Chapter 1

Code Listings

Used to check line length and formatting.

1.1 callback-example

```cpp
// -*- mode: c++; c-file-style: "gnu"; indent-tabs-mode: nil; -*-

#include "ns3/core-module.h"
#include <iostream>
#include <math.h>

using namespace ns3;

double AddInts(int a, int b)
{
  return a + b;
}

double AddDoubles(double a, double b)
{
  return a + b;
}

class Point
{
public:
  static double Norm(int a, int b)
  {
    return sqrt( a*a + b*b );
  }

  int x, y;

  double Distance(int a, int b)
  {
    return sqrt( (x-a)*(x-a) + (y-b)*(y-b) );
  }
};

int main()
{
  Callback<double, int, int> cb1;

  cb1 = MakeCallback(&AddInts);
  std::cout << cb1(2, 5) << std::endl;

  cb1 = MakeCallback(&Point::Norm);
  std::cout << cb1(2, 5) << std::endl;

  Point p1 = { 1, 1 };
  cb1 = MakeCallback(&Point::Distance, &p1);
  std::cout << cb1(2, 5) << std::endl;

  // not valid - throws compile time error:
  // cb1 = MakeCallback(&AddDoubles);
  // valid - different callback signature:
  Callback<double, double, double> cb2 = MakeCallback(&AddDoubles);
}

class Alpha
{
public:
  void ReceiveInput(double x);
};

class UsualLayer
{
```

```
public:
  Alpha* m_alpha;

  void DoWork()
  {
    double work = 5;
    m_alpha->ReceiveInput(work);
  }
};

class EnhancedLayer
{
public:
  Callback<void, double> m_receiveWork;

  void DoWork()
  {
    double work = 5;
    m_receiveWork(work);
  }
};
```

1.2 myprotocol-example

```
// -*- mode: c++; c-file-style: "gnu"; indent-tabs-mode: nil; -*-

#include <ns3/core-module.h>
#include <ns3/simulator-module.h>
#include <ns3/common-module.h>

using namespace ns3;

class MyProtocol : public Object
{
public:
  // returns a TypeId describing the class's Attributes and TraceSources.
  static TypeId GetTypeId();

  // do something interesting with the packet
  void ReceivePacket(Ptr<Packet> packet);

  // work finished after some time
  void FinishWork(Ptr<Packet> packet, Time workStarted);

  // signature of a callback for successful and failed work
  typedef Callback< void, Ptr<const Packet> > WorkSuccess;
  typedef Callback< void, Ptr<const Packet>, int > WorkFailed;

  // set an external function to be called on successful or failed work
  void SetWorkSuccessCallback(WorkSuccess callback);
  void SetWorkFailedCallback(WorkFailed callback);

private:
  // actual callback variables, set using public functions
  WorkSuccess m_workSuccessCallback;
  WorkFailed m_workFailedCallback;

  // parameters of the interesting work on the packet
  int m_param1;
  double m_param2;
  Time m_paramTime;

  // trace callback for start of work with initial packet
  TracedCallback< Time, Ptr<const Packet> > m_workStartTrace;

  // trace callback for time and duration of work done with processed packet
  TracedCallback< Time, Time, Ptr<const Packet> > m_workEndTrace;
};

NS_OBJECT_ENSURE_REGISTERED(MyProtocol);

TypeId
MyProtocol::GetTypeId()
{
  static TypeId tid = TypeId("ns3::MyProtocol")
    .SetParent<Object>()
    .AddConstructor<MyProtocol>()
    .AddAttribute("Param1",
                  "Important parameter of work done by this protocol.",
                  IntegerValue(502),
                  MakeIntegerAccessor(&MyProtocol::m_param1),
                  MakeIntegerChecker<int>())
    .AddAttribute("Param2",
                  "Another important parameter of work done by this protocol.",
                  DoubleValue(999.0),
                  MakeDoubleAccessor(&MyProtocol::m_param2),
                  MakeDoubleChecker<double>(100, 10000))
    .AddAttribute("ParamTime",
                  "Actual parameter specifying work time.",
                  TimeValue( MilliSeconds(10) ),
                  MakeTimeAccessor(&MyProtocol::m_paramTime),
                  MakeTimeChecker())
    .AddTraceSource("WorkStart",
                    "Time packet work started.",
                    MakeTraceSourceAccessor(&MyProtocol::m_workStartTrace))
    .AddTraceSource("WorkEnd",
                    "Triggered on work end.",
                    MakeTraceSourceAccessor(&MyProtocol::m_workEndTrace))
    ;
  return tid;
}

void
MyProtocol::SetWorkSuccessCallback(WorkSuccess callback)
{
  m_workSuccessCallback = callback;
}
void
MyProtocol::SetWorkFailedCallback(WorkFailed callback)
{
  m_workFailedCallback = callback;
}

void
MyProtocol::ReceivePacket(Ptr<Packet> packet)
{
  std::cerr << "MyProtocol::ReceivePacket() with "
            << "Param1=" << m_param1 << ", Param2=" << m_param2 << "\n";

  m_workStartTrace(Simulator::Now(), packet);

  Simulator::Schedule(Simulator::Now() + m_paramTime,
                      &MyProtocol::FinishWork, this,
                      packet, Simulator::Now());
}

void
MyProtocol::FinishWork(Ptr<Packet> packet, Time workStarted)
{
  // do something interesting with packet.
  bool workOk = (m_param1 % 2 == 1);

  if (workOk)
    {
      m_workSuccessCallback(packet);
    }
  else
    {
      m_workFailedCallback(packet, 10);
    }

  m_workEndTrace(Simulator::Now(), Simulator::Now() - workStarted,
                 packet);
}

/* main program */

void
Proto1WorkSuccessCallback(Ptr<const Packet> packet)
```

```
{
  std::cerr << "proto1's work succeeded on packet.\n";
}
void
Proto1WorkFailedCallback(Ptr<const Packet> packet, int reason)
{
  std::cerr << "proto1's work failed on packet, reason: " << reason << ".\n";
}

void
Proto1WorkStartTrace(std::string context,
                     Time start, Ptr<const Packet> packet)
{
  std::cerr << Simulator::Now() << " " << context
            << " time=" << start << ".\n";
}
void
Proto1WorkEndTrace(std::string context,
                   Time start, Time duration, Ptr<const Packet> packet)
{
  std::cerr << Simulator::Now() << " " << context
            << " time=" << start
            << " duration=" << duration << ".\n";
}

int main(int argc, char *argv[])
{
  Config::SetDefault("ns3::MyProtocol::Param1", IntegerValue(503));

  CommandLine cmd;
  cmd.Parse(argc, argv);

  Ptr<MyProtocol> proto1
    = CreateObject<MyProtocol>("Param2", DoubleValue(1001.0),
                               "ParamTime", TimeValue(Seconds(0.240)));

  proto1->SetAttribute("Param2", StringValue("1002.5"));

  proto1->SetWorkSuccessCallback(MakeCallback(&Proto1WorkSuccessCallback));
  proto1->SetWorkFailedCallback(MakeCallback(&Proto1WorkFailedCallback));

  proto1->TraceConnect("WorkStart", "main::proto1",
                       MakeCallback(&Proto1WorkStartTrace));
  proto1->TraceConnect("WorkEnd", "main::proto1",
                       MakeCallback(&Proto1WorkEndTrace));

  Simulator::Schedule(Seconds(1),
                      &MyProtocol::ReceivePacket, proto1,
                      Create<Packet>(100));

  Simulator::Run();
}
```

1.3 highway-example

```
// -*- mode: c++; c-file-style: "gnu"; indent-tabs-mode: nil; -*-

/*
 * Test case: 6*n nodes on a six lane highway
 *
 * 6*n nodes are put on a highway with 6 lanes. Each lane is 5 meters apart
 * from neighboring lanes. Cars are spaced at 90 meters on each lane (15 meters
 * between two nodes along the x axis) yielding a total of 66.6 nodes per kilometer.
 */

#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/wifi-module.h"
#include "ns3/helper-module.h"
#include "ns3/traffic-application.h"

#include <iostream>
#include <iomanip>
#include <numeric>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("Main");

class Experiment
{
public:

  static const double m_simulatedTime = 60.0;

  unsigned int m_appTxPackets;
  unsigned int m_appRxPackets;
  unsigned int m_phyRxErrors;

  void
  Run(unsigned int numNodes)
  {
    // Create nodes and store them in the container.

    NodeContainer nodes;
    nodes.Create(numNodes);

    // Add packet socket handlers.

    PacketSocketHelper packetSocket;
    packetSocket.Install(nodes);

    // Install wifi devices on the nodes.

    Ns2ExtWifiChannelHelper wifiChannel;
    wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
    wifiChannel.AddPropagationLoss("ns3::ThreeLogDistancePropagationLossModel");
    wifiChannel.AddPropagationLoss("ns3::NakagamiPropagationLossModel",
                                   "m0", DoubleValue(1.5),
                                   "m1", DoubleValue(1.0),
                                   "m2", DoubleValue(1.0));

    Ns2ExtWifiPhyHelper wifiPhy = Ns2ExtWifiPhyHelper::Default();
    wifiPhy.SetChannel(wifiChannel.Create());
    wifiPhy.Set("UseConstantNoiseFloor", BooleanValue(true));
    wifiPhy.Set("ConstantNoiseFloor", DoubleValue(-99.0));
    wifiPhy.Set("PreambleCapture", BooleanValue(true));
    wifiPhy.Set("DataCapture", BooleanValue(true));

    WifiHelper wifi = WifiHelper::Default();
    wifi.SetMac("ns3::AdhocWifiMac");
    wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
                                 "DataMode", StringValue("wifia-6mbs"),
                                 "NonUnicastMode", StringValue("wifia-6mbs"));

    wifi.Install(wifiPhy, nodes);

    // Position nodes on to highway lanes.

    Ptr<ListPositionAllocator> positionAlloc
      = CreateObject<ListPositionAllocator>();
    for (unsigned int i = 0; i < numNodes; ++i)
      {
        positionAlloc->Add(Vector(i * 15, (i % 6) * 5, 0.0));
      }

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

    // Use broadcast packet address for applications.

    PacketSocketAddress socketBroadcast;
    socketBroadcast.SetAllDevices();
    socketBroadcast.SetPhysicalAddress(Mac48Address::GetBroadcast());
    socketBroadcast.SetProtocol(1);

    // Install TrafficApplication on each node.

    Ptr<SimpleTrafficPacketFactory> packetFactory
      = CreateObject<SimpleTrafficPacketFactory>("Size", UintegerValue(400));

    TrafficHelper trafficApp("ns3::PacketSocketFactory", socketBroadcast);
    trafficApp.SetAttribute("PacketFactory", PointerValue(packetFactory) );
    trafficApp.SetAttribute("OnTime",
                            RandomVariableValue( ConstantVariable(m_simulatedTime) ));
    trafficApp.SetAttribute("OffTime",
                            RandomVariableValue( UniformVariable(0.0, 0.1) ));
    trafficApp.SetAttribute("Interval",
                            RandomVariableValue( ConstantVariable(0.1) ));

    ApplicationContainer app = trafficApp.Install(nodes);
    app.Start( Seconds(0.0) );
    app.Stop( Seconds(m_simulatedTime) );

    // Add Trace callbacks to gather statistics.

    Config::Connect("/NodeList/*/ApplicationList/*/$ns3::TrafficApplication/Tx",
                    MakeCallback(&Experiment::AppTxTrace, this));
    Config::Connect("/NodeList/*/ApplicationList/*/$ns3::TrafficApplication/Rx",
                    MakeCallback(&Experiment::AppRxTrace, this));

    Config::Connect("/NodeList/*/DeviceList/*/Phy/State/RxError",
                    MakeCallback(&Experiment::PhyRxErrorTrace, this));

    // Zero counters and run simulation.

    m_appTxPackets = 0;
    m_appRxPackets = 0;
    m_phyRxErrors = 0;

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

  void
  AppTxTrace(std::string context, Ptr<const Packet> p)
  {
    NS_LOG_DEBUG(context << " TX size=" << p->GetSize());
    ++m_appTxPackets;
  }

  void
  AppRxTrace(std::string context, Ptr<const Packet> p, const Address& from)
  {
    NS_LOG_DEBUG(context << " RX from=" << from << " size=" << p->GetSize());
    ++m_appRxPackets;
  }

  void
  PhyRxErrorTrace(std::string context, Ptr<const Packet> p,
                  Ptr<const WifiPhyTag> phytag, WifiPhy::RxErrorReason reason)
  {
    NS_LOG_DEBUG(context << " PHYRXERROR"
                 << " reason=" << WifiPhy::RxErrorReasonToString(reason)
                 << " phytag={" << *phytag << "} p={" << *p << "}");
    ++m_phyRxErrors;
  }
};

template <typename Container>
double meanValue(const Container& c)
{
  return std::accumulate(c.begin(), c.end(), 0.0) / c.size();
}

template <typename Container>
double standardDeviation(const Container& c)
{
  double squareSum = 0.0;
  double sum = 0.0;

  for (typename Container::const_iterator ei = c.begin();
       ei != c.end(); ++ei)
    {
      squareSum += (double)(*ei) * (double)(*ei);
      sum += *ei;
    }

  double mean = sum / c.size();
  return sqrt( (squareSum / c.size()) - (mean * mean) );
}

template <typename Container>
double errorMargin(const Container& c)
{
  return 2.576 * standardDeviation(c) / sqrt(c.size());
}

int main(int argc, char *argv[])
{
  CommandLine cmd;
  int replications = 1;
  unsigned int fixedNumNodes = 0;
  cmd.AddValue("Replications", "Perform independent replications.", replications);
  cmd.AddValue("NumNodes", "Run for a fixed number of node.", fixedNumNodes);
  cmd.Parse(argc, argv);

  for (unsigned int numNodes = 6; numNodes <= 180; numNodes += 6)
    {
      if (fixedNumNodes != 0 && numNodes != fixedNumNodes) continue;

      std::vector<unsigned int> appTxPackets;
      std::vector<unsigned int> appRxPackets;
      std::vector<unsigned int> phyRxErrors;

      for(int rep = 0; rep < replications; ++rep)
        {
          SeedManager::SetRun(rep);

          Experiment experiment;
          experiment.Run(numNodes);

          appTxPackets.push_back( experiment.m_appTxPackets );
          appRxPackets.push_back( experiment.m_appRxPackets );
          phyRxErrors.push_back( experiment.m_phyRxErrors );
        }

      std::cout << std::fixed
                << numNodes
                << " " << meanValue(appTxPackets) << " " << errorMargin(appTxPackets)
                << " " << meanValue(appRxPackets) << " " << errorMargin(appRxPackets)
                << " " << meanValue(phyRxErrors) << " " << errorMargin(phyRxErrors)
                << std::endl;
    }

  return 0;
}
```