Simple Propagation Loss Model on ns-3

Brief summary

In this class we implement a naive loss propagation model which depends exclusively on the distance (in meters) between the source station and the packet sink (receiver/destination). This measurement is brought about by the shortest path between the two elements, namely the linear distance between both of them.

This model behaves as a deterministic channel model, in which all packet are successfully received at the sink if the distance between the two involved nodes are smaller than a preconfigured value (namely, $\alpha \cdot d_{max}$). Once this point is reached, the *FER* (Frame Error Rate) begins to increase, until the maximum allowed distance (a.k.a. d_{max}). From this point, every frame will be discarded by the station.

In our model, we will consider that

* * This model corresponds with a deterministic channel model behavior, where all packets arrive * correctly to the receiver if the distance between the two nodes are smaller than a preset value. * Once this value is reached, the error rate begins to increase (following a beta-dependent decreasing * factor), according to the following expression: *

* * Receivers beyond MaxRange receive at power -1000 dBm (effectively zero).

Analytical analysis

In order to get the desired channel behaviour, we have carried out the following expression, which aims at modelling a channel which behaves in an ideal way when the nodes are closer than a configurable threshold (let us consider it as $\alpha \cdot d_{max}$); once trespassed this value, the channel will experiment propagation errors, following a decreasing function, until they reach a distance (d_{max}), from which all packets will be considered as errors.

$$\text{1 - FER} = \begin{cases} 1, if 0 \leqslant x < \alpha \cdot d_{max} \\ \frac{1-(\frac{x}{d_{max}})^{\beta}}{1-\alpha^{\beta}}, if \alpha \cdot d_{max} \leqslant x \leqslant d_{max} \\ 0, if d_{max} < x \end{cases} \tag{1}$$

Where:

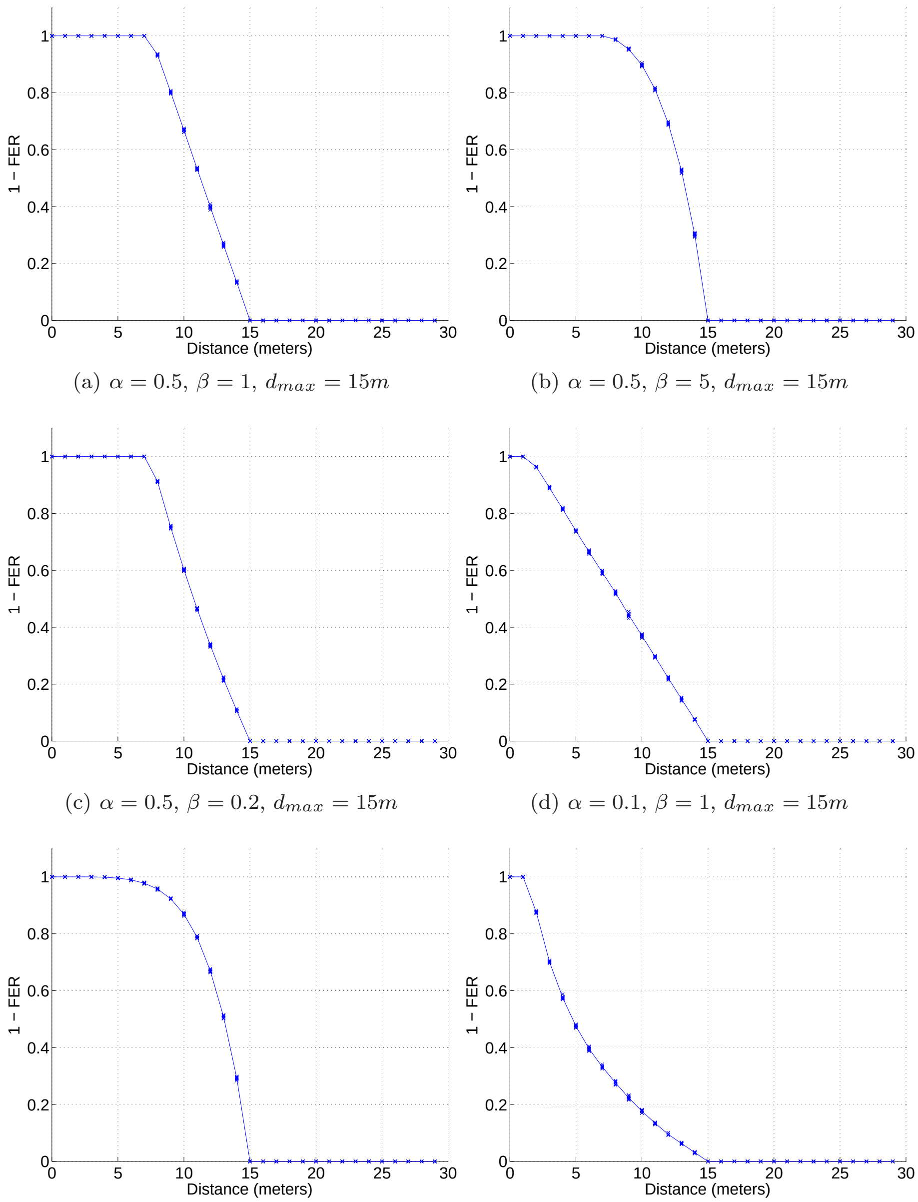
- x. Actual distance between the involved nodes
- α. distance (in meters) from which the transmission is set over an error-prone channel.
- β. Exponential parameter (1 for a linear behavior).
- d_{max}. The distance from which all packets will be errored (FER = 1).

Results

Code Annex

Header file

```
#ifndef SIMPLE_PROPAGATION_LOSS_MODEL_H
#define SIMPLE_PROPAGATION_LOSS_MODEL_H

#include "ns3/object.h"
#include "ns3/random-variable.h"
#include <map>

namespace ns3{
/**
 * \brief Naive propagation model which depends exclusively on the distance between the source and the destination
 * (shortest distance - linear path).
 *
 * This model corresponds with a deterministic channel model behavior, where all packets arrive
 * correctly to the receiver if the distance between the two nodes are smaller than a preset value.
 * Once this value is reached, the error rate begins to increase (following a beta-dependent decreasing
 * factor), according to the following expression:
 *
 *          /                 1,            if    0 <= x < alpha * d_max
 *          |
 *          |       1 - (x/(d_max))^beta
 *  1 - FER =  <      ------------------------ ,       if      alpha * d_max <= x <= d_max
 *          |     1 - alpha^beta
 *          |
 *          \             0,            if    d_max < x
 *
 * Receivers beyond MaxRange receive at power -1000 dBm (effectively zero).
 */

class SimplePropagationLossModel: public PropagationLossModel
{
public:
  static TypeId GetTypeId(void);

  SimplePropagationLossModel();
  virtual ~SimplePropagationLossModel();

  /**
   *  \param alpha the exponential parameter applied in the expression
   */
  void SetAlpha(float alpha);
  /**
   *  \returns the exponential parameter (alpha) to be used in the model
   */
  float GetAlpha(void) const;
  /**
   *  \param beta the exponential parameter applied in the expression
```



(a) $\alpha = 0.5$, $\beta = 1$, $d_{max} = 15m$

(b) $\alpha = 0.5$, $\beta = 5$, $d_{max} = 15m$

(c) $\alpha = 0.5$, $\beta = 0.2$, $d_{max} = 15m$

(d) $\alpha = 0.1$, $\beta = 1$, $d_{max} = 15m$

(e) $\alpha = 0.1$, $\beta = 5$, $d_{max} = 15m$

(f) $\alpha = 0.1$, $\beta = 0.002$, $d_{max} = 15m$

Figure 1: Simple propagation loss model testbed results

```
   */
  void SetBeta(float beta);
  /**
   *  \returns the exponential parameter (beta) to be used in the
       model
   */
  float GetBeta(void) const;
  /**
   *  \param alpha the exponential parameter applied in the
       expression
   */
  void SetMaxDistance(float maxDistance);
  /**
   * \returns the maximum distance for a wireless transmission in
       the model. From this distance, FER = 1
   */
  float GetMaxDistance(void) const;

private:
  SimplePropagationLossModel (const SimplePropagationLossModel& o);
  SimplePropagationLossModel & operator=(const
      SimplePropagationLossModel& o);
  virtual double DoCalcRxPower(double txPowerDbm, Ptr<MobilityModel
      > a,
      Ptr<MobilityModel> b) const;

  //class specific parameters
  float m_maxDistance;
  float m_alpha;
  float m_beta;
  RandomVariable m_ranvar;

};


} //namespace ns3

#endif  /* SIMPLE_PROPAGATION_LOSS_MODEL_H */

%\lstinputlisting[language=C++]{model/simple-propagation-loss-model
    .cc}
```

Source file

```
#include "propagation-loss-model.h"
#include "ns3/log.h"
#include "ns3/mobility-model.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include <math.h>

NS_LOG_COMPONENT_DEFINE ("SimplePropagationLossModel");

namespace ns3
{

NS_OBJECT_ENSURE_REGISTERED (SimplePropagationLossModel);
```

```
TypeId
SimplePropagationLossModel::GetTypeId(void) {
  static TypeId tid = TypeId ("ns3::SimplePropagationLossModel")
      .SetParent<PropagationLossModel> ()
      .AddConstructor<SimplePropagationLossModel> ()

  .AddAttribute("MaxDistance",
      "The distance from which all packets will be errored (FER =
          1)",
      DoubleValue(15.0),
      MakeDoubleAccessor(&SimplePropagationLossModel::m_maxDistance
          ),
      MakeDoubleChecker<double> ())

  .AddAttribute("Alpha", "Determines the distance (in meters) from
      which the transmission is set over an error-prone channel",
      DoubleValue(0.5),
      MakeDoubleAccessor(&SimplePropagationLossModel::m_alpha),
      MakeDoubleChecker<double> ())

  .AddAttribute("Beta", "Exponential parameter (1 for a linear
      behavior)",
      DoubleValue(1.0),
      MakeDoubleAccessor(&SimplePropagationLossModel::m_beta),
      MakeDoubleChecker<double> ())

  .AddAttribute ("RanVar", "Random variable which determine a
      packet to be successfully received or not.",
      RandomVariableValue (UniformVariable (0.0, 1.0)),
      MakeRandomVariableAccessor (&SimplePropagationLossModel::
          m_ranvar),
      MakeRandomVariableChecker ())

  ;
  return tid;
}


SimplePropagationLossModel::SimplePropagationLossModel ()
{
  NS_LOG_FUNCTION_NOARGS ();
}

SimplePropagationLossModel::~SimplePropagationLossModel ()
{
  NS_LOG_FUNCTION_NOARGS ();
}

float SimplePropagationLossModel::GetAlpha() const
{
  NS_LOG_FUNCTION_NOARGS ();
  return m_alpha;
}

void SimplePropagationLossModel::SetAlpha(float alpha)
{
  NS_LOG_FUNCTION_NOARGS ();
  m_alpha = alpha;
}

float SimplePropagationLossModel::GetBeta() const
{
```

```
  NS_LOG_FUNCTION_NOARGS ();
  return m_beta;
}

void SimplePropagationLossModel::SetBeta(float beta)
{
  NS_LOG_FUNCTION_NOARGS ();
  m_beta = beta;
}

float SimplePropagationLossModel::GetMaxDistance(void) const
{
  NS_LOG_FUNCTION_NOARGS ();
  return m_maxDistance;
}

void SimplePropagationLossModel::SetMaxDistance(float maxDistance)
{
  NS_LOG_FUNCTION_NOARGS ();
  m_maxDistance = maxDistance;
}

double SimplePropagationLossModel::DoCalcRxPower(double txPowerDbm,
     Ptr<MobilityModel> a,
    Ptr<MobilityModel> b) const
{
        NS_LOG_FUNCTION_NOARGS ();
  double distance = a->GetDistanceFrom (b);
  double fer;


  NS_ASSERT (distance >= 0);

  if (distance < m_alpha * m_maxDistance)
  {
    fer = 0;
  }
  else if (distance < m_maxDistance)
  {
    fer = 1 - ((1 - pow((distance / m_maxDistance), m_beta))/ (1 -
        pow(m_alpha, m_beta)));
  }
  else
  {
    fer = 1;
  }

  NS_ASSERT(fer <=1);

  NS_LOG_DEBUG ("FER =" << fer << " Distance = " << distance << "
      Max_distance = " << m_maxDistance << " Alpha = " << m_alpha
      << " Beta = " << m_beta);

  if(m_ranvar.GetValue() <= fer)
  {
    NS_LOG_DEBUG("Frame error");
    return -10000;
  }
  else
  {
    NS_LOG_DEBUG("Frame OK");
    return txPowerDbm;
```

```
  }
}
}   //namespace ns3
```

Test file

```
#include <stdio.h>
#include <string.h>
#include <fstream>

#include "ns3/propagation-loss-model.h"
#include "ns3/constant-position-mobility-model.h"
#include "ns3/config.h"
#include "ns3/string.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/gnuplot.h"
#include "ns3/simulator.h"
#include "ns3/core-module.h"

using namespace ns3;
using namespace std;

float maxDistance_g = 15.0;
float alpha_g = 0.5;
float beta_g = 1.0;


std::string getcwd() {
  char buf[FILENAME_MAX];
  char* succ = getcwd(buf, FILENAME_MAX);
  if (succ)
    return std::string(succ);
  return ""; // raise a flag, throw an exception, ...
}

void TestProbabilistic(Ptr<PropagationLossModel> model, double
    maxDistance,
    unsigned int samples = 10000) {
  Ptr<ConstantPositionMobilityModel> a = CreateObject<
      ConstantPositionMobilityModel>();
  Ptr<ConstantPositionMobilityModel> b = CreateObject<
      ConstantPositionMobilityModel>();

  string path;
  fstream fileOutput;

  u_int8_t nTestCounter=1;
  u_int8_t nTest = 10;

  double txPowerDbm = +20; // dBm
  double rxPowerDbm;

  u_int32_t packetOk;
  u_int32_t packetTotal;
```

```
  path = getcwd() + "/results/SimplePropagationLossModelTest_Type_6
      .txt";

  fileOutput.open(path.c_str(), fstream::out);

  fileOutput << "Alpha = " << alpha_g << ",\tBeta= " << beta_g << "
      ,\tMax Distance = " << maxDistance_g << ",\tSamples = " <<
      samples  \
      << ",\tNumber of tests = 10 " << endl;

  // Take given number of samples from CalcRxPower() and show
      probability
  // density for discrete distances.

  for (nTestCounter = 1; nTestCounter <= nTest; nTestCounter++) {
    a->SetPosition(Vector(0.0, 0.0, 0.0));

    for (double distance = 0; distance <= 2.0 * maxDistance;
        distance += 1.0) {
      packetOk = 0;
      packetTotal = 0;
      b->SetPosition(Vector(distance, 0.0, 0.0));

      for (unsigned int samp = 0; samp < samples; ++samp) {
        // CalcRxPower() returns dBm.
        rxPowerDbm = model->CalcRxPower(txPowerDbm, a, b);
        rxPowerDbm = rxPowerDbm;

        if (rxPowerDbm > 0) {
          packetOk++;
          packetTotal++;
        } else
          packetTotal++;

        Simulator::Stop(Seconds(0.01));
        Simulator::Run();
      }

      fileOutput << distance << "\t" << (double) (packetTotal -
          packetOk) / packetTotal << "\t" << packetOk << "\t" <<
          packetTotal <<endl;

    }

  }
  fileOutput.close();
}

int main(int argc, char *argv[]) {
  CommandLine cmd;

  Ptr<SimplePropagationLossModel> simpleProp = CreateObject<
      SimplePropagationLossModel>();

  cmd.AddValue("maxDistance",
      "Distance (in meters) from which all frames will be errored",
      maxDistance_g);
  cmd.AddValue(
      "Alpha",
      "Determines the distance (in meters) from which the
          transmission is set over an error-prone channel",
      alpha_g);
```

```
  cmd.AddValue("Beta", "Exponential parameter (1 for a linear
      behavior",
      beta_g);


  cmd.Parse(argc, argv);

  if (alpha_g != 0.5)
    simpleProp->SetAlpha(alpha_g);

  if (beta_g != 1.0)
    simpleProp->SetBeta(beta_g);

  if (maxDistance_g != 15.0)
    simpleProp->SetMaxDistance(maxDistance_g);

  TestProbabilistic(simpleProp, maxDistance_g);

  return 0;
}
```