="antenna_omnidirectionnal">
  <default loss="0" angle-xy="random" angle-z="random"/>
</entity>

<entity name="radio" library="radio_half1d">
  <default sensibility="-92" T_b="727" dBm="10" channel="0"
       modulation="none"/>
</entity>

<!-- == MAC ============================================== -->
<entity name="mac" library="mac_dcf_802_11">
</entity>

<!-- == APPLICATION ====================================== -->
<entity name="Esource0" library="application_DTNNC_Source">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
       "3" storageOrder="1" decodingPolicy="1" encodingType="1"
       encodingPacSelection="1" chanceOfSending="10"/>
  <default type="0" nodenum="0" period="2s" inData="15"/>
</entity>

<entity name="Esource1" library="application_DTNNC_Source">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
       "3" storageOrder="1" decodingPolicy="1" encodingType="1"
       encodingPacSelection="1" chanceOfSending="10"/>
  <default type="0" nodenum="1" period="2s" inData="15"/>
</entity>

<entity name="Esensor0" library="application_DTNNC_Sensor">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
       "3" storageOrder="1" decodingPolicy="1" encodingType="1"
       encodingPacSelection="1" chanceOfSending="100"/>
  <default type="1" nodenum="0" period="2s" inData="15"/>
</entity>

<entity name="ErelayDumb0" library="application_DTNNC_RelayDumb">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
       "3" storageOrder="1" decodingPolicy="1" encodingType="1"
       encodingPacSelection="1" chanceOfSending="100"/>
  <default type="3" nodenum="0"/>
</entity>

<entity name="Esink0" library="application_DTNNC_Sink">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
       "3" storageOrder="1" decodingPolicy="1" encodingType="1"
       encodingPacSelection="1" chanceOfSending="0"/>
  <default type="2" nodenum="0"/>
</entity>

<entity name="Esink1" library="application_DTNNC_Sink">
  <init debugMode="1" dataStruct="1" maxNumberOfCombinedPerPacket="
       30" maxNumberOfPackets="100" numberOfCoefficients="2" FPower=
```

```
        "3" storageOrder="1" decodingPolicy="1" encodingType="1"
        encodingPacSelection="1" chanceOfSending="0"/>
    <default type="2" nodenum="1"/>
</entity>
```

Listing 1: DTNNC_Butterfly_example.xml

3.2 Architecture Overview

The Figure 5 presents the architecture of the NC module. Each node includes the `functions.h` header file, entry point of the framework. This file defines (i) the common node structure, variables and data storage - detailed in section 3.3, (ii) the common masking/encoding/decoding functions - detailed in section 3.4 and (iii) useful logging functions - detailed in section 3.5.

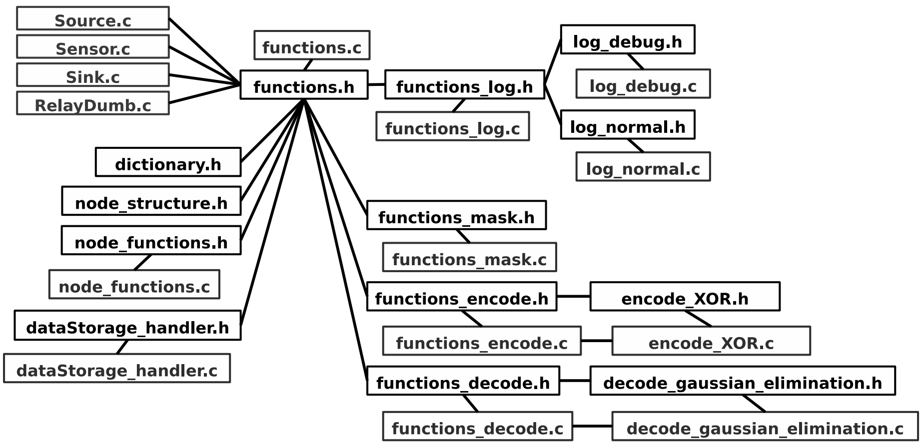

Figure 5: Network-Coding Module Architecture

3.3 Node Definition

3.3.1 Node Type

All common aspects of a node are included in `DTNNC_dictionary.h`, `DTNNC_node_structure.h`, `DTNNC_functions.h`, `DTNNC_functions.c` and `DTNNC_dataStorage_handler.h`. Each node can then be specialized and instantiated to be a:

Source

- Creates packets
- Sends packets

Sensor

- Stores received packets
- Encodes packets, including masking and xor-ing packets
- Decodes encoded packets, if possible and needed
- Sends encoded packets out

RelayDumb

- Forwards received packets

Sink

- Stores received packets
- Decodes

3.3.2 Node Common Variable Definitions: `dictionary.h`

`DTNNC_dictionary.h` defines keywords to make code more readable and easier to use.

```
#ifndef __DTNNC_dictionary__
#define __DTNNC_dictionary__

/* Node Mode */
#define SOURCE 0
#define SENSOR 1
#define SINK 2
#define RELAYDUMB 3

/* Log Mode */
// change to 1 for debug
// change to 0 for normal
#define DEBUG_MODE 1

/* Debug output prints */
#define DEBUG_MASK 0
#define DEBUG_DECODING 1
#define DEBUG_NODE_INFO 1

/* Debug LogFunction */
#define PRINT_NODE_STORAGE 40
#define PRINT_DATA_HOLDER 41
#define PRINT_RECONSTRUCTED_PACKETS 42

/* LogFunction */
#define PRINT_SEND_EVENT 1
#define PRINT_RECEIVE_EVENT 2
#define PRINT_STORE_PACKET 3
#define PRINT_DROP_PACKET_REPEAT 4
#define PRINT_DROP_PACKET_DECODED 5
#define PRINT_LINEAR_CHECK_FAILED 6
#define PRINT_LINEAR_CHECK_PASSED 7
#define PRINT_DECODED 8
#define PRINT_ENCODED 9
#define PRINT_UNSETNODE 10 // to be changed to more detailed

#endif // __DTNNC_dictionary__
```

Listing 2: `DTNNC_dictionary.h`

3.3.3 Node Structure: `node_structure.h`

`DTNNC_node_structure.h` defines the common structure of all nodes. It is easier to edit one structure for all nodes than to make a specific structure for each node type. Most variables are needed in all node types.

```
#ifndef __DTNNC_node_structure__
#define __DTNNC_node_structure__

/* Node private data */
struct nodedata {

  int *overhead;
  // Source, Sensor, Sink or RelayDumb
  int type;
  // ID of the node. i.e. node 0 of Source type
  int nodenum;
  // Flag if Node has decoded packets
  int decodedFlag;

  int seqNum;

  // period in which to recall node
  uint64_t period;

  /* temp data holder */
  struct packet_data* dataHolder;
  struct header_packets_combined **
      dataHolder_header_packets_combined;
  uint16_t dataHolder_numberOfpacketsCombined;

  int* dataHolder_args;

  /* data storage */
  // data storage type 1(basic arrays)
  struct packet_data** stored_data;
  struct header_packets_combined ***
      stored_data_header_packets_combined;
  uint16_t* stored_data_numberOfpacketsCombined;

  /* arguments storage */
  int** stored_data_args;

  /* reconstructed packets storage */
  struct packet_data** reconstructed_pack_data;
  int** reconstructed_pack_args;

  /* num of packets stored counter */
  int num_of_packets_stored;

  /* for stats */
  // Number of packets transmited by node
  int packet_tx;
  // Number of packets received by node
  int packet_rx;

};

#endif // __DTNNC_node_structure__
```

Listing 3: `DTNNC_node_structure.h`

3.3.4 Node Management Functions: `node_functions.h`

`DTNNC_node_functions.h` allows to manage a node. It creates the variables, allocates the memory for variable structures, and sets the variables to default. This API simplifies the development. Editing or adding a new node variable requires the editing and adding of this variable in each node type's source code file. With this API, it requires to be done only one time in one place.

```
#ifndef __DTNNC_node_functions__
#define __DTNNC_node_functions__

#include "DTNNC_functions.h"

// Called when setting Node Entity (from int())
int setNodeEntity(call_t *c, void *params);

// Called when setting Node Variables (from setnode())
int setNodeVariables(call_t *c, void *params);

// Called when unsetting node(from unsetnode())
// frees memory allocated
int freeNode(call_t *c);

#endif // __DTNNC_node_functions__
```

Listing 4: `DTNNC_node_functions.h`

3.3.5 Node Storage: `packet.h` `dataStorage_handler.h`

Data stored are IP packets. These packets are potentially xor-mixed packets, so a packet header includes the number and a table of sub-packet headers. A final sub-packet header contains an id (can be an application or a service id), a sequence number ordering a packet flow, a source and n destinations (for broadcast or multicast). Packet data structure contains the real data (here a dummy example with 4 characters).

```
#ifndef __DTNNC_node_dummy_packet__
#define __DTNNC_node_dummy_packet__

/* Packet header */
struct packet_header {
  struct header_packets_combined ** header_packets_combined_info;
  uint16_t numberOfpacketsCombined;
  uint16_t *args;
};

struct header_packets_combined {
  uint16_t seqNum;
  uint16_t id;
  uint16_t source;
  uint16_t numOfDest;
  uint16_t *destination;
};

/* Dummy packet data*/
struct packet_data{
  unsigned char packetdata;
  unsigned char packetdataB;
  unsigned char packetdataC;
```

```
  unsigned char packetdataD;
};

#endif // __DTNNC_node_dummy_packet__
```

Listing 5: `DTNNC_node_dummy_packet.h`

`DTNNC_dataStorage_handler.h` defines generic functions providing the storage functionality. The data storage structure can easily be changed without changing many other code source files. Accessing data is allowed by using get and set methods and not accessing to the data directly.

```
#ifndef __DTNNC_dataStorage_handler__
#define __DTNNC_dataStorage_handler__

#include "DTNNC_functions.h"

// dataHolder get and set methods
void* getDataHolderDataAt(call_t* c);

void* getDataHolderData_packetInfoAt(call_t*c, int positionInHeader
    );

int getDataHolder_numberOfpacketsCombinedAt(call_t*c);

int setDataHolderDataAt(call_t* c, struct packet_data* dataT);

int setDataHolderData_packetInfoAt(call_t*c, int positionInHeader,
    struct header_packets_combined* tempHPC);

int setDataHolder_numberOfpacketsCombinedAt(call_t*c, int
    numberOfpacketsCombinedIn);

// dataStorage get and set methods

// Get method to get packet data
void* getDataAt(call_t* c, int position);

void* getData_packetInfoAt(call_t*c, int positionInDataStorage, int
     positionInHeader);

int getData_numberOfpacketsCombinedAt(call_t*c, int position);

// Set method to set packet data
int setDataAt(call_t* c, int position, struct packet_data* dataT);

int setData_packetInfoAt(call_t*c,int positionInDataStorage, int
    positionInHeader, struct header_packets_combined* tempHPC);

int setData_numberOfpacketsCombinedAt(call_t*c, int position, int
    numberOfpacketsCombinedIn);

// data Handling functions

// Generic method to add packet received to storage
// Check if repeat packet is already stored.
// Defination of non repeat packet is received packet
// contains an int not found in that column of storage
// Check storage strategy
int addAllData(call_t * c, packet_t * packet);

// use set methods to add data.
```

```
// for adding in FIFO other.
int addAllDataFIFO(call_t *c, packet_t *packet);

// Swap data between 2 specified data in storage
int swapRowAllData(call_t * c, int dataA, int dataB);

// Reconstructed packet storage get and set methods

// Get method to get data in reconstructed storage
void* getRCDataAt(call_t* c, int position);

// Set method to set data in reconstructed storage
int setRCDataAt(call_t* c, int position, struct packet_data* dataT)
    ;

#endif // __DTNNC_dataStorage_handler__
```

Listing 6: `DTNNC_dataStorage_handler.h`

3.4 Network-Coding API Definition

3.4.1 Node NC Entry Point: `functions.h`

`DTNNC_functions.h` header file is the main entry point of our framework and connects all other files of the module. It defines the four functionalities of one node: storing / dropping / encoding / decoding IP packets.

Creation of a node only needs to include `DTNNC_functions.h`

```
#ifndef __DTNNC_functions__
#define __DTNNC_functions__

#include <stdio.h>
#include <include/modelutils.h>

#include "DTNNC_dictionary.h"
#include "DTNNC_node_structure.h"
#include "DTNNC_node_entity.h"
#include "DTNNC_node_dummy_packet.h"

// The headers files below have included this header file
//#include "DTNNC_dataStorage_handler.h"
//#include "DTNNC_functions_encode.h"
//#include "DTNNC_functions_decode.h"
//#include "DTNNC_functions_log.h"
//#include "DTNNC_functions_mask.h"

/* Functions */

/* Encodes packets */
// Calls encodeFunction() from "DTN_functions_encode.h"
// which does network coding on two packets depending on policy set
// Stores resultant packet into dataHolder storage
int encode(call_t* c);

/* Decodes packets */
// Calls decodeFunction() from "DTN_functions_decode.h"
// which attempts to do the decoding
// stores decoded packets into reconstructed packets storage
int decode(call_t* c);
```

```
/* Stores received packet */
// Checks if node has decoded packets. if yes than drop received
    packet
// Copy received packet to dataHolder storage, This is for sending(
    read report)
// Calls addAllData() from "DTN_dataStorage_handler.h"
// which stores the packet data structure
int store(call_t* c, packet_t* packet);

/* Drops packets */
// Have not been used
// Currently empty
int drop(call_t* c);

#endif // __DTNNC_functions__
```

Listing 7: `DTNNC_functions.h`

3.4.2 NC Masking: `functions_mask.h`

Before combining different packets, a choice of fragmenting data information of one packet can be done. Part of the data can be kept and part can be 'masked'. We use the word 'mask' for the randomly chosen coefficient used in the random linear network coding ($p = \sum_i \lambda_i p_i$, with p_i the packet fragments, λ_i the coefficients which are referred to in the following as 'masks'). `DTNNC_functions_mask.h` offers this masking function.

```
#ifndef __DTNNC_functions_mask__
#define __DTNNC_functions_mask__

#include "DTNNC_functions.h"

/* Mask data */
// mask data chosen from data structure (dataChoice)
// which holds the column in the matrix (coefficientCol)
// with the mask (maskA)
// input maskA of 0 means a random mask
int mask(call_t *c,int dataChoice,int coefficientCol,int maskA);

#endif // __DTNNC_functions_mask__
```

Listing 8: `DTNNC_functions_mask.h`

3.4.3 NC Masking - one implementation: `functions_mask.c`

There are various methods that can be used to mask packet data. Upon consideration, the method described in Figure 6 is used.

One additional byte is added to the data packet during the packet memory allocation. The purpose is to tackle the problem of improper masking when the total bit size of the packet data is not a factor of the mask size.

Packet data is copied byte by byte into a byte storage (byteStoreA). Data in byteStoreA is then transferred bit by bit into a bit storage (bitStoreB). Data in bitStoreB is then transferred bit by bit into a FPower/Size of Mask storage (fPowerStoreC) (in Figure 6, size of the mask is 3). fPowerStoreC is then masked using the selected mask (in Figure 6, the mask is 001). fPowerStoreC is then

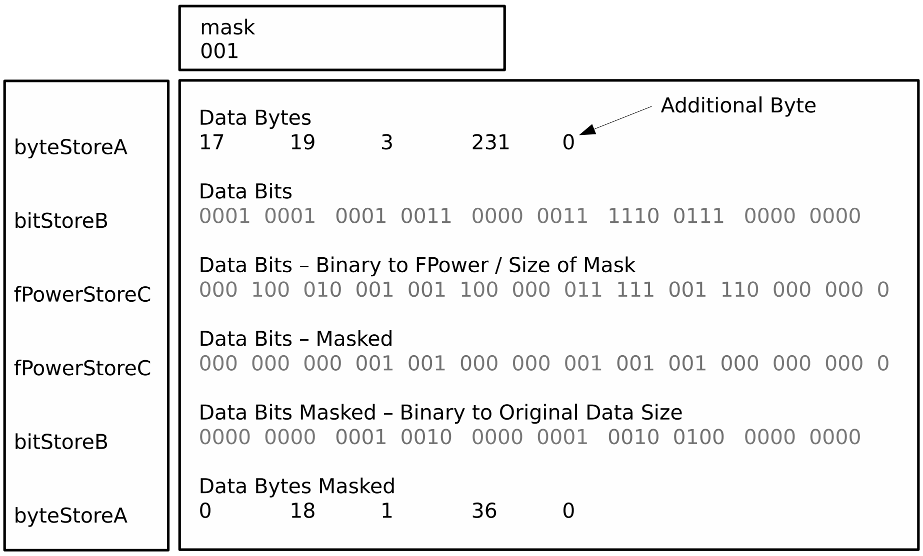

Figure 6: One Data Masking Implementation: One Additional Byte / Byte to Bit conversion / Flow masking

transferred back bit by bit to bitStoreB. bitStoreB is transferred bit by bit to byteStoreA. byteStoreA is then copied byte by byte into the dataHolder. Finally, the mask is stored into the argument storage of the dataHolder.

3.4.4 NC Encoding: `functions_encode.h`

`DTNNC_functions_encode.h` encodes data depending on an encoding strategy.

The `encodeFunction` acts as a controller function. When this function is called, it first checks the encoding validity and then calls the real specific encoding function. Currently the checking consists in testing the node role: if the node is only a relay, it checks if there is only one packet in storage buffer and forwards that packet without changing it; if the node has an encoding role and several packets in the buffer, then the real specific encoding function is called.

```
#ifndef DTNNC_ENCODE_H_
#define DTNNC_ENCODE_H_

#include "DTNNC_functions.h"

// Controller function
// Encoding takes place according to strategy chosen (i.e. random)
// Checks number of packets stored, if only 1 packet in storage
    than return
// Checks encoding type chosen and calls function for encoding
int encodeFunction(call_t *c);

// Specific function
// Encoding takes place between two specified datas
// This function is used for swapping data, needed in decoding
    process
// Checks and encoding type chosen and calls function for encoding
```

```
int encodeFunctionSpecific(call_t *c, int sourceDataA, int
    sourceDataB, int destinationData);

#endif /* DTNNC_ENCODE_H_ */
```

Listing 9: DTNNC_functions_encode.h

3.4.5 NC Encoding - one implementation: functions_encode.c

We provide in the module one random XOR encoding implementation in DTNNC_encode_XOR.h and DTNNC_encode_XOR.c. Two random packets are chosen from the stored data in the node. As rand() of C is biased, an improved version of random is used: seeding of the random number is done at start of node setup at init() of each node type.

```
#ifndef DTNNC_ENCODE_XOR_H_
#define DTNNC_ENCODE_XOR_H_

#include "DTNNC_functions_encode.h"

// Encoding using XOR

// Check encoding strategy and encodes accordingly
// Result stored in dataHolder
int encodeXOR(call_t *c);

// Encode choosing 2 random packets in storage
// Result stored in dataHolder
int encodeXOR_random(call_t *c);

// Encoding using XOR between two specified datas
// Result stored in dataHolder
int encodeXORSpecific(call_t *c, int sourceDataA, int sourceDataB,
    int destinationData);

#endif /* DTNNC_ENCODE_XOR_H_ */
```

Listing 10: DTNNC_encode_XOR.h

3.4.6 NC Decoding: functions_decode.h

DTNNC_functions_decode.h decodes data following a decoding strategy.

The decodeFunction acts also as a controller function. It checks linear dependency before decoding: it checks if the last received packet contains enough relevant new information comparing to existing information in the storage buffer. If so the decoding testing is applied.

```
#ifndef DTNNC_DECODE_H_
#define DTNNC_DECODE_H_

#include "DTNNC_functions.h"

// Controller function
// Decoding takes place according to strategy chosen
int decodeFunction(call_t * c);

#endif /* DTNNC_DECODE_H_ */
```

Listing 11: DTNNC_functions_decode.h

3.4.7 NC Decoding - one implementation: `functions_decode.c`

We provide in the module one Gaussian Elimination implementation in `DTNNC_decode_gaussian_elimination.h` and `DTNNC_decode_gaussian_elimination.c` (cf Listing 12). The `linearIndependentCheck` function implements the Linear Independence Checking; the `forwardSubstitution` function implements the first phase of the Gaussian method: the Forward Substitution; the `reverseElimination` function implements the second phase of the Gaussian method: the Reverse Elimination; the `reconstructPacket` function finally implements the third phase of the Gaussian method and retrieves the original packet.

```
#ifndef DTNNC_DECODE_GAUSSIAN_ELIMINATION_H_
#define DTNNC_DECODE_GAUSSIAN_ELIMINATION_H_

#include "DTNNC_functions_decode.h"

// decoding using the created gaussian elimination method
// 0) linear independence (return 0 and stop decoding if not
    linearly independent)
// 1) Forward Substitution
// 2) Reverse Elimination
// 3) Reconstruct packet
int decode_gaussian_elimination(call_t *c);

// Forward substitution, step 1 of gaussian elimination
// Changes the argument matrix
// Use linearIndependentCheck() as guard to ensure linear
    independent
int forwardSubstitution(call_t *);

// Reverse elimination, step 2 of gaussian elimination
int reverseElimination(call_t *);

// Reconstruction of packet, step 3 of gaussian elimination
// Store reconstructed packets in reconstruction data holder
// Node data storage stores the data up to Step2 (data of each bit)
int reconstructPacket(call_t *);

// Check that the argument matrix is linearly independent
// Return 0 if non linearly independent
int linearIndependentCheck(call_t *);

#endif /* DTNNC_DECODE_GAUSSIAN_ELIMINATION_H_ */
```

Listing 12: `DTNNC_decode_gaussian_elimination.h`

We illustrate the Gaussian implementation with an example in Figures 7-14: a masking in the finite group F_{2^3} has been applied on 2 packets from different sources.

1. Checking Linear Independence

   Checking Linear Independence is basically the same as the Forward Substitution process and code. However doing forward substitution corrupts the matrix and if the modified matrix is not linearly independent, the original matrix cannot be retrieved easily.

   Therefore the first step of checking linear independence phase is to clone the matrix into a temporary matrix for testing linear independence. As this is just a checking phase, data is not touched and therefore not cloned. Check for linear independence fails when the matrix lines swap is not successful, meaning a triangulation can not be performed.

2. Forward Substitution

   Figures 7 to 9 show how forward substitution is done. It starts from the top of the matrix and works in the binary format (even if the matrix is stored in an integer format).

   - Swap phase
     If the first bit of the first column is not at 1, the algorithm finds the first row containing this 1 and data of the rows are swapped.

Figure 7: First step of Forward Substitution phase

Figure 8: Second step of Forward Substitution phase

   - XORing phase
     The algorithm then checks other rows to find if the first bit of the

first column is also at 1. Each matching rows are then xor-ed with the first swapped row, ensuring that only the first row has the first bit positioned to 1. All data of the row are xor-ed as well.

These Swap phase and the XORing phase are repeated for each row from top to bottom until a triangle of 1 is achieved in Figure 9.

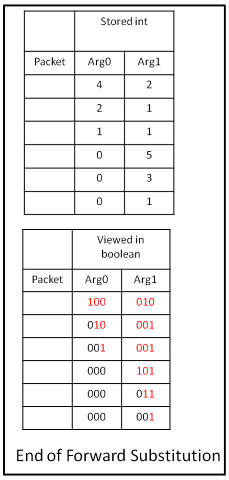

Figure 9: Final state of Forward Substitution phase

3. Reverse Elimination

   Figures 10 to 12 show the reverse elimination process. It starts from the bottom of the matrix. From the bottom matrix last sub column, rows are scanned from bottom to up to ensure only that row is set at 1 in the sub column. Should a 1 be found, a XORing phase is applied and that row is xor-ed along with its data to remove the 1. The algorithm then proceeds on next rows till a diagonal line of 1, like in Figure 12, is achieved.

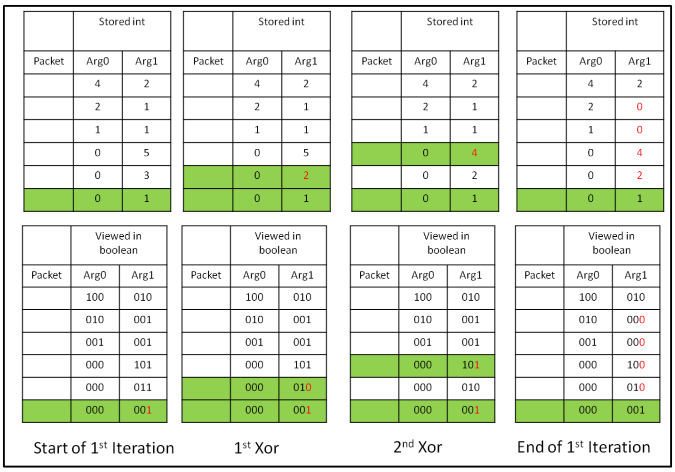

Figure 10: First step of the Reverse Elimination phase

	Stored int	
Packet	Arg0	Arg1
	4	2
	2	0
	1	0
	0	4
	0	2
	0	1

	Viewed in boolean	
Packet	Arg0	Arg1
	100	010
	010	000
	001	000
	000	100
	000	010
	000	001

Start of 2nd Iteration

	Stored int	
Packet	Arg0	Arg1
	4	2
	2	0
	1	0
	0	4
	0	2
	0	1

	Viewed in boolean	
Packet	Arg0	Arg1
	100	010
	010	000
	001	000
	000	100
	000	010
	000	001

1st Xor

	Stored int	
Packet	Arg0	Arg1
	4	2
	2	0
	1	0
	0	4
	0	2
	0	1

	Viewed in boolean	
Packet	Arg0	Arg1
	100	010
	010	000
	001	000
	000	100
	000	010
	000	001

2nd Xor

	Stored int	
Packet	Arg0	Arg1
	4	2
	2	0
	1	0
	0	4
	0	2
	0	1

	Viewed in boolean	
Packet	Arg0	Arg1
	100	000
	010	000
	001	000
	000	100
	000	010
	000	001

End of 2nd Iteration

Figure 11: Second step of the Reverse Elimination phase

	Stored int	
Packet	Arg0	Arg1
	4	0
	2	0
	1	0
	0	4
	0	2
	0	1

	Viewed in boolean	
Packet	Arg0	Arg1
	100	000
	010	000
	001	000
	000	100
	000	010
	000	001

End of Reverse Elimination

Figure 12: Final state of the Reverse Elimination phase

4. Reconstruction of packets

   Figure 13 and Figure 14 show how the reconstruction is done. A reconstruction packet storage is used to separate the fragmented packets from the reconstructed ones (the original data storage may be use later). From the top, (size of mask) number of rows are xor-ed to the first packet of the reconstruction storage to reconstruct the original packet. The data is Xor-ed as well. Then the same process of (size of mask) number of rows xor-ing is applied until the end of the matrix.

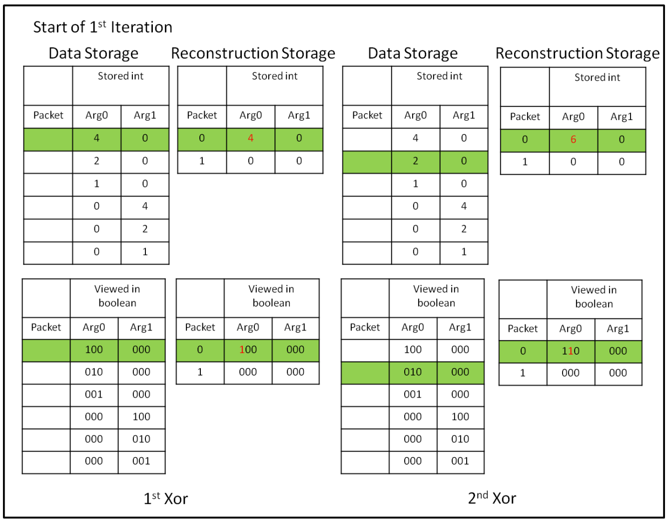

Figure 13: First xor-ings of the Reconstruction phase

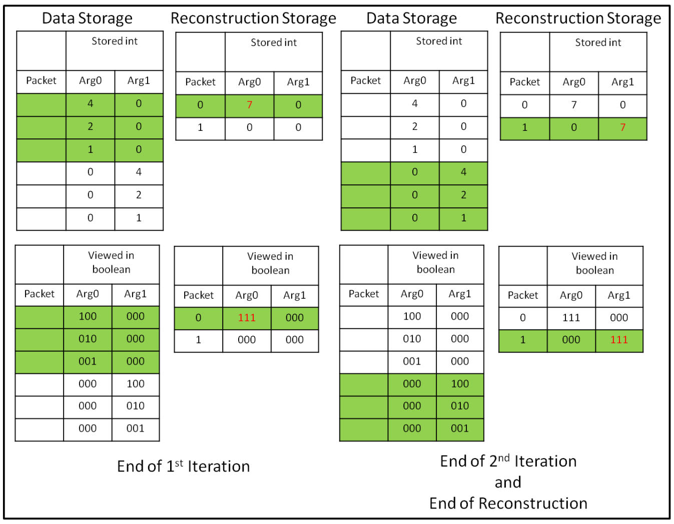

Figure 14: End of xor-ings, final state of Reconstruction phase

3.5 Log API Definition

WSNet produces ASCII prints on standard output. `DTNNC_functions_log.h` defines some useful functions for printing out statistics logs and debug outputs. `DTNNC_functions_log.h` includes node common variables outputs: id, position, etc.

```
#ifndef __DTNNC_functions_log__
#define __DTNNC_functions_log__

#include "DTNNC_functions.h"

// generic function
```

```
// Use dicitonary.h and input at choice
// Check debug first
int logFunction(call_t * c, int choice);

// Check which log chosen
int logNormal(call_t * c, int choice);

// Check which print out chosen
int logDebug(call_t * c, int choice);

// Function to print out node type
int print_node_type(call_t * c);

// Function to print out node id
int print_nodeid(call_t * c);

// Function to print out node position
int print_nodePosition(call_t * c);

// Function to print out failure to print log
// Can add what to print when failed to print log
// Currenly printing node type id and position
int print_log_failed(call_t * c,int choice);

#endif // __DTNNC_functions_log__
```

Listing 13: `DTNNC_functions_log.h`

`DTNNC_log_normal.h` outputs statistics about 'normal' events of the Network Coding module use: number of packets received/sent, encoding/decoding/ linear checking, etc.

```
#ifndef __DTNNC_log_normal__
#define __DTNNC_log_normal__

#include "DTNNC_functions_log.h"

// print send event
int print_send_event(call_t *);

// print receive event
int print_receive_event(call_t *);

// print store event
int print_store_packet(call_t *);

// print drop packet event when packet is repeat packet
int print_drop_packet_repeat(call_t *);

// print drop packet event when node has already decoded packets
int print_drop_packet_decoded(call_t *);

// print linear check fail event
int print_linear_check_failed(call_t *);

// print linear check pass event
int print_linear_check_passed(call_t *);

// print event that node has decoded packets
int print_decoded(call_t *);

// print event that node has encoded packets
```

```
int print_encoded(call_t *);

// print event of destroying or unsettting node at end of
    simulation
// may print information like counters and such held in node here
int print_unsetnode(call_t *);

#endif // __DTNNC_log_normal__
```

Listing 14: `DTNNC_log_normal.h`

`DTNNC_log_debug.h` implements debugging log functions. These functions print the internal node state.

```
#ifndef __DTNNC_log_debug__
#define __DTNNC_log_debug__

#include "DTNNC_functions_log.h"

// prints what is in data storage
int print_node_storage(call_t *);

// prints what is in data holder
int print_data_holder(call_t *);

// prints what is in reconstructed packets
int print_reconstructed_packets(call_t *);

#endif // __DTNNC_log_debug__
```

Listing 15: `DTNNC_log_debug.h`

Debug outputs are data-packet specific. These functions do not show only the packet header but output also the packet data to check its correctness. Therefore changing the dummy packet data structure will require to reimplement these functions. For instance, such implementation of the `DTNNC_log_debug.h` API needs to be adapted.

```
printf("] dataA [%d], dataB [%d], dataC [%d], dataD [%d] \n",((
    struct packet_data *)getDataAt(c,i))->packetdata,((struct
    packet_data *)getDataAt(c,i))->packetdataB,((struct packet_data
     *)getDataAt(c,i))->packetdataC,((struct packet_data *)
    getDataAt(c,i))->packetdataD);
```

4 Conclusion

This technical report has described the implementation of a *Network Coding module for Wireless and Mobile DTN* in WSNet - a Wireless Sensor Network simulator. This module provides a generic framework that includes:

- Programming Interfaces that defines a generic DTN node and its functionalities: IP packet storing, selecting/dropping, encoding/decoding.
- Implementations for the main Network Coding functionalities: random selecting, random linear coding over F_{2^n}, Gaussian Elimination decoding.

Programming Interfaces has been generically defined to allow an easy specialization for future different coding strategies: source/destination-oriented, intra/inter-flow, application-oriented, social-oriented.

References

[1] Eitan Altman, Francesco De Pellegrini, and Lucile Sassatelli. Dynamic control of coding in delay tolerant networks. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 121–125, Piscataway, NJ, USA, 2010. IEEE Press. 2.1

[2] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 27–34, New York, NY, USA, 2003. ACM. 2.1

[3] Antoine Fraboulet, Guillaume Chelius, and Eric Fleury. Worldsens: development and prototyping tools for application specific wireless sensors networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 176–185, New York, NY, USA, 2007. ACM. 2.3

[4] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. Network coding: an instant primer. *SIGCOMM Comput. Commun. Rev.*, 36:63–68, January 2006. 2.2

[5] Elyes Ben Hamida and Guillaume Chelius. *WSNet : How to write a new application module ?*, 2007. http://wsnet.gforge.inria.fr/tutorials/dev-app/index.html. 3.1

[6] Elyes Ben Hamida and Guillaume Chelius. *WSNet : How to write and setup an xml configuration file ?*, 2007. http://wsnet.gforge.inria.fr/tutorials/configuration/index.html. 3.1

[7] Sachin Katti, Dina Katabi, Wenjun Hu, Hariharan Rahul, and Muriel Medard. The importance of being opportunistic: Practical network coding for wireless environments. In *Proceedings of 43rd Allerton Conference on Communication, Control, and Computing*, 2005. 2.2

[8] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, New York, NY, USA, 2009. ICST. 2.3

[9] Yunfeng Lin, Baochun Li, and Ben Liang. Efficient network coded data transmissions in disruption tolerant networks. In *Proceedings of the IEEE 27th Conference on Computer Communications.*, INFOCOM'08, pages 1508 –1516, April 2008. 2.1

[10] Lídice Romero Amondaray and Joaquín Seoane Pascual. Delay tolerant network simulation with vnuml. In *Proceedings of the third ACM workshop on Challenged networks*, CHANTS '08, pages 109–112, New York, NY, USA, 2008. ACM. 2.3

[11] Xiaolan Zhang, Giovanni Neglia, Jim Kurose, and Don Towsley. On the benefits of random linear coding for unicast applications in disruption tolerant networks. In *Proceedings of IEEE Workshop on Network Coding, Theory, and Applications.*, NETCOD'06, 2006. 2.1

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803