ns-3 Experiment Guide

Release ns-3-dev

ns-3 project

March 03, 2013

CONTENTS

1 Introduction 3

2 Experiment lifecycle and ns-3 5

3 Data Collection Module 7
- 3.1 Model Description . . . 7

4 Probe 9
- 4.1 Overview . . . 9
- 4.2 Creation . . . 9
- 4.3 Importing and exporting data . . . 10
- 4.4 DoubleProbe overview . . . 10
- 4.5 Other probes . . . 11
- 4.6 Creating new Probe types . . . 12
- 4.7 Examples . . . 12
- 4.8 References . . . 15

5 Collector 17
- 5.1 Overview . . . 17
- 5.2 Creation . . . 18
- 5.3 Importing and exporting data . . . 19
- 5.4 BasicStatsCollector overview . . . 19
- 5.5 Statistical Quantities Calculated on the Fly . . . 19
- 5.6 Examples . . . 20
- 5.7 Tests . . . 20

6 Aggregator 23
- 6.1 Overview . . . 23
- 6.2 GnuplotAggregator Example . . . 23
- 6.3 GnuplotHelper Example . . . 24

7 Configuration management 27

8 Single run experiment management 29

9 Experiment Execution Manager for MRIP 31

Bibliography 33

This is the *ns-3 Experiment Guide*. Primary documentation for the ns-3 project is available in five forms:

- ns-3 Doxygen: Documentation of the public APIs of the simulator
- Tutorial , Manual, and Model Library for the latest release and development tree
- ns-3 wiki

This document is written in reStructuredText for Sphinx and is maintained in the `doc/experiment` directory of ns-3's source code.

CHAPTER

ONE

INTRODUCTION

This guide documents *ns-3* simulation modules and best practices to conduct sound networking simulations using *ns-3*. While the core of *ns-3* enables users to write simulation programs that can be compiled and run, it does not offer guidance on how to build experiments for *ns-3* using established experimental methodology. This document aims to address this gap.

This guide first looks at the **experiment lifecycle** in the context of *ns-3*. Two primary use cases are described and later supported:

1. Writing a standalone *ns-3* program that embodies an experiment within a single run.
2. Using an external framework to create experiments that execute, possibly many times with different parameters and factors, one or more *ns-3* programs, and that collect, process, and archive the inputs and outputs of the experiment.

Data collection is a primary experimental concern, and the next section documents a module used for data collection within *ns-3*.

Output analysis of data is necessary not only for processing good data, but for deciding when simulation transients and termination conditions have been reached. The next section describes some support for such output analysis.

Configuration management and the capture of enough data to make *ns-3* experiments fully reproducible is the subject of the next section.

Two modes for running experiments, both **single run experiment management** and an external **Experiment Execution Manager** for multiple replications, are next described.

Finally, **web-based automation** tools to construct and manage *ns-3* experiments are documented.

CHAPTER

TWO

EXPERIMENT LIFECYCLE AND NS-3

CHAPTER

THREE

DATA COLLECTION MODULE

This chapter describes the ns-3 Data Collection Framework (DCF, not to be confused with the *distributed coordination function* in the IEEE 802.11 standard). The framework provides functionality to monitor simulation data, to perform on-line reduction, and to marshal data into various output formats.

The framework supports two kinds of ns-3 simulations: standalone ns-3 runs that don't rely on any external programs and, in the future, also ns-3 runs that execute within the context of the Simulation Automation Framework for Experiments (SAFE).

3.1 Model Description

The source code for the classes lives in the directory `src/data-collection`.

3.1.1 Design

The DCF consists of three basic classes:

- *Probe* is a mechanism to instrument and control the output of simulation data that is used to monitor interesting events. It produces output in the form of one or more *ns-3* trace sources. Probe objects are hooked up to one or more trace *sinks* (called *Collectors*), which process samples on-line and prepare them for output.
- *Collector* consumes the data generated by one or more Probe objects. It performs transformations on the data, such as normalization, reduction, and the computation of basic statistics. Collector objects do not produce data that is directly output by the ns-3 run; instead, they output data downstream to another type of object, called *Aggregator*, which performs that function. Typically, Collectors output their data in the form of trace sources as well, allowing collectors to be chained in series.
- *Aggregator* is the end point of the data collected by a network of Probes and Collectors. The main responsibility of the Aggregator is to marshal data and their corresponding metadata, into different output formats such as plain text files, spreadsheet files, or databases.

All three of these classes provide the capability to dynamically turn themselves on or off throughout a simulation.

Any standalone *ns-3* simulation run that uses the DCF will typically create at least one instance of each of the three classes above.

3.1.2 References

CHAPTER
FOUR

PROBE

This section details the functionalities provided by the Probe class to an *ns-3* simulation, and gives examples on how to code them in a program. This section is meant for users interested in developing simulations with the *ns-3* tools and using the Data Collection Framework, of which the Probe class is a part, to generate artifacts with their simulation's results.

4.1 Overview

A Probe object is supposed to be connected to a variable from the simulation whose values throughout the experiment are relevant to the user. The Probe will record what were values assumed by the variable throughout the simulation and pass such data to another member of the Data Collection Framework. While it is out of this section's scope to discuss what happens after the Probe produces its output, it is sufficient to say that, by the end of the simulation, the user will have detailed information about what values were stored inside the variable being probed during the simulation.

Typically, a Probe is connected to an *ns-3* trace source. In this manner, whenever the trace source exports a new value, the Probe consumes the value (and exports it downstream to another object via its own trace source).

The Probe can be thought of as kind of a filter on trace sources. The main reasons for possibly hooking to a Probe rather than directly to a trace source are as follows:

- Probes may be dynamically turned on and off during the simulation with calls to `Enable()` and `Disable()`. For example, the outputting of data may be turned off during the simulation warmup phase.
- Probes may perform operations on the data to extract values from more complicated structures; for instance, outputting the packet size value from a received ns3::Packet.
- Probes register a name in the ns3::Config namespace (using `Names::Add ()`) so that other objects may refer to them.
- Probes provide a static method that allows one to manipulate a Probe by name, such as what is done in ns2measure **[1]_**

  ```
  Stat::put ("my_metric", ID, sample);
  ```

4.2 Creation

Note that a Probe base class object can not be created because it is an abstract base class, i.e. it has pure virtual functions that have not been implemented. An object of type DoubleProbe, which is a subclass of the Probe class, will be created here to show what needs to be done.

One declares a DoubleProbe in dynamic memory by using the smart pointer class (Ptr<T>). To create a DoubleProbe in dynamic memory with smart pointers, one just needs to call the ns-3 method CreateObject:

```
Ptr<DoubleProbe> myprobe = CreateObject<DoubleProbe> ();
```

The declaration above creates DoubleProbes using the default values for its attributes. There are four attributes in the DoubleProbe class; two in the base class object DcfObject, and two in the Probe base class:

- "Name" (DcfObject), a StringValue
- "Enabled" (DcfObject), a BooleanValue
- "Start" (Probe), a TimeValue
- "Stop" (Probe), a TimeValue

One can set such attributes at object creation by using the following method:

```
Ptr<DoubleProbe> myprobe = CreateObjectWithAttributes<DoubleProbe> (
    "Name", StringValue ("myprobe"),
    "Enabled", BooleanValue (false),
    "Start", TimeValue (Seconds (100.0)),
    "Stop", TimeValue (Seconds (1000.0)));
```

Start and Stop are Time variables which determine the interval of action of the Probe. The Probe will only output data if the current time of the Simulation is inside of that interval. The special time value of 0 seconds for Stop will disable this attribute (i.e. keep the Probe on for the whole simulation). Enabled is a flag that turns the Probe on or off, and must be set to true for the Probe to export data. The Name is the object's name in the DCF framework.

4.3 Importing and exporting data

ns-3 trace sources are strongly typed, so the mechanisms for hooking Probes to a trace source and for exporting data belong to its subclasses. For instance, the default distribution of *ns-3* provides a class DoubleProbe that is designed to hook to a trace source exporting a double value. We'll next detail the operation of the DoubleProbe, and then discuss how other Probe classes may be defined by the user.

4.4 DoubleProbe overview

The DoubleProbe connects to a double-valued *ns-3* trace source, and itself exports a different double-valued *ns-3* trace source.

The following code, drawn from `src/data-collection/examples/double-probe-example.cc`, shows the basic operations of plumbing the DoubleProbe into a simulation, where it is probing a Counter exported by an emitter object (class Emitter).

```
Ptr<Emitter> emitter = CreateObject<Emitter> ();
Names::Add ("/Names/Emitter", emitter);
...

Ptr<DoubleProbe> probe1 = CreateObject<DoubleProbe> ();

// Connect the probe to the emitter's Counter
bool connected = probe1->ConnectByObject ("Counter", emitter);
```

The following code is probing the same Counter exported by the same emitter object. This DoubleProbe, however, is using a path in the configuration namespace to make the connection. Note that the emitter registered itself in the configuration namespace after it was created; otherwise, the ConnectByPath would not work.

```
Ptr<DoubleProbe> probe2 = CreateObject<DoubleProbe> ();

// Note, no return value is checked here
probe2->ConnectByPath ("/Names/Emitter/Counter");
```

The next DoubleProbe shown that is shown below will have its value set using its path in the configuration namespace. Note that this time the DoubleProbe registered itself in the configuration namespace after it was created.

```
Ptr<DoubleProbe> probe3 = CreateObject<DoubleProbe> ();
probe3->SetName ("StaticallyAccessedProbe");

// We must add it to the config database
Names::Add ("/Names/Probes", probe3->GetName (), probe3);
```

The emitter's Count() function is now able to set the value for this DoubleProbe as follows:

```
void
Emitter::Count (void)
{
  ...
  m_counter += 1.0;
  DoubleProbe::SetValueByPath ("/Names/StaticallyAccessedProbe", m_counter);
  ...
}
```

The above example shows how the code calling the Probe does not have to have an explicit reference to the Probe, but can direct the value setting through the Config namespace. This is similar in functionality to the *Stat::Put* method introduced by ns2measure paper [Cic06], and allows users to temporarily insert Probe statements like *printf* statements within existing ns-3 models. Note that in order to be able to use the DoubleProbe in this example like this, 2 things were necessary:

1. the data-collection module header file was included in the example .cc file
2. the example was made dependent on the data-collection module in its wscript file.

Analogous things need to be done in order to add other Probes in other places in the *ns-3* code base.

The values for the DoubleProbe can also be set using the function DoubleProbe::SetValue(), while the values for the DoubleProbe can be gotten using the function DoubleProbe::GetValue().

The DoubleProbe exports double values in its "Output" trace source; a downstream object can hook a trace sink (NotifyViaProbe) to this as follows:

```
connected = probe1->TraceConnect ("Output", probe1->GetName (), MakeCallback (&NotifyViaProbe));
```

4.5 Other probes

Besides the DoubleProbe, the following Probes are also available:

- Uinteger8Probe connects to an *ns-3* trace source exporting an uint8_t.
- Uinteger16Probe connects to an *ns-3* trace source exporting an uint16_t.
- Uinteger32Probe connects to an *ns-3* trace source exporting an uint32_t.
- PacketProbe connects to an *ns-3* trace source exporting a packet.
- Ipv4PacketProbe connects to an *ns-3* trace source exporting a packet, an IPv4 object, and an interface.

4.6 Creating new Probe types

To create a new Probe type, you need to perform the following steps:

- Be sure that your new Probe class is derived from the Probe base class.
- Be sure that the pure virtual functions that your new Probe class inherits from the Probe base class are implemented.
- Find an existing Probe class that uses a trace source that is closest in type to the type of trace source your Probe will be using.
- Copy that existing Probe class's header file (.h) and implementation file (.cc) to two new files with names matching your new Probe.
- Replace the types, arguments, and variables in the copied files with the appropriate type for your Probe.
- Make necessary modifications to make the code compile and to make it behave as you would like.

4.7 Examples

Two examples will be discussed in detail here:

- Double Probe Example
- IPv4 Packet Plot Example

4.7.1 Double Probe Example

The double probe example has been discussed previously. The example program can be found in `src/data-collection/examples/double-probe-example.cc`. To summarize what occurs in this program, there is an emitter that exports a counter that increments according to a Poisson process. In particular, two ways of emitting data are shown:

1. through a traced variable hooked to one Probe:

```
TracedValue<double> m_counter;  // normally this would be integer type
```

2. through a counter whose value is posted to a second Probe, referenced by its name in the Config system:

```
void
Emitter::Count (void)
{
  NS_LOG_FUNCTION (this);
  NS_LOG_DEBUG ("Counting at " << Simulator::Now ().GetSeconds ());
  m_counter += 1.0;
  DoubleProbe::SetValueByPath ("/Names/StaticallyAccessedProbe", m_counter);
  Simulator::Schedule (Seconds (m_var->GetValue ()), &Emitter::Count, this);
}
```

Let's look at the Probe more carefully. Probes can receive their values in a multiple ways:

1. by the Probe accessing the trace source directly and connecting a trace sink to it
2. by the Probe accessing the trace source through the config namespace and connecting a trace sink to it
3. by the calling code explicitly calling the Probe's *SetValue()* method
4. by the calling code explicitly calling *SetValueByPath ("/path/through/Config/namespace", ...)*

The first two techniques are expected to be the most common. Also in the example, the hooking of a normal callback function is shown, as is typically done in *ns-3*. This callback function is not associated with a Probe object. We'll call this case 0) below.

```
// This is a function to test hooking a raw function to the trace source
void
NotifyViaTraceSource (std::string context, double oldVal, double newVal)
{
  NS_LOG_DEBUG ("context: " << context << " old " << oldVal << " new " << newVal);
}
```

First, the emitter needs to be setup:

```
Ptr<Emitter> emitter = CreateObject<Emitter> ();
Names::Add ("/Names/Emitter", emitter);

// The Emitter object is not associated with an ns-3 node, so
// it won't get started automatically, so we need to do this ourselves
Simulator::Schedule (Seconds (0.0), &Emitter::Start, emitter);
```

The various DoubleProbes interact with the emitter in the example as shown below.

Case 0):

```
// The below shows typical functionality without a probe
// (connect a sink function to a trace source)
//
connected = emitter->TraceConnect ("Counter", "sample context", MakeCallback (&NotifyViaTraceSource))
NS_ASSERT_MSG (connected, "Trace source not connected");
```

case 1):

```
//
// Probe1 will be hooked directly to the Emitter trace source object
//

// probe1 will be hooked to the Emitter trace source
Ptr<DoubleProbe> probe1 = CreateObject<DoubleProbe> ();
// the probe's name can serve as its context in the tracing
probe1->SetName ("ObjectProbe");

// Connect the probe to the emitter's Counter
connected = probe1->ConnectByObject ("Counter", emitter);
NS_ASSERT_MSG (connected, "Trace source not connected to probe1");
```

case 2):

```
//
// Probe2 will be hooked to the Emitter trace source object by
// accessing it by path name in the Config database
//

// Create another similar probe; this will hook up via a Config path
Ptr<DoubleProbe> probe2 = CreateObject<DoubleProbe> ();
probe2->SetName ("PathProbe");

// Note, no return value is checked here
probe2->ConnectByPath ("/Names/Emitter/Counter");
```

case 4) (case 3 is not shown in this example):

```
//
// Probe3 will be called by the emitter directly through the
// static method SetValueByPath().
//
Ptr<DoubleProbe> probe3 = CreateObject<DoubleProbe> ();
probe3->SetName ("StaticallyAccessedProbe");
// We must add it to the config database
Names::Add ("/Names/Probes", probe3->GetName (), probe3);
```

And finally, the example shows how the probes can be hooked to generate output:

```
// The probe itself should generate output.  The context that we provide
// to this probe (in this case, the probe name) will help to disambiguate
// the source of the trace
connected = probe3->TraceConnect ("Output", "/Names/Probes/StaticallyAccessedProbe/Output", MakeCallb
NS_ASSERT_MSG (connected, "Trace source not .. connected to probe3 Output");
```

The following callback is hooked to the Probe in this example for illustrative purposes; normally, the Probe would be hooked to a Collector object.

```
// This is a function to test hooking it to the probe output
void
NotifyViaProbe (std::string context, double oldVal, double newVal)
{
  NS_LOG_DEBUG ("context: " << context << " old " << oldVal << " new " << newVal);
}
```

4.7.2 IPv4 Packet Plot Example

The IPv4 packet plot example is based on the fifth.cc example from the *ns-3* Tutorial. It can be found in `src/data-collection/examples/ipv4-packet-plot-example.cc`.

```
// ===========================================================================
//
//         node 0                 node 1
//   +----------------+    +----------------+
//   |    ns-3 TCP    |    |    ns-3 TCP    |
//   +----------------+    +----------------+
//   |    10.1.1.1    |    |    10.1.1.2    |
//   +----------------+    +----------------+
//   | point-to-point |    | point-to-point |
//   +----------------+    +----------------+
//           |                     |
//           +---------------------+
```

We'll just look at the Probe, as it illustrates that Probes may also unpack values from structures (in this case, packets) and report those values as trace source outputs, rather than just passing through the same type of data.

There are other aspects of this example that will be explained later in the documentation. The two types of data that are exported are the packet itself (*Output*) and a count of the number of bytes in the packet (*OutputBytes*).

```
TypeId
Ipv4PacketProbe::GetTypeId ()
{
  static TypeId tid = TypeId ("ns3::Ipv4PacketProbe")
    .SetParent<Probe> ()
    .AddConstructor<Ipv4PacketProbe> ()
    .AddTraceSource ( "Output",
                      "The packet plus its IPv4 object and interface that serve as the output for th
                      MakeTraceSourceAccessor (&Ipv4PacketProbe::m_output))
    .AddTraceSource ( "OutputBytes",
                      "The number of bytes in the packet",
                      MakeTraceSourceAccessor (&Ipv4PacketProbe::m_outputBytes))
  ;
  return tid;
}
```

When the Probe's trace sink gets a packet, if the Probe is enabled, then it will output the packet on its *Output* trace source, but it will also output the number of bytes on the *OutputBytes* trace source.

```
void
Ipv4PacketProbe::TraceSink (Ptr<const Packet> packet, Ptr<Ipv4> ipv4, uint32_t interface)
{
  NS_LOG_FUNCTION (this << packet << ipv4 << interface);
  if (IsEnabled ())
    {
      m_packet    = packet;
      m_ipv4      = ipv4;
      m_interface = interface;
      m_output (packet, ipv4, interface);

      uint32_t packetSizeNew = packet->GetSize ();
      m_outputBytes (m_packetSizeOld, packetSizeNew);
      m_packetSizeOld = packetSizeNew;
    }
}
```

4.8 References

CHAPTER

FIVE

COLLECTOR

This section details the functionalities provided by the Collector class to an *ns-3* simulation, and gives examples on how to code them in a program. This section is meant for users interested in developing simulations with the *ns-3* tools and using the Data Collection Framework, of which the Collector class is a part, to generate artifacts with their simulation's results.

5.1 Overview

A Collector object is supposed to be hooked to a trace source in order to receive input. A Collector accepts data from one or more data sources, performs transformation on that data, and reports the transformed data to a downstream object in the Data Collection Framework via its own trace sources. While it is out of this section's scope to discuss what happens after the Collector collects its values, it is sufficient to say that, by the end of the simulation, the user will have detailed information about the values being collected during the simulation.

Typically, a Collector is connected to a Probe. In this manner, whenever the Probe's trace source exports a new value, the Collector consumes the value (and exports it downstream to another object via its own trace sources at the appropriate time).

Periodic Collectors will report output according to a regular, configured time period. Asynchronous Collectors will report output according to a configured number (one or greater) of input samples. This number is called the batch size.

If the mode of the Collector is periodic, then it will report data at the end of every time period. If the mode of the Collector is asynchronous, then it will wait until it has enough new values to fill the current batch before it will report data.

Note the following about Collectors:

- Collectors may be dynamically turned on and off during the simulation with calls to `Enable()` and `Disable()`. For example, the collecting of data may be turned off during the simulation warmup phase, which means those values won't be included when computing the statistical measures.
- Collectors do not generate values like Probes do. Instead, they use the values generated by Probes for calculations of things like statistical quantities.
- Collectors receive data from Probes via callbacks. When a Probe is associated to a Collector, a call to TraceConnectWithoutContext is made to establish the Collector's trace sink method as a callback.
- Collectors send the data that they generate to Aggregators, or to other Collectors via trace sources.

5.2 Creation

Note that a Collector base class object can not be created because it is an abstract base class, i.e. it has pure virtual functions that have not been implemented. An object of type BasicStatsCollector, which is a subclass of the Collector class, will be created here to show what needs to be done.

One declares a BasicStatsCollector in dynamic memory by using the smart pointer class (Ptr<T>). To create a BasicStatsCollector in dynamic memory with smart pointers, one just needs to call the ns-3 method CreateObject:

```
Ptr<BasicStatsCollector> mycollector = CreateObject<BasicStatsCollector> ();
```

The declaration above creates BasicStatsCollectors using the default values for its attributes. There are six attributes in the BasicStatsCollector class; two in the base class object, DcfObject, and four in the Collector base class:

- "Name" (DcfObject), a StringValue
- "Enabled" (DcfObject), a BooleanValue
- "IsPeriodic" (Collector), a BooleanValue
- "ResetStatisticsEveryInterval" (Collector), a BooleanValue
- "Period" (Collector), a TimeValue
- "BatchSize" (Collector), a UintegerValue

One can set such attributes at object creation by using the following method:

```
Ptr<BasicStatsCollector> mycollector = CreateObjectWithAttributes<BasicStatsCollector> (
    "Name", StringValue ("mycollector"),
    "Enabled", BooleanValue (false),
    "IsPeriodic", BooleanValue (true),
    "ResetStatisticsEveryInterval", BooleanValue (true),
    "Period", TimeValue (Seconds (100.0)),
    "BatchSize", UintegerValue (10));
```

Enabled is a flag that turns the Collector on or off, and must be set to true for the Collector to collect data. The Name is the object's name in the DCF framework. Set IsPeriodic equal to true if you would like the Collector to report data at the end of every time period, and set it equal to false if you would like it to operate in asynchronous mode, i.e. wait until it has enough values to fill the current batch before it reports data. Set ResetStatisticsEveryInterval equal to true if you would like the Collector to reset its statistics every time period or batch and set it equal to false if you do not want them reset during the simulation. Period is a Time variable, which determines the time period between the output of values for periodic mode. BatchSize is the maximum number of samples allowed in a batch for asynchronous mode.

Note that in a real simulation you probably wouldn't set the IsPeriodic attribute equal to true and set the BatchSize attribute because the BatchSize attribute is ignored in periodic mode. Similarly, in a real simulation you probably wouldn't set the IsPeriodic attribute equal to false and set the Period attribute because the Period attribute is ignored in asynchronous mode.

The Collector base class also provides two more ways to specify its mode. You can specify that the Collector is periodic with a specified period as follows:

```
Ptr<BasicStatsCollector> mycollector = CreateObject<BasicStatsCollector> ();
mycollector->SetPeriodic (Seconds (100.0));
```

You can specify that the Collector is asynchronous with a specified batch size as follows:

```
Ptr<BasicStatsCollector> mycollector = CreateObject<BasicStatsCollector> ();
mycollector->SetAsynchronous (5.0);
```

5.3 Importing and exporting data

ns-3 trace sources are strongly typed, so the mechanisms for hooking Collectors to a trace source and for exporting data belong to its subclasses. For instance, the default distribution of *ns-3* provides a class BasicStatsCollector that is designed to hook to Probes or other Collectors exporting a double valued trace source. We'll next detail the operation of the BasicStatsCollector.

5.4 BasicStatsCollector overview

The BasicStatsCollector connects to a double-valued *ns-3* trace source, and itself exports different double-valued *ns-3* trace sources.

The following code, drawn from `src/data-collection/test/basic-stats-collector-test-suite.cc`, shows the basic operations of plumbing the BasicStatsCollector into a simulation, where it is collecting values from a Probe attached to an emitter object (class SampleEmitter2) exporting random values via its trace source.

```
m_emitter   = CreateObject<SampleEmitter2> ();

m_probe     = CreateObject<DoubleProbe> ();

m_collector = CreateObject<BasicStatsCollector> ();

...

m_probe->ConnectByObject ("Emitter", m_emitter);

m_probe->TraceConnectWithoutContext ("Output", MakeCallback(&BasicStatsCollector::TraceSink, m_colle
```

The BasicStatsCollector exports three pairs of double values in its trace sources:

Trace Source	Description
"SampleMean"	The current simulation time versus the mean of the values in the time period or batch
"SampleCount"	The current simulation time versus the number of the values in the time period or batch
"SampleSum"	The current simulation time versus the sum of the values in the time period or batch

Note that if the batch size is 1, which is the default, then the sample mean will just be equal to the value itself.

If the Collector is in periodic mode, then output after will occur after every time period has passed. If it is in asynchronous mode, then output will occur after every time the number of samples in the batch have reached the maximum batch size.

A downstream object can hook a trace sink (TraceSink) to any of these three as follows:

```
m_collector->TraceConnectWithoutContext ("SampleMean", MakeCallback (&BasicStatsCollectorAsynchronous

m_collector->TraceConnectWithoutContext ("SampleCount", MakeCallback (&BasicStatsCollectorAsynchronou

m_collector->TraceConnectWithoutContext ("SampleSum", MakeCallback (&BasicStatsCollectorAsynchronousI
```

5.5 Statistical Quantities Calculated on the Fly

BasicStatsCollectors perform running calcutions for various statistical metrics for the data values passed to them. At any given point in the simulation, you can retrieve the current value of each of these metrics. The following table shows the statistical metrics and the appropriate functions to use to get their value:

Statistical Metric	Function
Number of samples	`GetCount()`
Sum	`GetSum()`
Maximum	`GetMax()`
Minimum	`GetMin()`
Mean	`GetMean()`
Standard deviation	`GetStdDev()`
Variance	`GetVariance()`
Square total	`GetSqrSum()`

You can reset the values for all of the metrics by calling the function `Reset()`.

5.6 Examples

There are currently no examples that exercise the BasicStatsCollector.

5.7 Tests

One test will be discussed in detail here:

- Basic Stats Collector Test

5.7.1 Basic Stats Collector Test

The basic stats collector test has been discussed previously. The test suite can be found in `src/data-collection/test/basic-stats-collector-test-suite.cc`. To summarize what occurs in this test, there is an emitter that exports a sequence of random values as a trace source that is connected to a Probe, which produces values that are then collected by the BasicStatsCollector.

The test suite has 4 test cases that exercise both modes of the BasicStatsCollector (Periodic and Asynchronous) and having statistical values reset at the end of the interval or not for both modes:

1. Asynchronous Mode: Resets Statistics
2. Periodic Mode: Resets Statistics
3. Asynchronous Mode: DoesNot Reset Statistics
4. Periodic Mode: DoesNot Reset Statistics

Test case 1 will be discussed here.

The emitter, Probe, and BasicStatsCollector are all created in the constructor for the TestCase:

```
BasicStatsCollectorAsynchronousResetsStatisticsTestCase::BasicStatsCollectorAsynchronousResetsStatist
    : TestCase           ("Double Collector Asynchronous Resets Statistics Test"),
      m_collectorStart   (200),
      m_collectorStop    (300),
      m_batchSize        (5)
{
  // Create the emitter, probe, and collector.
  m_emitter   = CreateObject<SampleEmitter2> ();
  m_probe     = CreateObject<DoubleProbe> ();
  m_collector = CreateObject<BasicStatsCollector> ();
}
```

This function is used to set the mode for the BasicStatsCollector when it is called:

```
void
BasicStatsCollectorAsynchronousResetsStatisticsTestCase::EnableCollector ()
{
  // Enable the collector in asynchronous mode.
  m_collector->Enable ();
  m_collector->SetAsynchronous (m_batchSize);

  // Reset all of the emmiter's statistical variables.
  m_emitter->Reset ();
}
```

During the test, the Probe is connected to the emitter.

```
// Prepare the probe.
m_probe->SetAttribute ("Start", TimeValue (Seconds (100.0)));
m_probe->SetAttribute ("Stop", TimeValue (Seconds (400.0)));
Simulator::Stop (Seconds (500.0));
m_probe->ConnectByObject ("Emitter", m_emitter);
m_probe->TraceConnectWithoutContext ("Output", MakeCallback(&BasicStatsCollector::TraceSink, m_collec
```

During the test, the BasicStatsCollector is connected to the Probe.

```
// Prepare the collector, which is initially disabled to wait for
// output from the probe.
m_collector->Disable ();
m_collector->TraceConnectWithoutContext ("SampleMean", MakeCallback (&BasicStatsCollectorAsynchronous
```

This function, which is connected to the BasicStatsCollector's SampleMean trace source, checks the values that were calculated.

```
void
BasicStatsCollectorAsynchronousResetsStatisticsTestCase::TraceSink (double time, double value)
{
  // See if the number of values in the batch matches the expected batch size.
  NS_TEST_ASSERT_MSG_EQ (m_collector->GetCount (), m_batchSize, "Batch size is wrong");

  // Test the other statistics.
  NS_TEST_ASSERT_MSG_EQ (m_emitter->m_count, m_collector->GetCount (), "Count is wrong");

  ...

  // Reset all of the emmiter's statistical variables.
  m_emitter->Reset ();
}
```

CHAPTER

SIX

AGGREGATOR

This section details the functionalities provided by the Aggregator class to an *ns-3* simulation, and gives examples on how to code them in a program. This section is meant for users interested in developing simulations with the *ns-3* tools and using the Data Collection Framework, of which the Aggregator class is a part, to generate artifacts with their simulation's results.

6.1 Overview

An Aggregator object is supposed to be hooked to one or more trace sources in order to receive input. Aggregators are the end point of the data collected by the network of Probes and Collectors during the simulation. It is the Aggregator's job to take these values and transform them into their final output format such as plain text files, spreadsheet files, plots, or databases.

Typically, an aggregator is connected to one or more Collectors. In this manner, whenever the Collectors' trace sources export new values, the Aggregator can process the value so that it can be used in the final output format where the data values will reside after the simulation.

Note the following about Aggregators:

- Aggregators may be dynamically turned on and off during the simulation with calls to `Enable()` and `Disable()`. For example, the aggregating of data may be turned off during the simulation warmup phase, which means those values won't be included in the final output medium.
- Aggregators receive data from Collectors via callbacks. When a Collector is associated to an aggregator, a call to TraceConnect is made to establish the Aggregator's trace sink method as a callback.

6.2 GnuplotAggregator Example

To date, one aggregator, GnuplotAggregator, has been implemented.

An example that exercises this aggregator can be found in `src/data-collection/examples/gnuplot-aggregator-examp`

The code from the example shows how the GnuplotAggregator is used.

```
using namespace std;

string fileNameWithoutExtension = "gnuplot-aggregator";
string plotTitle                = "Gnuplot Aggregator Plot";
string plotXAxisHeading         = "Time (seconds)";
string plotYAxisHeading         = "Double Values";
string plotDatasetLabel         = "Data Values";
string datasetContext           = "Dataset/Context/String";

// Create an aggregator.
Ptr<GnuplotAggregator> aggregator =
  CreateObject<GnuplotAggregator> (fileNameWithoutExtension);

// Set the aggregator's properties.
aggregator->SetTerminal ("png");
aggregator->SetTitle (plotTitle);
aggregator->SetLegend (plotXAxisHeading, plotYAxisHeading);

// Add a data set to the aggregator.
aggregator->Add2dDataset (datasetContext, plotDatasetLabel);

// Enable logging of data for the aggregator.
aggregator->Enable ();

double time;
double value;

// Create the 2-D dataset.
for (time = -5.0; time <= +5.0; time += 1.0)
  {
    // Calculate the 2-D curve
    //
    //                   2
    //     value  =  time   .
    //
    value = time * time;

    // Add this point to the plot.
    aggregator->Write2d (datasetContext, time, value);
  }

// Disable logging of data for the aggregator.
aggregator->Disable ();
```

6.3 GnuplotHelper Example

To date, one aggregator helper, GnuplotHelper, has been implemented.

An example that exercises this aggregator helper can be found in `src/data-collection/examples/manet-safe.cc`.

The code from the example shows how the GnuplotHelper is used.

```
//===============================================================
//
// An application packet probe will be hooked to the PacketSink Rx
// trace source using a gnuplot helper.
//
//===============================================================

// Create the gnuplot helper.
GnuplotHelper packetPlotHelper;

// Configure the plot.
packetPlotHelper.ConfigurePlot ("manet-safe-packet-byte-count",
                                "MANET SAFE Packet Aggregator",
                                "Time (Seconds)",
                                "Count",
                                "png");

// Add a probe to the gnuplot helper.
packetPlotHelper.AddProbe ("ns3::ApplicationPacketProbe",
                           "PacketSinkRxProbe",
                           "/NodeList/*/ApplicationList/0/$ns3::PacketSink/Rx");

// Get a pointer to the helper's probe so that it can be configured.
Ptr<Probe> packetProbe = packetPlotHelper.GetProbe ("PacketSinkRxProbe");
packetProbe->SetAttribute ("Start", TimeValue (Seconds (120.0)));
packetProbe->SetAttribute ("Stop", TimeValue (Seconds (150.0)));

// Add a collector to the gnuplot helper.
packetPlotHelper.AddCollector ("ns3::BasicStatsCollector",
                               "PacketSinkRxCollector",
                               "PacketSinkRxProbe",
                               "OutputBytes");

// Get a pointer to the helper's collector so that it can be configured.
Ptr<Collector> packetCollector = packetPlotHelper.GetCollector ("PacketSinkRxCollector");
packetCollector->SetPeriodic (Seconds (0.5));

// Get a pointer to the helper's aggregator so that it can be configured.
Ptr<GnuplotAggregator> packetAggregator = packetPlotHelper.GetAggregator ();
packetAggregator->Set2dDatasetDefaultStyle (Gnuplot2dDataset::POINTS);

// Add some datasets to the plot.  Note that the dataset context
// strings, which are the third arguments in these function calls,
// must be unique
packetPlotHelper.Add2dDataset ("PacketSinkRxCollector",
                               "SampleCount",
                               "PacketSinkRxCollector/SampleCount",
                               "Packet Count");
packetPlotHelper.Add2dDataset ("PacketSinkRxCollector",
                               "SampleSum",
                               "PacketSinkRxCollector/SampleSum",
                               "Total Packet Byte Count");
packetPlotHelper.Add2dDataset ("PacketSinkRxCollector",
                               "SampleMean",
                               "PacketSinkRxCollector/SampleMean",
                               "Mean Packet Byte Count");

// Set this dataset's sytyle.
packetAggregator->Set2dDatasetStyle ("PacketSinkRxCollector/SampleCount",
                                     Gnuplot2dDataset::LINES_POINTS);
```

CHAPTER

SEVEN

CONFIGURATION MANAGEMENT

CHAPTER

EIGHT

SINGLE RUN EXPERIMENT MANAGEMENT

CHAPTER

NINE

EXPERIMENT EXECUTION MANAGER FOR MRIP

BIBLIOGRAPHY

[Cic06] Claudio Cicconetti, Enzo Mingozzi, Giovanni Stea, “An Integrated Framework for Enabling Effective Data Collection and Statistical Analysis with ns2, Workshop on ns-2 (WNS2), Pisa, Italy, October 2006.