Kodo-ns3-examples Documentation

Release master

Steinwurf ApS

May 18, 2017

Contents

Welcome to the documentation of the Steinwurf Kodo with ns-3 examples repository!

This set of documents provides a tutorial showing how to use Steinwurf C++ Kodo library together with the ns-3 simulator. Particularly, here we employ C++ bindings for the library to ease the deployment of the examples. This tutorial is designed as a starting point for ns-3 developers that intend to use the Kodo library for a specific application of Random Linear Network Coding (RLNC) within the simulator.

CHAPTER 1

Before You Start

The project description, licensing, source code and how to compile are available here. **Once you have built the project**, you can follow this tutorial. Otherwise, please be sure to have the project up and running first.

Overview

Currently the repository `kodo-ns3-examples` contains 3 basic examples in its main path regarding how to use the library with ns-3, namely:

- `kodo-wifi-broadcast`: This example consists on broadcasting packets with RLNC from a transmitter to N receivers with an IEEE 802.11b WiFi channel.
- `kodo-wired-broadcast`: This example consists on broadcasting packets with RLNC from a transmitter to N receivers with the same erasure channel.
- `kodo-recoders`: This example shows the gain of RLNC with recoding in a 2-hop line network consisting of an encoder, N recoders and a decoder with different erasure rates. Recoding can be set on or off and erasure rates modified by command-line parsing.

Automated Builds

You can check the build status of the repository master branch on our buildbot page. Our buildbot displays the supported combinations of platforms, operating systems, and compilers. At the link, you can check build status and build statistics for them in the respective waterfall link. This information is provided also for other Steinwurf projects such as Kodo itself.

What You Should Know Before the Tutorial

C++

In order for the tutorial to be easy to read, some basic knowledge of C++ is recommended. If you are also a C++ beginner, you can refer to this tutorial as a guide from the basics to more advanced features of the language. Given that both ns-3 and Kodo are highly object-oriented projects, we strongly recommend you to spend some time on

class related topics, particularly object properties (polymorphism, inheritance) and templates (generic classes or functions based on abstract types). In the mentioned C++ tutorial you may find plenty examples for this.

Kodo

Kodo is a C++ library from Steinwurf that implements Random Linear Network Coding and its variants, but also other codes like Reed-Solomon. The links you should review first to get a clear idea of what the library does, are:

- Kodo documentation
- Kodo source code

Kodo C++ bindings

To ease the development process, we use C++ bindings that are high level wrappers of the basic core application programming interfaces of Kodo. Besides making the process easier, the bindings provide encapsulation and abstraction from the library. If you want to get more details, please refer to the previous link. There you will find documentation and examples regading on how to use the bindings on their own.

ns-3

ns-3 is a network simulator of the OSI layers written in C++ for research and educational purposes under the GPLv2 license. You can find documentation regarding a tutorial, a manual and a model description in this link.

Waf

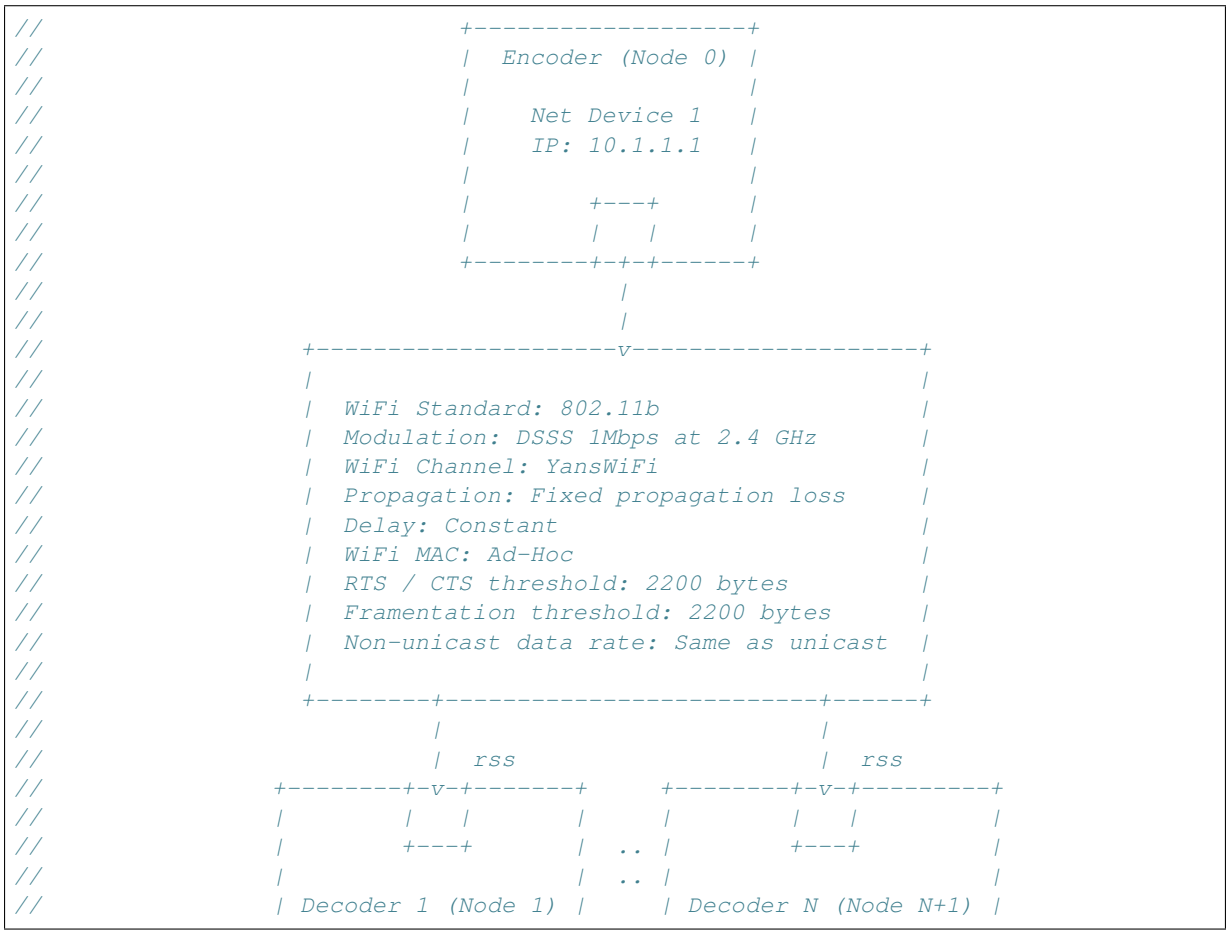
Our main building tool is `waf` in the same way it is for ns-3. So, if you have used ns-3 before this should be transparent to you. If you have not used `waf` , you can review a description of the tool with some examples in the waf project. For this reason, we highly recommend you to follow the instructions in the repository link mentioned at the beginning of this document.

CHAPTER 2

Broadcast RLNC with a WiFi Channel

General Topology

The topology considered describes a transmitter sending coded packets with RLNC from a generation size g and field size q in a broadcast fashion through a 802.11b channel to N receivers. For the purpose of our example, we will start with $g = 5$, $q = 2$ (i.e. the binary field), $N = 2$ and we will check the completion time in terms of transmissions through the WiFi channel under different situations. Topology is shown as follows:



```
//                 |                      |     |                        |
//                 |    Net Device 1      |     |    Net Device 1        |
//                 |    IP: 10.1.1.2      |     |    IP: 10.1.1.N+1      |
//                 +----------------------+     +------------------------+

//                 N: number of decoders    rss: Received Signal Strength
```

What to Simulate

We will consider the following guidelines for our simulation:

- Behavior: The sender keeps transmitting the generation until the receiver has g linearly independent (l.i.) coded packets (combinations). Packets might or might not be loss due to channel impairments.
- Inputs: As main parameters regarding RLNC, we choose the generation and field size. A parameter regarding channel control should be included
- Outputs: A counter to indicate how much transmissions did the process required and some prints to indicate when decoding is completed.
- Scenarios: We will variate the generation and field size to verify theoretical expected values regarding the amount of transmissions to decode. Also, the number of transmissions should somehow change as we vary the channel.

Program Description

After the project has been properly configured and built, you should have a folder named `kodo` in the examples folder of your local ns-3 repository (e.g. in `~/ns-3-dev/examples`). If you check there, you will find two type of files. First, the header files are the implementations for the considered topologies in our examples. You are free to add new topologies as well. Second, the source files are the actual applications of those topologies within ns-3. For our case here, we will first review the broadcast topology contained in `kodo-broadcast.h` and its application in `kodo-wifi-broadcast.cc`, which contains the source code of this simulation. You can open them with your preferred editor to review the source code. We will briefly review some of its parts.

Overview Comments and Includes

```
// General comments: E-macs descriptor, ns-3 license and example description

// ns-3 includes
#include <ns3/core-module.h>
#include <ns3/network-module.h>
#include <ns3/mobility-module.h>
#include <ns3/config-store-module.h>
#include <ns3/wifi-module.h>
#include <ns3/internet-module.h>

// Simulation includes
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <ctime>

// Kodo includes
#include "kodo-broadcast.h" // Contains the broadcast topology class
```

As with any source code, the overview comments provide a reference to the users regarding project general aspects. The E-macs descriptor is part of the ns-3 coding style to allow E-macs developers' editor to recognize the document type. Following, licensing terms and an introduction to what we are simulating are displayed. Header includes are ordered from most general to particular functionalities within ns-3 and Kodo. From ns-3, the necessary modules are:

- Core module: For simulation event handling. This module provide a set of class-based APIs that control the simulation behavior. It is essential for every ns-3 simulation.
- Network module: For creation and management of simulated devices. ns-3 has mainly two building blocks which represent a transmission/reception interface on physical equipment, namely **nodes** ("equipment") from a networking perspective and **net devices** ("interface cards") which actually deal with the physical layer. A node may have various net devices, but a net device cannot be shared by various nodes. We will also use the `Packet` and `ErrorModel` classes from this module to represent other simulation objects in the examples.
- Mobility module: For providing a description of how the nodes move in an environment. We will use briefly this module for a representation of our physical channel.
- Config-store module: A specialized ns-3 database for internal attributes and default values for the different APIs.
- WiFi module: A PHY and MAC layer models for WiFi. Necessary for our medium access control and channel model.
- Internet module: For handling the IPv4 protocol at the network layer.

Other includes are particular to this implementation and they can be found in standard C++ code. From Kodo, the header `kodo-broadcast.h` contains the description of the encoder and decoder objects that we use.

Default Namespace

We will be working within the ns-3 scope given that most of our objects are from this library. This is typical across ns-3 code.

```
using namespace ns3;
```

Simulation Class

Before starting, we describe the object created in `kodo-broadcast.h` with the purpose to represent the RLNC broadcast topology. In this sense, we represent our Kodo simulation as a class with different functionalities. Of course, this is purely subjective. You may choose how you represent your objects in your simulation. Although, we choose this way because it enabled us to modularize all the simulation into a single object which is controlled by the network through the tasks of the net devices. Also, other ns-3 objects can extract information from it in an easy way.

The `Broadcast` class can be roughly defined in the following way:

```
#pragma once

#include <kodocpp/kodocpp.hpp>

class Broadcast
{
public:

  Broadcast (const kodocpp::codec codeType, const kodocpp::field field,
    const uint32_t users, const uint32_t generationSize,
    const uint32_t packetSize,
    const ns3::Ptr<ns3::Socket>& source,
    const std::vector<ns3::Ptr<ns3::Socket>>& sinks)
```

```
    : m_codeType (codeType),
      m_field (field),
      m_users (users),
      m_generationSize (generationSize),
      m_packetSize (packetSize),
      m_source (source),
      m_sinks (sinks)
  {
    // Constructor
  }

  void SendPacket (ns3::Ptr<ns3::Socket> socket, ns3::Time pktInterval)
  {
    // Encoder logic
  }

  void ReceivePacket (ns3::Ptr<ns3::Socket> socket)
  {
    // Decoders logic
  }

private:

  const kodocpp::codec codeType;
  const kodocpp::field field
  const uint32_t m_users;
  const uint32_t m_generationSize;
  const uint32_t m_packetSize;

  ns3::Ptr<ns3::Socket> m_source;
  std::vector<ns3::Ptr<ns3::Socket>> m_sinks;
  kodocpp::encoder m_encoder;
  std::vector<uint8_t> m_encoderBuffer;
  std::vector<kodocpp::decoder> m_decoders;
  std::vector<std::vector<uint8_t>> m_decoderBuffers;

  std::vector<uint8_t> m_payload;
  uint32_t m_transmissionCount;
};
```

The broadcast topology is a simple class. We will describe its parts in detail what they model and control from a high level perspective. We first need to include main header for the bindings `<kodocpp/kodocpp.hpp>` since they contain all the objects required for the C++ bindings to work.

First, we need to define our encoder and decoders. For this purpose, we have to specify the coding scheme that we are going to employ and the field type in which the finite field arithmetics are going to be carried. This is done by employing the data types `kodocpp::codec` and `kodocpp::field` which are defined in the bindings. To avoid using templated classes, we pass the required code type and field type as constructor arguments.

Given that we want to perform our basic simulation with RLNC, the type in the bindings is `kodocpp::codec::full_vector`. You can look at it as a wrapper for the codecs in the Kodo library. Similarly, `kodocpp::field::binary` is the field type in the bindings for the binary field implementation (since we are interested in $q = 2$) which is defined in the bindings repository. Think of it as a wrapper for the fields described in the Fifi library. However, other field types from Fifi might be chosen too from their bindings according to your application. Current available field sizes are: $q = 2, 2^4, 2^8$.

For the simulation, `void SendPacket(ns3::Ptr<ns3::Socket> socket, ns3::Time pktInterval)` generates coded packets from generic data (created in the constructor) every `pktInterval` units of `Time` and sends them to the decoders through their socket connections, represented by the ns-3 template-based smart pointer object `ns3::Ptr<ns3::Socket>`. Several ns-3 objects are represented in this way. So quite often you will see this kind of pointer employed. This ns-3 pointer is intended to make a proper memory usage.

As we will check later, `void ReceivePacket(ns3::Ptr<ns3::Socket> socket)` will be invoked through a callback whenever a packet is received at a decoder socket. In this case, the decoder that triggered the callback is obtained from looking into a vector container. The transmitter creates coded packets from the data and puts them in `m_payload` to send it over. Conversely, a received coded packet is placed in a local `payload` to be read by the inteded decoder and its respective decoding matrix.

You can check the source code to verify that these functionalities are performed by the APIs `m_encoder.write_payload()` and `m_decoders[n].read_payload()`. For the encoding case, the amount of bytes required from the buffer to store the coded packet and its coefficients is returned. This amount is needed for `ns3::Create<ns3::Packet>` template-based constructor to create the ns-3 coded packet that is actually sent (and received). Finally, `m_transmission_count` indicates how many packets were sent by the encoder during the whole process. Please make a review to the implementation of `SendPacket` and `ReceivePacket` to verify the expected behavior of the nodes when packets are sent or received respectively.

Default Parameters and Command-line Parsing

```
1  std::string phyMode ("DsssRate1Mbps");
2  double rss = -93;  // -dBm
3  uint32_t packetSize = 1000; // bytes
4  double interval = 1.0; // seconds
5  uint32_t generationSize = 5;
6  uint32_t users = 2; // Number of users
7  std::string field = "binary"; // Finite field used
8
9  // Create a map for the field values
10 std::map<std::string,kodocpp::field> fieldMap;
11 fieldMap["binary"] = kodocpp::field::binary;
12 fieldMap["binary4"] = kodocpp::field::binary4;
13 fieldMap["binary8"] = kodocpp::field::binary8;
14
15 CommandLine cmd;
16
17 cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
18 cmd.AddValue ("rss", "received signal strength", rss);
19 cmd.AddValue ("packetSize", "size of application packet sent", packetSize);
20 cmd.AddValue ("interval", "interval (seconds) between packets", interval);
21 cmd.AddValue ("generationSize", "Set the generation size to use",
22               generationSize);
23 cmd.AddValue ("users", "Number of receivers", users);
24 cmd.AddValue ("field", "Finite field used", field);
25
26 cmd.Parse (argc, argv);
27
28 // Use the binary field in case of errors
29 if (fieldMap.find (field) == fieldMap.end ())
30   {
31     field = "binary";
32   }
33
34 // Convert to time object
35 Time interPacketInterval = Seconds (interval);
```

The first part of the `main` function introduces us to the basic simulation parameters regarding physical layer mode for WiFi (Direct Sequence Spread Spectrum of 1 Mbps rate), receiver signal strength of -93 dBm, 1 KB for packet size, 1 second interval duration between ns-3 events (we will use it later), a generation size of 5 packets, 2 users (receivers) and a string for the finite field to be employed. The string name will be searched in a `std::map` container that has the proper `kodocpp::field` instance. After that, the `CommandLine` class is ns-3's command line parser used to modify those values (if required) with `AddValue` and `Parse`. Then, the interval duration is converted to the ns-3 `Time` format.

Configuration defaults

```
// disable fragmentation for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
  StringValue ("2200"));

// turn off RTS/CTS for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
  StringValue ("2200"));

// Fix non-unicast data rate to be the same as that of unicast
Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
  StringValue (phyMode));
```

Before continuing, you will see many features of ns-3's WiFi implementation. Besides the WiFi properties, in the previous link you will find a typical workflow about setting and configuring WiFi devices in your simulation.

This part basically sets off some MAC properties that we do not need (at least for our purposes), namely frame fragmentation to be applied for frames larger than 2200 bytes, disabling the RTS/CTS frame collision protocol for the less than 2200 bytes and setting the broadcast data rate to be the same as unicast for the given `phyMode`. However, they need to be included in order to work with the WiFi MAC.

WiFi PHY and Channel Helpers for Nodes

```
// Source and destination
NodeContainer nodes;
nodes.Create (1 + users); // Sender + receivers

// The below set of helpers will help us to put together the wifi NICs we
// want
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211b); // OFDM at 2.4 GHz

YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// The default error rate model is ns3::NistErrorRateModel

// This is one parameter that matters when using FixedRssLossModel
// set it to zero; otherwise, gain will be added
wifiPhy.Set ("RxGain", DoubleValue (0));

// ns-3 supports RadioTap and Prism tracing extensions for 802.11g
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");

// The below FixedRssLossModel will cause the rss to be fixed regardless
// of the distance between the two stations, and the transmit power
wifiChannel.AddPropagationLoss ("ns3::FixedRssLossModel","Rss",
  DoubleValue (rss));
wifiPhy.SetChannel (wifiChannel.Create ());
```

In this part we start to build the topology for our simulation following a typical ns-3 workflow. By typical we mean that this can be done in different ways, but this one you might see regularly within ns-3 simulations. We start by creating the nodes that we need with the `NodeContainer` class. You can create the nodes separately but this way offers the possibility to easily assign common properties to the nodes.

We aid ourselves by using the `WiFiHelper` class to set the standard to use. Since we are working with DSSS, this means we need to use IEEE 802.11b. For the physical layer, we use the `YansWifiPhyHelper::Default()` constructor and from it, we disable any gains in the receiver and set the pcap (packet capture) tracing format at the data link layer. ns-3 supports different formats, here we picked the RadioTap format but you can

choose other format available in the helper description in its Doxygen documentation. In a similar way, we use the `YansWifiChannelHelper` to create our WiFi channel, where we have set the class property named `SetPropagationDelay` to `ConstantSpeedPropagationDelayMode`. This means that the delay between the transmitter and the receiver signals is set by their distance between them, divided by the speed of light. The `AddPropagationLoss` defines how do we calculate the receiver signal strength (received power) in our model. In this case, we have chosen a `FixedRssLossModel` which sets the received power to a fixed value regardless of the position the nodes have. This fixed value is set to -93 dBm, but we can modify it through argument parsing. With these settings, we create our WiFi PHY layer channel by doing `wifiPhy.SetChannel (wifiChannel.Create ());`. If you want to read more about how the helpers are implemented, you can check the Yans description for further details.

WiFi MAC and Net Device Helpers

```
// Add a non-QoS upper mac, and disable rate control
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
  "DataMode",StringValue (phyMode), "ControlMode",StringValue (phyMode));

// Set WiFi type and configuration parameters for MAC
// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");

// Create the net devices
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, nodes);
```

Now that we have created the physical objects (the nodes, remember our previous definition), we proceed to create the network interface cards (NICs, i.e. net devices) that will communicate the different nodes. But first, we need to set up the MAC layer. For this, we use the `NqosWifiMacHelper` which provides an object factory to create instances of WiFi MACs, that do not have 802.11e/WMM-style QoS support enabled. We picked this one because we are just interested in sending and receiving some data without QoS. By setting the type as `AdhocWifiMac`, we tell ns-3 that the nodes work in a decentralized way. We also need to set the devices data rate control algorithms, which we do with the `WifiHelper`. This is achieved by setting the remote station manager property to `ConstantRateWifiManager` for data and control packets using the given `phyMode`. This implies that we a fixed data rate for data and control packet transmissions. With all the previous settings we create our (2) WiFi cards and put them in a container by doing `NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, nodes);`.

Mobility Model and Helper

```
// Note that with FixedRssLossModel, the positions below are not
// used for received signal strength. However, they are required for the
// YansWiFiChannelHelper
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc =
  CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (0.0, 0.0, 0.0));  // Source node

for (uint32_t n = 1; n <= users; n++)
  {
    positionAlloc->Add (Vector (5.0, 5.0*n, 0.0));
  }

mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (nodes);
```

The ns-3 `MobilityHelper` class assigns a model for the velocities of the receivers within ns-3. Even though we had fixed the received power of the decoder, it is a necessary component for the `YansWiFiChannelHelper`.

We create a `Vector` describing the initial (and remaining) coordinates for both transmitter and receiver in a 3D grid. Then, we put them in the helper with a `ConstantPositionMobilityModel` for the nodes.

Internet and Application Protocol Helpers

```
1  InternetStackHelper internet;
2  internet.Install (nodes);
3
4  Ipv4AddressHelper ipv4;
5  ipv4.SetBase ("10.1.1.0", "255.255.255.0");
6  ipv4.Assign (devices);
```

After we have set up the devices and the two lowest layers, we need to set up the network and application layer protocols. The `InternetStackHelper` provides functionalities for IPv4, ARP, UDP, TCP, IPv6, Neighbor Discovery, and other related protocols. You can find more about the implementation of the helper in this link. A similar process is made for the IPv4 address assignment. We use the address range `10.1.1.0` with the subnet mask `255.255.255.0`, assign it to the `devices`.

Sockets Construction

```
1  TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
2
3  // Transmitter socket
4  Ptr<Socket> source = Socket::CreateSocket (nodes.Get (0), tid);
5
6  // Receiver sockets
7  std::vector<Ptr<Socket>> sinks (users);
8
9  for (uint32_t n = 0; n < users; n++)
10   {
11     sinks[n] = Socket::CreateSocket (nodes.Get (1 + n), tid);
12   }
```

For the application protocols to work with a given data, we need a pair between an IP address and a logical port to create a socket address for socket communication (besides of course, the socket itself). ns-3 supports two sockets APIs for user space applications. The first is ns-3 native, while the second (which is based on the first) resembles more a real system POSIX-like socket API. For further information about the differences, please refer to ns-3's socket implementation. We will focus on the ns-3 socket API variant. The first line is meant to create the socket type from a lookup search given by the name `UdpSocketFactory`. It creates this type of socket on the receivers and the transmitter. We have chosen the previous socket type in order to represent a UDP connection that sends RLNC coded packets.

Simulation Calls

```
1  // Creates the Broadcast helper for this broadcast topology
2  Broadcast wifiBroadcast (kodocpp::codec::full_vector, fieldMap[field],
3    users, generationSize, packetSize, source, sinks);
```

As we mentioned earlier, we use the RLNC codec and binary for our encoder and decoders. Then, we call the object that handles the topology by doing `Broadcast wifiBroadcast (kodocpp::codec::full_vector, fieldMap[field], users, generationSize, packetSize, sinks);` to call the broadcast class constructor. Notice also that we have separated the code type from the topology in case that you want to try another code. This does not run the simulation as we will see, but it creates the objets called by ns-3 to perform the tasks of the transmitter and receiver.

Sockets Connections

```
// Transmitter socket connections. Set transmitter for broadcasting
uint16_t port = 80;
InetSocketAddress remote = InetSocketAddress (
  Ipv4Address ("255.255.255.255"), port);
source->SetAllowBroadcast (true);
source->Connect (remote);

// Receiver socket connections
InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), port);
for (const auto sink : sinks)
  {
    sink->Bind (local);
    sink->SetRecvCallback (MakeCallback (&Broadcast::ReceivePacket,
      &wifiBroadcast));
  }

// Turn on global static routing so we can be routed across the network
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Then, we create the remote and local socket addresses for binding purposes. For the transmitter (source) we make a similar process but instead we allow broadcasting with `source->SetAllowBroadcast (true)` and connect to the broadcast address. For the receivers, we choose the default `0.0.0.0` address obtained from `Ipv4Address::GetAny ()` and port 80 (to represent random HTTP traffic). The receiver binds to this address for socket listening. Every time a packet is received we trigger a callback to the reference `&Broadcast::ReceivePacket` which takes the listening socket as an argument. This executes the respective member function of the reference `&wifiBroadcast`. This completes our socket connection process and links the pieces for the simulation. Finally, we populate the routing tables to ensure that we are routed inside the topology.

Simulation Event Handler

```
// Pcap tracing
wifiPhy.EnablePcap ("kodo-wifi-broadcast", devices);

Simulator::ScheduleWithContext (source->GetNode ()->GetId (), Seconds (1.0),
  &Broadcast::SendPacket, &wifiBroadcast, source, interPacketInterval);

Simulator::Run ();
Simulator::Destroy ();

return 0;
```

Finally, `wifiPhy.EnablePcap ("kodo-wifi-broadcast", devices);` allows the net devices to create pcap files from the given devices. One file per net device. File naming will be: `kodo-wifi-broadcast-[NODE_ID]-[DEVICE_ID].pcap` and the format of these files should be the one of RadioTap and should be located on your `~/kodo-ns3-examples/` folder. Later, we will review how to read those files.

After the pcap setting, we use one of the ns-3 core features, event scheduling. The `Simulator` class is inherent to ns-3 and defines how events are handled discretely. The `ScheduleWithContext` member function basically tells ns-3 to schedule the `Broadcast::SendPacket` function every second from the transmitter instance of `wifiBroadcast` and provide its arguments, e.g. ns-3 socket pointer `source` and `Time` packet interval `interPacketInterval`. Among the event schedulers, you will see `Schedule` vs. `ScheduleWithContext`. The main difference between these two functions is that the `ScheduleWithContext` tells ns-3 that the scheduled's event context (the node identifier of the currently executed network node) belongs to the given node. Meanwhile, `Schedule` may receive the context from a previous scheduled event, which can have the context from a different node. You can find more details about the simulator functions in the ns-3 event scheduling manual. With all previous descriptions, we are able to run the simulation to see some basic effects of network coding in ns-3 with Kodo.

Simulation Runs

Now that we know each part of our setup, we will run some simulations in order that you should know what to expect. We will run the default behavior and change some parameters to check known results.

Default Run

First type `cd ~/ns-3-dev` in your terminal for you to be in the path of your ns-3 cloned repository. Also remember that at this point, **you need to have configured and built the project with no errors**. If you review the constructor of the `Broadcast` class, you will observe that there is a local callback function made with a lambda expression.

Note: A lambda expression is basically a pointer to a function and is a feature available since C++11. If you need further information on this topic, please check the C++ tutorial mentioned at the beginning of this guide.

The callback is to enable tracing in our examples, e.g. to observe how the packets are being processed. The callback uses filters to control the outputs that we want to observe. In the class constructor, the options that are enabled are: (i) `symbol_coefficients_before_read_symbol` which tells how are the coding coefficients of the received packet before inserting it in the coding matrix and (ii) `decoder_state` which tells how is the state of the coding coefficients matrix at the decoder after performing Gaussian elimination. Laer, the default run goes with 5 packets in the binary field with 2 users and the previosuly tracing options defined.

As a starter (once located in the path described earlier), type:

```
python waf --run kodo-wifi-broadcast
```

You should see an output similar to this:

```
+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 1

decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 1 1 1
002 ?:  0 0 0 0 0
003 ?:  0 0 0 0 0
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 1

decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 1 1 1
002 ?:  0 0 0 0 0
003 ?:  0 0 0 0 0
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 1 1 0 0
```

```
decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 1 0 0
002 ?:  0 0 0 0 0
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 1 1 0 0

decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 1 0 0
002 ?:  0 0 0 0 0
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 0 1 0 1

decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 0 1 0 1

decoder_state:
000 ?:  0 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 1 0 1 1 0

decoder_state:
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 1 0 1 1 0

decoder_state:
```

```
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 1 0 0 1

decoder_state:
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 1 0 0 1

decoder_state:
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 1

decoder_state:
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 1

decoder_state:
000 S:  1 0 0 0 0
001 S:  0 1 0 0 1
002 S:  0 0 1 0 1
003 S:  0 0 0 1 1
004 ?:  0 0 0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 0
```

```
decoder_state:
000 U:  1 0 0 0 0
001 U:  0 1 0 0 0
002 U:  0 0 1 0 0
003 U:  0 0 0 1 0
004 U:  0 0 0 0 1

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 0 1 1 1 0

decoder_state:
000 U:  1 0 0 0 0
001 U:  0 1 0 0 0
002 U:  0 0 1 0 0
003 U:  0 0 0 1 0
004 U:  0 0 0 0 1

Decoding completed! Total transmissions: 7
```

Here we observe that every time a packet is received, the previously mentioned information is printed for each receiver. For the `symbol_coefficients_before_read_symbol` output, `C:` indicates that we have a received a *coded* packet with the given coding vector. In this output, the first given coded packet (CP) is: $CP_1 = p_2 + p_3 + p_4 + p_5$.

Note: Normally the encoder (based on the `kodo_full_vector`), would have generated packets in a systematic way, but here we set that feature off in the `Broadcast` class constructor, through the encoder API `m_encoder.set_systematic_off()`. Also, normally the encoder starts with the same seed in every run but we have also changed that too in the constructor with `srand(static_cast<uint32_t>(time(0)))`. So, we proceed with this example to explain the simulation, but you will obtain another result in your runs. However, the results obtained with this example apply in general.

After the input symbols have been checked, the decoder trace shows the `decoder_state`. This is the current decoding matrix in an equivalent row echelon form. Given that we have received $p_2 + p_3 + p_4 + p_5$, we put them in the second row because the pivot for p_2 is there. Also, we can argue that the pivot for p_3 is in the third row and so on. The second received coded packet is $CP_2 = p_2 + p_3$. Notice that when we print the decoder state again, we have changed the equation of the second and fourth rows because with the current information we can calculate $CP_{1,new} = CP_2 = p_2 + p_3$ (remember we are in modulo-2 arithmetic) and $CP_{2,new} = CP_1 + CP_2 = p_4 + p_5$. Packet reception continues until we have g linearly independent (l.i.) coded packets.

You can also see two more types of symbols indicators. First, `S:` indicates that the corresponding pivot packet has not been *seen* by the decoder. Seeing packet k means that we are able to compute $p_k + \sum_{l>k} \alpha_l p_l$, i.e. to be able to compute p_k plus a combinations of packets of indexes greater than k. A seen packet helps to reduce the numbers of operations required for decoding. Second, `?:` indicates that the pivot of the corresponding pivot has not been seen yet. Finally, `U:` indicates that the packet is uncoded, you will see this when the complete generation is decoded.

At the end, we see that decoding was performed after 7 transmissions. There might be two reasons for this to occur: (i) a linearly dependent (l.d.) coded packet was sent during the process or (ii) there were some packet erasures during the process. From the example, we see that during the fifth and sixth transmission, l.d. coded packets were sent, thus conveying no new information. We will make some changes to see more about these effects.

Changing the Field and Generation Size

Try to run the example again several times, you should see that the amount of transmissions vary between 5 and 7, maybe sometimes a little more, due to randomness. On average, for $q = 2$ you should expect that $g + 1.6$ transmissions are necessary to transmit g l.i. packets. To verify this, we will run the examples many

times to collect this statistic. In order to not build the examples every time by using `python waf --run kodo-wifi-broadcast`, we use a feature of ns-3 for running scripts shown here where we set the required environment variables. To accomplish this, we just do:

```
python waf shell
```

Later, we type for testing:

```
./build/examples/kodo/ns3-dev-kodo-wifi-broadcast-debug
```

Here, if you see the regular example output, everything should have worked fine.

Note: The path for the built program may change depending on the way ns-3 was compiled in your system. We show the resulting path the way we compile it by following this tutorial.

Later, you can save the following bash script as `extra_packet_per_generation.bash` in your `~/ns-3-dev/` folder:

```
#!/bin/bash
#Check the number of extra transmission per generation

SUM=0
N=$1  # Number of runs
GENERATION_SIZE=$2  #  Generation size
#  For-loop with range for bash to run the experiment many times
#  and collect the total transmissions to get the average

for (( c=1; c<=${N}; c++ ))
do
   COMB=`./build/examples/kodo/ns3-dev-kodo-wifi-broadcast-debug | \
   grep "Total transmissions:" | cut -f5 -d\ `
   SUM=$(( ${SUM} + ${COMB} ))
done

EXTRA=`echo "scale= 4; (${SUM} / ${N}) - ${GENERATION_SIZE}" | bc`
echo "Extra packets per generation: ${EXTRA}"
```

To set the permissions for this file, type in your terminal:

```
chmod 755 extra_packet_per_generation.bash
```

This enables you and others to run and read the script, but only you to write it. You can set this according to the needs in your system. For further permissions, you can refer to the `chmod` instruction for Unix-like systems.

The script receives two arguments: numbers of runs and generation size. Basically it returns how much extra packets per generation were necessary for decoding. Try to running as follows:

```
./extra_packet_per_generation.bash 100 5
Extra packets per generation: 1.3600
./extra_packet_per_generation.bash 1000 5
Extra packets per generation: 1.5770
./extra_packet_per_generation.bash 10000 5
Extra packets per generation: 1.5823
```

You can see that as we increase the amount of runs, we approach to 1.6 extra packets per generation. This is due to the linear dependency process of the coded packets. The numbers that you get might be different, but the tendency should be the same.

The previous result happens because we are using the binary field. Set the field to $q = 2^8$ by parsing `binary8` in the commandline arguments when calling `kodo-wifi-broadcast`. To do it so, in the line that says `COMB` in the bash script, include: `--field=binary8` after the simulation filename and rerun the script even with 100

samples. You will see that the amount of extra packets is zero (at least with 4 decimal places). This is because it is very unlikely to receive linearly dependent packets, even when the last coded packet is being sent.

To see the new coding coefficients for $q = 2^8$, but for only a generation size of 3 packets, type now:

```
./build/examples/kodo/ns3-dev-kodo-wifi-broadcast-debug --generationSize=3 --
↪field=binary8
```

Note: If you just want to run it in the common way, you can also do it but for command-line parsing to change the generation size, just type: `python waf --run kodo-wifi-broadcast --command-template="%s --generationSize=3 --field=binary8"`

You should see something similar to:

```
+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 134 44 251

decoder_state:
000 S:  1 21 169
001 ?:  0 0 0
002 ?:  0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 134 44 251

decoder_state:
000 S:  1 21 169
001 ?:  0 0 0
002 ?:  0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
C: 9 166 29

decoder_state:
000 S:  1 0 149
001 S:  0 1 65
002 ?:  0 0 0

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 9 166 29

decoder_state:
000 S:  1 0 149
001 S:  0 1 65
002 ?:  0 0 0

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
symbol_coefficients_before_read_symbol:
```

```
C: 144 87 3

decoder_state:
000 U:  1 0 0
001 U:  0 1 0
002 U:  0 0 1

Received a packet at Decoder 2
symbol_coefficients_before_read_symbol:
C: 144 87 3

decoder_state:
000 U:  1 0 0
001 U:  0 1 0
002 U:  0 0 1

Decoding completed! Total transmissions: 3
```

Notice how the size of the decoding matrix changes due to the effect of the generation size. This is expected because the size of the decoding matrix is given by the minimum amount of linear combinations required to decode. Also, you can verify the coding coefficients now vary between 0 and 255 given that we have changed the field size. Try running the example with these changes a couple of times so you can verify the above in general.

Changing the Receiver Signal Strength

As we mentioned earlier, our WiFi PHY layer relies on constant position and power values. We originally set up the `rss` value to -93 dBm to indicate our received power. In general, the packet error rate varies with the signal reception level, so we will adjust this. In this case, the receiver sensitivity is -96 dBm. It means that for `rss` values lower than this, we will have no packet recovery. This goes a little further from a typical erasure channel where we may or may not have packet losses regularly, the reason being that receiver position and power are both fixed.

To change the `rss` value , we can do it in the regular way as

```
python waf --run kodo-wifi-broadcast --command-template="%s --rss=-96"
```

You will see no output because the program gets into an infinite loop. To finish the program type `Ctrl+C` in your terminal. To verify that the running program ended, verify that a `^C` sign appears in your terminal (also `Interrupted` may appear as well). The program enters a loop because we receive no packets at all and the decoder will never be full rank.

Using Other Tracing Features

So far we have seen only the decoder state in terms of rank and symbol coefficients. In the constructor, just after, we create the decoder instances in the `kodo-broadcast.h` file, you can comment the callback and its setting in each decoder and just add the line `decoder.set_trace_stdout ();`. With this setting you will remove all the filters and see the full decoder trace in the simulation. To avoid a many prints, we will use a low generation and field size with 1 user in the binary field. To do so, set the field again to `kodo_binary` in `kodo-wifi-broadcast.cc` for the field type, save your files, rebuild and type:

```
python waf --run kodo-wifi-broadcast --command-template="%s --generationSize=2 --
↪users=1"
```

Then, you will get an output like the following:

```
set_mutable_symbols:
size: 2000
mutable_partial_shallow_symbol_storage:
Has partial symbol = 0
```

```
+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
rank_before_decode:
Rank = 0

symbol_data_before_read_symbol:
0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
....
03e0

symbol_coefficients_before_read_symbol:
C: 1 1

decoding_status_tracker:
Update symbol index = 0 status from ? to S

decoder_state:
000 S:  1 1
001 ?:  0 0

symbol_data_after_read_symbol:
0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
....
03e0

symbol_coefficients_after_read_symbol:
C: 1 1

rank_after_decode:
Rank = 1

+----------------------+
|Sending a coded packet|
+----------------------+
Received a packet at Decoder 1
rank_before_decode:
Rank = 1

symbol_data_before_read_symbol:
0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
....
03e0

symbol_coefficients_before_read_symbol:
C: 1 0

decoding_status_tracker:
Update symbol index = 1 status from ? to S

decoding_status_tracker:
Update symbol index = 0 status from S to U

decoding_status_tracker:
Update symbol index = 1 status from S to U

decoder_state:
000 U:  1 0
001 U:  0 1
```

```
symbol_data_after_read_symbol:
0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
....
03e0

symbol_coefficients_after_read_symbol:
C: 0 1

rank_after_decode:
Rank = 2

Decoding completed! Total transmissions: 2
```

Now, we see the data in rows of 16 bytes. To the right of the packets, you can see the converted ASCII character. In this way, you can observe that we constantly fill the buffer with empty data, since the example is just for showing purposes.

We first store our two raw symbols in a mutable symbol storage meaning that its content may vary since we will perform elementary row operations on them. The memory handling of a given symbol storage can be in different states depending on how the memory is assigned in Kodo. In the library we have 2 types of memory assignment for object creation, i.e. we can create a shallow copy or a deep copy. For this implementation, we use a shallow copy by default since it is the one that expenses the less amount of memory by keeping to a minimum the amount of duplications.

As you can see, the process in this mode is quite verbose. We can each detail of the decoding when a coded symbol (packet) is read in every step. Also, we can see the state of the coding matrix before and after each read. Similarly, we can check how is the variation of eah pivot from one reception to another.

Finally, try disabling the decoder trace and enable the encoder trace. This trace only has the symbol storage feature. Simply add `m_encoder.set_trace_stdout ()` the creation of the encoder, save, rebuild your project and rerun the example with the previous setting, you will only see your data in the encoder. Go back to the filters settings and try to also add or remove some of the parameters in the filters. For example add "symbol_data_after_read_symbol" in the filters and verify that the output matches your expectations.

Review pcap Traces

As we described earlier, the simulation leaves pcap format files (`kodo-wifi-broadcast-*-*.pcap`) in your `~/ns-3-dev/` folder. You can read these files with different programs like tcpdump or Wireshark. tcpdump is standard on most Unix-like systems and is based on the libpcap library. Wireshark is another free, open-source packet analyzer which you can get online. Just for showing purposes we will use tcpdump, but you can choose the one you prefer the most. For reading both files, simply type in the respective folder:

```
tcpdump -r kodo-wifi-broadcast-0-0.pcap -nn -tt
tcpdump -r kodo-wifi-broadcast-1-0.pcap -nn -tt
```

You will get this output (it will look different on your terminal):

```
reading from file wifi-simple-adhoc-0-0.pcap, link-type IEEE802_11_RADIO
(802.11 plus radiotap header)
1.000000 1000000us tsft 1.0 Mb/s 2412 MHz 11b IP 10.1.1.1.49153 >
10.1.1.255.80: UDP, length 1002
2.000000 2000000us tsft 1.0 Mb/s 2412 MHz 11b IP 10.1.1.1.49153 >
10.1.1.255.80: UDP, length 1002
3.000000 3000000us tsft 1.0 Mb/s 2412 MHz 11b IP 10.1.1.1.49153 >
10.1.1.255.80: UDP, length 1002
4.000000 4000000us tsft 1.0 Mb/s 2412 MHz 11b IP 10.1.1.1.49153 >
10.1.1.255.80: UDP, length 1002

reading from file kodo-wifi-broadcast-1-0.pcap, link-type IEEE802_11_RADIO
```

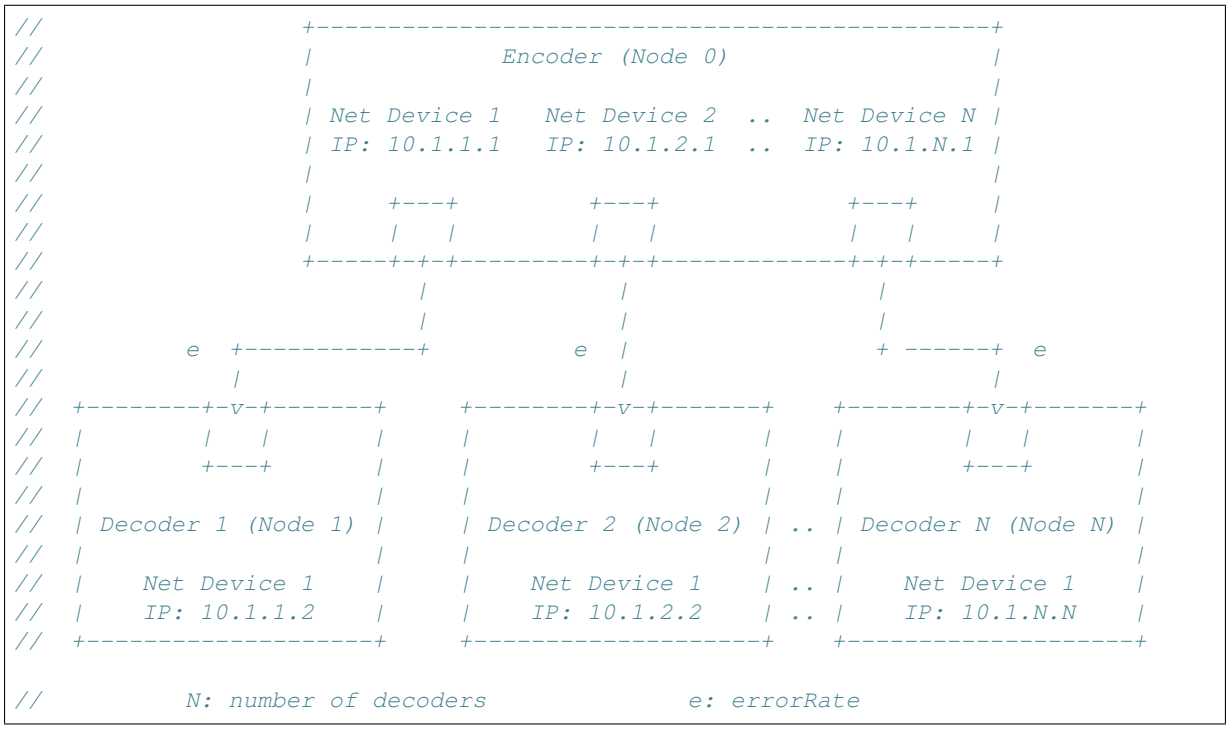
```
(802.11 plus radiotap header)
1.008720 1008720us tsft 1.0 Mb/s 2412 MHz 11b -93dB signal -101dB noise
IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1002
2.008720 2008720us tsft 1.0 Mb/s 2412 MHz 11b -93dB signal -101dB noise
IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1002
3.008720 3008720us tsft 1.0 Mb/s 2412 MHz 11b -93dB signal -101dB noise
IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1002
4.008720 4008720us tsft 1.0 Mb/s 2412 MHz 11b -93dB signal -101dB noise
IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1002
```

The `0-0` file stands for the encoder net device and `1-0` for the receiver net device. There you can confirm the RadioTap format of the pcap files and also can check other features like bit rate, frequency channel, protocol used, rss, noise floor and the transmitter and receiver IP addresses with their respective ports. Notice that these fit with our settings configuration.

CHAPTER 3

Broadcast RLNC with a N-user Erasure Channel

This example is similar to the first, but now the channel will be modeled with an erasure rate. In this way, the topology considered describes a transmitter sending coded packets with RLNC from a generation size g, field size q in a broadcast erasure channel to N receivers. As with the previous example, we will start with $g = 5$, $q = 2$, $N = 2$ and we will again observe completion time in terms of transmissions. Although, now we include the erasure rate (in percentage), $0 \leq \epsilon < 1$, to indicate packet losses. For this case, we will assume that all links have the same erasure rate for simplicity, $\epsilon = 0.3$, i.e. 30% packet losses. Topology is shown as follows:



What to Simulate

- Behavior: The sender keeps transmitting the generation until all receivers has g linearly independent (l.i.) coded packets. Packets might or might not be loss at the given rate.

- Inputs: Main parameters will be generation size, field size and packet loss rate.
- Outputs: A counter to indicate how much transmissions did the process required and some prints to indicate when decoding is completed. The number of transmissions should change as we vary the input parameters.
- Scenarios: We will variate the generation and field size to verify theoretical expected values regarding the amount of transmissions to decode.

Program Description

In the `~/ns-3-dev/examples/kodo` folder you should have a `kodo-wired-broadcast.cc` file which contains the source code of this simulation. Its structure is similar to the previous one, so now we will focus on the main differences.

Default Parameters and Command-line Parsing

For the default parameters, we show what has been added for the erasure rate:

```
// Main parameters
double errorRate = 0.3; // Error rate for all the links

// Command parsing
cmd.AddValue ("errorRate", "Packet erasure rate for the links", errorRate);
```

Topology and Net Helpers

For this part, there are some changes because we have removed the WiFi protocol and we have represented our channel as a packet erasure channel. For creating the topology, we proceed in a different way than the used for the first example. We use the `PointToPointHelper` to create a point-to-point link type. Then, we create the links from the source to each receiver using the `PointToPointStarHelper` which takes as an input the desired number of links and a `PointToPointHelper` instance, namely `pointToPoint` in our case. After that, we create the error rate model for each net device in the topology, configure and set it to affect the packets and enable them. Finally, we set up the Internet stack and IP addresses to our topology.

```
1  // Set the basic helper for a single link
2  PointToPointHelper pointToPoint;
3
4  // N receivers against a centralized hub.
5  PointToPointStarHelper star (users, pointToPoint);
6
7  // Set error model for the net devices
8  Config::SetDefault ("ns3::RateErrorModel::ErrorUnit",
9    StringValue ("ERROR_UNIT_PACKET"));
10
11 std::vector<Ptr<RateErrorModel>> errorModel (users,
12   CreateObject<RateErrorModel> ());
13
14 for (uint32_t n = 0; n < users; n++)
15 {
16   errorModel[n]->SetAttribute ("ErrorRate", DoubleValue (errorRate));
17   star.GetSpokeNode (n)->GetDevice (0)->SetAttribute ("ReceiveErrorModel",
18     PointerValue (errorModel[n]));
19   errorModel[n]->Enable ();
20 }
21
22 // Setting IP protocol stack
23 InternetStackHelper internet;
24 star.InstallStack (internet);
```

```
25
26   // Set IP addresses
27   star.AssignIpv4Addresses (Ipv4AddressHelper ("10.1.1.0", "255.255.255.0"));
```

The remaining elements of the simulation are very similar to the first example, you can review them and check the differences.

Simulation Runs

Default Run

Given that now we have an erasure rate different from zero and we can control it, packet reception will differ randomly. Then, as a default we disable the decoder tracing and keep the reception prints in order to check just when a packet has arrived to each receiver. After building the project, run the example just as before by typing:

```
python waf --run kodo-wired-broadcast
```

You will get an output like this:

```
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 1
Received a packet at decoder 2
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 1
Received a packet at decoder 2
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 1
Received a packet at decoder 2
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 2
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 1
Received a packet at decoder 2
+---------------------+
|Sending a combination|
+---------------------+
Received a packet at decoder 1
Received a packet at decoder 2
Decoding completed! Total transmissions: 6
```

Now we can see when a packet is received at each decoder. As expected, a packet is sent every time slot to both decoders and the process stops when both decoders have g l.i. combinations. We can observe this behavior in the previous output. At the 4th transmission, receiver 1 did not get the combination although receiver 2 did. Nevertheless, this is compensated in the last transmission where receiver 1 gets its remaining combination. Besides, receiver 2 gets a non-innovative extra combination which occurs for the packet being sent to both decoders.

Again, we can verify for a broadcast with coding scenario that on average we need 9.4847 transmissions for $q = 2, g = 5, \epsilon = 0.3$ and 2 users. To verify it, save the following script as `mean_packets.bash` and follow the same procedure as before. As you will notice, it is a modification of the script used in the first example.

Most likely you will not need to do `python waf shell` if you already made it before. Just remember that the purpose of this command is to run the script many times without rebuilding.

```
#!/bin/bash

#Check the number of extra transmission per generation

SUM=0
N=$1   # Number of runs

#  For-loop with range for bash
#  Basically run the experiment several times and collect the total
#  transmissions to get the average

for (( c=1; c<=${N}; c++ ))
do
    COMB=`./build/examples/kodo/ns3-dev-kodo-wired-broadcast-debug | \
          grep "Total transmissions:" | cut -f5 -d\ `
    SUM=$(( ${SUM} + ${COMB} ))
done

BROADCAST_MEAN=`echo "scale= 4; (${SUM} / ${N})" | bc`

echo "Wired broadcast example mean: ${BROADCAST_MEAN}"
```

To change its settings do a `chmod 755 extra_packet_per_generation.bash`. Run it in a similar way as first script. You will get an output similar to this:

```
./mean_packets.bash 1
Wired broadcast example mean: 9.0000
./mean_packets.bash 10
Wired broadcast example mean: 12.0000
./mean_packets.bash 100
Wired broadcast example mean: 10.2500
./mean_packets.bash 1000
Wired broadcast example mean: 10.0200
./mean_packets.bash 10000
Wired broadcast example mean: 9.5914
```

As we check, by increasing the numbers of runs we see that the mean number of transmissions to decode in a pure broadcast RLNC for two receivers, converges to 9.4847 transmissions for the previous setting. We will also set some parameters to observe the difference in the total number of transmissions.

Changing the Field Size

Set `binary8` as the field size in the string parsing as described previously and rerun the previous script. You will get an output similar to this:

```
./mean_packets.bash 1
Wired broadcast example mean: 8.0000
./mean_packets.bash 10
Wired broadcast example mean: 6.0000
./mean_packets.bash 100
Wired broadcast example mean: 8.5600
./mean_packets.bash 1000
Wired broadcast example mean: 8.2630
./mean_packets.bash 10000
Wired broadcast example mean: 8.2335
```

Now we observe that the amount of transmissions reduces to less than 9 transmissions on average. Similarly as with the WiFi example, in this case the decoding probability increases with a higher field size for each decoder.

This ensures that, on average, each decoder requires less transmissions to complete decoding.

Changing the Packet Erasure Rate

One interesting feature that we have added is a `RateErrorModel` which basically includes a packet error rate at each receiver. Currently we have set the error rates to be both the same and a 30% loss rate is set by default. Keep `binary8` as a field size in order to exclude retransmissions due to linear dependency and account them only for losses. As we saw, with 30% losses we see an average of 8.2335 transmissions (for 10000 example runs). Now, we will check that by adjusting the loss rate (with the same amount of runs). So, we just need to make some small modifications of our bash script.

We will add a new input parameter to set the loss rate and call it in the script as follows:

```
#!/bin/bash

#Check the number of extra transmission per generation

SUM=0
N=$1  # Number of runs
LOSS_RATE=$2  # Loss rate for both links

#  For-loop with range for bash
#  Basically run the experiment several times and collect the total
#  transmissions to get the average

for (( c=1; c<=${N}; c++ ))
do
    COMB=`./build/examples/kodo/ns3-dev-kodo-wired-broadcast-debug \
          --errorRate=${LOSS_RATE} | grep "Total transmissions:" | \
          cut -f5 -d\ `
    SUM=$(( ${SUM} + ${COMB} ))
done

BROADCAST_MEAN=`echo "scale= 4; (${SUM} / ${N})" | bc`

echo "Wired broadcast example mean: ${BROADCAST_MEAN}"
```

Save the changes in the script. Then, let us observe the output with 10% and 50% losses in both links:

```
./mean_packets.bash 100000 0.1
Wired broadcast example mean: 6.0049
./mean_packets.bash 100000 0.5
Wired broadcast example mean: 9.0138
```

For the required erasures rate, we observe that if we modify the erasure rate in the links, the expected number of transmissions changes. By increasing the error rate, we need more transmissions to overcome the losses.

Review pcap Traces

We have added also a trace file per each net device in order to observe packet routing.

```
> tcpdump -r wired-broadcast-0-0.pcap -nn -tt
reading from file wired-broadcast-0-0.pcap, link-type PPP (PPP)
1.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
2.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
3.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
4.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
5.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
6.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
7.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
```

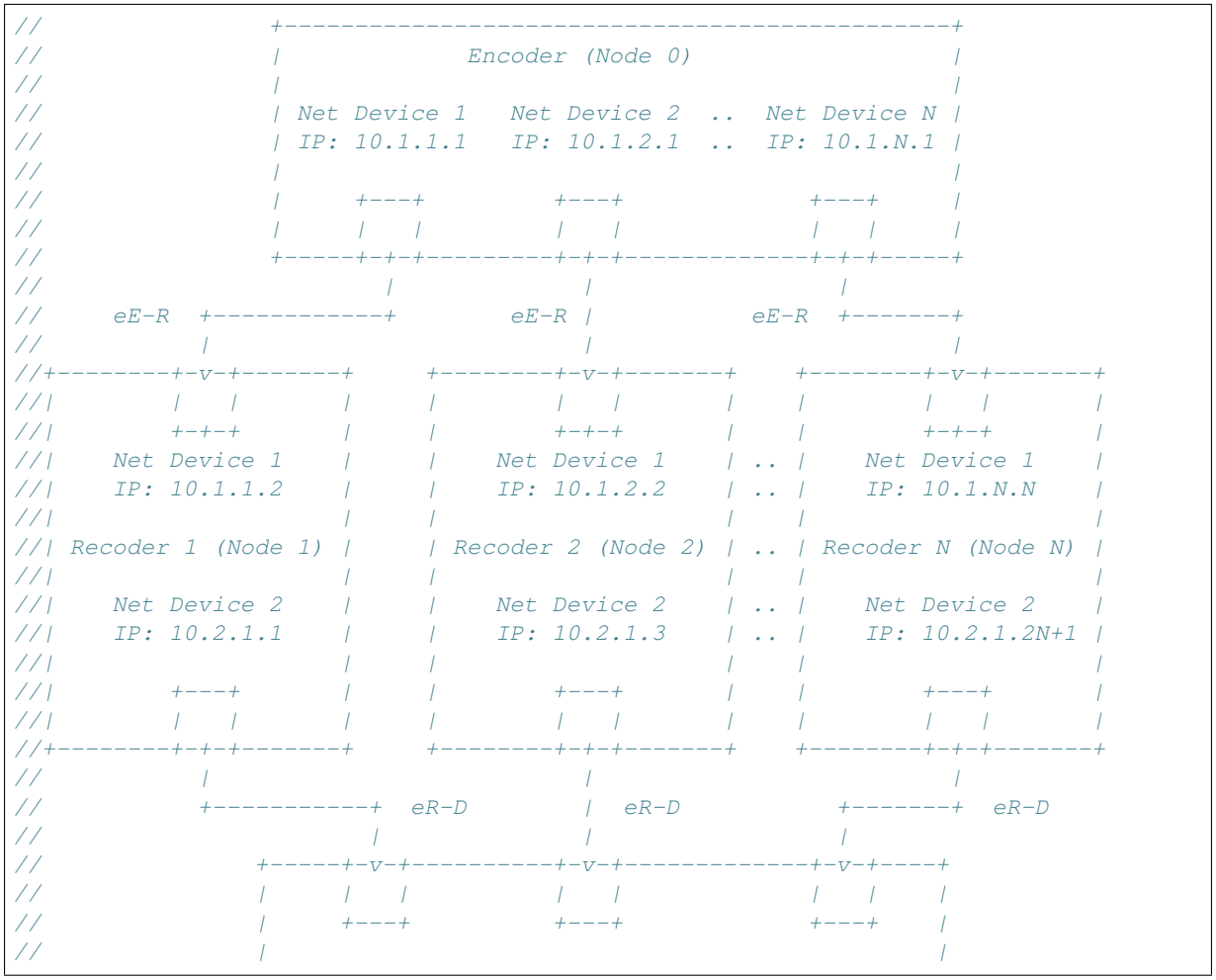
```
8.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
9.000000 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
> tcpdump -r wired-broadcast-0-1.pcap -nn -tt
reading from file wired-broadcast-0-1.pcap, link-type PPP (PPP)
1.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
2.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
3.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
4.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
5.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
6.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
7.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
8.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
9.000000 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
> tcpdump -r wired-broadcast-1-0.pcap -nn -tt
reading from file wired-broadcast-1-0.pcap, link-type PPP (PPP)
1.252929 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
2.252929 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
6.252929 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
8.252929 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
9.252929 IP 10.1.1.1.49153 > 10.1.1.255.80: UDP, length 1006
> tcpdump -r wired-broadcast-2-0.pcap -nn -tt
reading from file wired-broadcast-2-0.pcap, link-type PPP (PPP)
1.252929 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
2.252929 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
6.252929 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
7.252929 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
8.252929 IP 10.1.2.1.49153 > 10.1.2.255.80: UDP, length 1006
```

From the trace files we see the broadcast nature set before. We have two net devices in the transmitter given that the `PointToPointStarHelper` creates a point-to-point link at each receiver. By setting `SetAllowBroadcast (true)` in the transmitter socket, we ensure to be using the broadcast channel on the source node.

CHAPTER 4

Recoders with Erasure Channels

This example considers a transmitter sending coded packets with RLNC from a generation size g and field size q to a decoder. Packets are set through N relays in a broadcast erasure channel. Each link from the encoder to the relays has the same erasure rate ϵ_{E-R}. In the second hop, all the links from the relays to the decoder have a common erasure rate ϵ_{R-D}. By default, we consider: $g = 5$, $q = 2^8$, $N = 2$, $\epsilon_{E-R} = 0.4$ and $\epsilon_{R-D} = 0.2$. Topology is shown as follows:



```
//                |             Decoder (Node N+1)                |
//                |                                               |
//                | Net Device 1   Net Device 2  ..  Net Device N |
//                | IP: 10.2.1.2   IP: 10.2.1.4  ..  IP: 10.2.1.2N|
//                +-----------------------------------------------+

//                            N: Number of recoders
//                            eE-R: errorRateEncoderRecoders
//                            eR-D: errorRateRecodersDecoder
```

What to Simulate

- Behavior: The sender keeps transmitting the generation until the all the relays have g linearly independent (l.i.) coded packets. The relays keep sending packets until the decoder has received this data. As with previous examples, packets might or might not be loss at the given erasure rates in the respective links. Additionally, we have 2 transmission policies depending if recoding is enabled or not. If recoding is enabled (default), received packets in each recoder are recoded to create new coded packets which are sent to the decoder. If recoding is disabled, any previously received packet is forwarded to the decoder. The packet to be forwarded is selected at random. To avoid collisions from the relays, all the relays access the medium to transmit with probability p. If two or more relays access the medium at the same time, then event simulator schedules them to transmit in a non-colliding order.
- Inputs: Main parameters will be generation size, field size, number of relays, packet losses in each hop, a boolean flag for the recoding policy and a transmit probability to access the shared medium.
- Outputs: Two counters to indicate how much transmissions did the process required and some prints to indicate when decoding is completed. The number of transmissions should change as we vary the input parameters.
- Scenarios: We will variate the generation and field size to verify theoretical expected values regarding the amount of transmissions to decode.

Program Description

In your `~/ns-3-dev/examples/kodo` folder, you will see the `kodo-recoders.cc` file which contains the source code of this simulation. Its structure is similar to previous simulations, so again we will focus on the main differences.

Header Includes

```
#include "kodo-recoders.h"
```

The `Recoders` class in `kodo-recoders.h` is used to model the expected behavior of the nodes in our simulation.

Simulation Class

The `Recoders` class can be roughly defined in the following way:

```
class Recoders
{
public:

  Recoders (const kodocpp::codec codeType, const kodocpp::field field,
```

```
    const uint32_t users, const uint32_t generationSize, const uint32_t packetSize,
    const std::vector<ns3::Ptr<ns3::Socket>>& recodersSockets,
    const bool recodingFlag)
    : m_codeType (codeType),
      m_field (field),
      m_users (users),
      m_generationSize (generationSize),
      m_packetSize (packetSize),
      m_recodingFlag (recodingFlag),
      m_recodersSockets (recodersSockets)
  {
    // Constructor
  }

  void SendPacketEncoder (ns3::Ptr<ns3::Socket> socket, ns3::Time pktInterval)
  {
    // Encoder logic
  }

  void ReceivePacketRecoder (ns3::Ptr<ns3::Socket> socket)
  {
    // Recoders logic for reception
  }

  void SendPacketRecoder (ns3::Ptr<ns3::Socket> socket, ns3::Time pktInterval)
  {
    // Recoders logic for transmission
  }

  void ReceivePacketDecoder (ns3::Ptr<ns3::Socket> socket)
  {
    // Decoder logic
  }

private:

  const kodocpp::codec m_codeType;
  const kodocpp::field m_field;
  const uint32_t m_users;
  const uint32_t m_generationSize;
  const uint32_t m_packetSize;
  const bool m_recodingFlag;

  kodocpp::encoder m_encoder;
  std::vector<uint8_t> m_encoderBuffer;
  std::vector<kodocpp::decoder> m_recoders;
  std::vector<std::vector<uint8_t>> m_recoderBuffers;
  kodocpp::decoder m_decoder;
  std::vector<uint8_t> m_decoderBuffers;
  std::vector<ns3::Ptr<ns3::Socket>> m_recodersSockets;

  std::vector<uint8_t> m_payload;
  uint32_t m_encoderTransmissionCount;
  uint32_t m_recodersTransmissionCount;
  uint32_t m_decoderRank;
  std::map<uint32_t, std::map<uint32_t, ns3::Ptr<ns3::Packet>>> m_previousPackets;

  const double m_transmitProbability;
  ns3::Ptr<ns3::UniformRandomVariable> m_uniformRandomVariable;
};
```

The `Recoders` design is similar as the one for `Broadcast`. Still, some differences exist. For this case

`kodocpp::decoder` is the type for the relays since it behaves in the same way.

Also, now we add different functions for what a relay might perform. Hereby, we include `SendPacketRecoder` and `ReceivePacketRecoder` to split the functionality of recoding (or forwarding) and receiving with decoding. The recoding functionality is performed again with `recoder.write_payload()`.

For control variables, for the recoding or forwarding behavior, we included a boolean as a construction argument. Also we keep track of the decoder rank with a counter, in order to notify when a coded packet is received at the decoder. Furthermore, for each of the relays we store its received packets so we can forward one of them at random later.

For the medium probability access p, we use samples from a `ns3::Ptr<ns3::UniformRandomVariable>` and convert them to samples of a Bernoulli random variable at a given transmission using the Inverse Transmformating Sampling. In this way, we guarantee that a node attemps a medium access with the desired probability. Finally, we include a counter for the total number of transmissions from all the relays for counting total transmissions in general.

Default Parameters and Command-line Parsing

For the default parameters, we show what has been added for this example:

```
// Main parameters
double errorRateEncoderRecoder = 0.4; // Error rate for encoder-recoder link
double errorRateRecoderDecoder = 0.2; // Error rate for recoder-decoder link
bool recodingFlag = true; // Flag to control recoding
uint32_t recoders = 2; // Number of recoders
std::string field = "binary"; // Finite field used
double transmitProbability = 0.5; // Transmit probability for the relays

// Command parsing
cmd.AddValue ("errorRateEncoderRecoder",
              "Packet erasure rate for the encoder-recoder link",
              errorRateEncoderRecoder);
cmd.AddValue ("errorRateRecoderDecoder",
              "Packet erasure rate for the recoder-decoder link",
              errorRateRecoderDecoder);
cmd.AddValue ("recodingFlag", "Enable packet recoding", recodingFlag);
cmd.AddValue ("recoders", "Amount of recoders", recoders);
cmd.AddValue ("field", "Finite field used", field);
cmd.AddValue ("transmitProbability", "Transmit probability from recoder",
              transmitProbability);
```

Topology and Net Helpers

```
  // We group the nodes in different sets because
  // we want many net devices per node. For the broadcast
  // topology we create a subnet and for the recoders to
  // the decoders, we create a secondary one. This in order to
  // properly set the net devices and socket connections later


  // First we set the basic helper for a single link.
  PointToPointHelper ptp;

  // Encoder to recoders
  PointToPointStarHelper toRecoders (recoders, ptp);
  NodeContainer decoder;
  decoder.Create (1);

```

```
  // Recoders to decoder
  NetDeviceContainer recodersDecoderDev;

  for (uint32_t n = 0; n < recoders; n++)
   {
      recodersDecoderDev.Add (ptp.Install (
        NodeContainer (toRecoders.GetSpokeNode (n), decoder.Get (0))) );
   }

  // Internet stack for the broadcast topology and decoder
  InternetStackHelper internet;
  toRecoders.InstallStack (internet);
  internet.Install (decoder);

  // Here, we first create a total of N net devices in the encoder
  // (N = recoders amount) and a net device per recoder
  // Second, we mirror this procedure in the second hop.
  // Each net device in the recoder is in a different subnet.

  toRecoders.AssignIpv4Addresses (Ipv4AddressHelper ("10.1.1.0",
    "255.255.255.0"));

  // The IP set of the recoders to the decoder is calculated
  // in order to not collide with the one from the encoder
  // to the recoders.
  Ipv4AddressHelper fromRecoders ("10.2.1.0", "255.255.255.0");
  fromRecoders.Assign (recodersDecoderDev);
```

In order to be able to construct various net devices in the relays from the helpers, we separate the relays in 2 subnets: a one-to-many subnet which covers the encoder and the relays, and a many-to-one subnet which covers the relays and the decoder. For the one-to-many (broadcast) subnet, we use the `PointToPointStarHelper` and for the many-to-one we create the net devices with the `Install` member function of the `PointToPointHelper` and store it in a container for easy IP address assignment. Then, we assign the IP addresses as shown in the previous topology figure. For the one-to-many, we use the `"10.1.X.X"` subnet and for the many-to-one, the `"10.2.1.X"` subnet.

Simulation Event Handler

```
  // Recoders
  for (auto recoderSocket : recodersSockets)
    {
      Simulator::ScheduleWithContext (recoderSocket->GetNode ()->GetId (),
        Seconds (1.5), &Recoders::SendPacketRecoder,
        &multihop, recoderSocket, interPacketInterval);
    }
```

Now we aggregate the generation of coded packets from the relays to the scheduling process. We send packets from each recoder independently from previously having received a packet. However, the recoder will only send coded packets from the coded packets that it has l.i. packets and it sucessfully access the medium. If some relays send their coded packet at, the same time, the event scheduling organizes them properly.

Simulation Runs

Default Run

To run the default simulation, just type:

```
python waf --run kodo-recoders
```

You will see an output similar to this:

```
+----------------------------------+
|Sending a coded packet from ENCODER|
+----------------------------------+
Received a packet at RECODER 2
+-----------------------------------+
|Sending a coded packet from RECODER 2|
+-----------------------------------+
Received an innovative packet at DECODER!
Decoder rank: 1

+----------------------------------+
|Sending a coded packet from ENCODER|
+----------------------------------+
Received a packet at RECODER 1
+-----------------------------------+
|Sending a coded packet from RECODER 1|
+-----------------------------------+
+-----------------------------------+
|Sending a coded packet from RECODER 2|
+-----------------------------------+
Received an innovative packet at DECODER!
Decoder rank: 2

+----------------------------------+
|Sending a coded packet from ENCODER|
+----------------------------------+
Received a packet at RECODER 1
+-----------------------------------+
|Sending a coded packet from RECODER 1|
+-----------------------------------+
+-----------------------------------+
|Sending a coded packet from RECODER 2|
+-----------------------------------+
Received an innovative packet at DECODER!
Decoder rank: 3

*** Decoding completed! ***
Encoder transmissions: 3
Recoders transmissions: 5
Total transmissions: 8
```

From the simulation output, it can be seen that in the first transmission only recoder 2 got the coded packet from the source and it conveyed properly to the decoder. Here recoder 1 does not make any transmissions since it does not have any possible information to convey. For the second and third transmission, recoder 1 got the packet and conveyed properly to the decoder. Observe that for the second and third transmissions, recoder 2 did not get any coded packets but it still tries to send them. However, it will only be possible to send only one degree of freedom given that its set of packets only allow this. Whenever it receives more combinations, it will be possible for it to send more. You can modify the number of relays and erasure rates in the hops to check the effects in the number of transmissions. Also, you may verify the pcap traces as well. We invite you to modify the parameters as you might prefer to verify your intuitions and known results.