CHAPTER

TEN

DATA COLLECTION

This chapter describes the ns-3 Data Collection Framework (DCF), which provides capabilities to obtain data generated by models in the simulator, to perform on-line reduction and data processing, and to marshal raw or transformed data into various output formats.

The framework presently supports standalone ns-3 runs that don't rely on any external program execution control. The objects provided by the DCF may be hooked to *ns-3* trace sources to enable data processing.

The source code for the classes lives in the directory `src/stats`.

This chapter is organized as follows. First, an overview of the architecture is presented. Next, the helpers for these classes are presented; this initial treatment should allow basic use of the data collection framework for many use cases. Users who wish to produce output outside of the scope of the current helpers, or who wish to create their own data collection objects, should read the remainder of the chapter, which goes into detail about all of the basic DCF object types and provides low-level coding examples.

10.1 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.

The overall flow of data processing is depicted in *Data Collection Framework overview*. On the left side, a running *ns-3* simulation is depicted. In the course of running the simulation, data is made available by models through trace sources, or via other means. The diagram depicts that probes can be connected to these trace sources to receive data asynchronously, or probes can poll for data. Data is then passed to a collector object that transforms the data. Finally, an aggregator can be connected to the outputs of the collector, to generate plots, files, or databases.

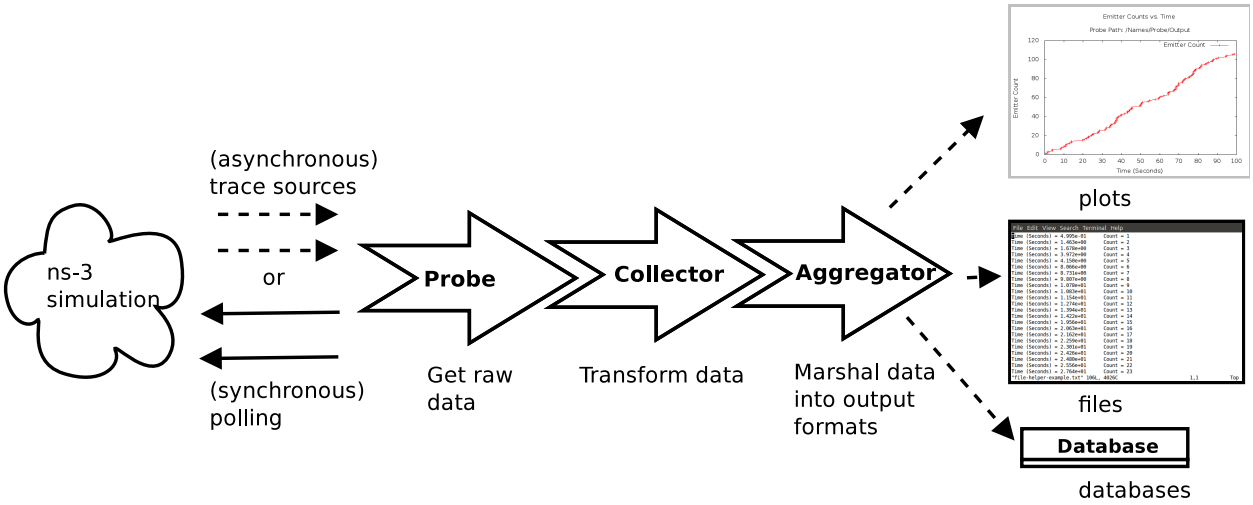

Figure 10.1: Data Collection Framework overview

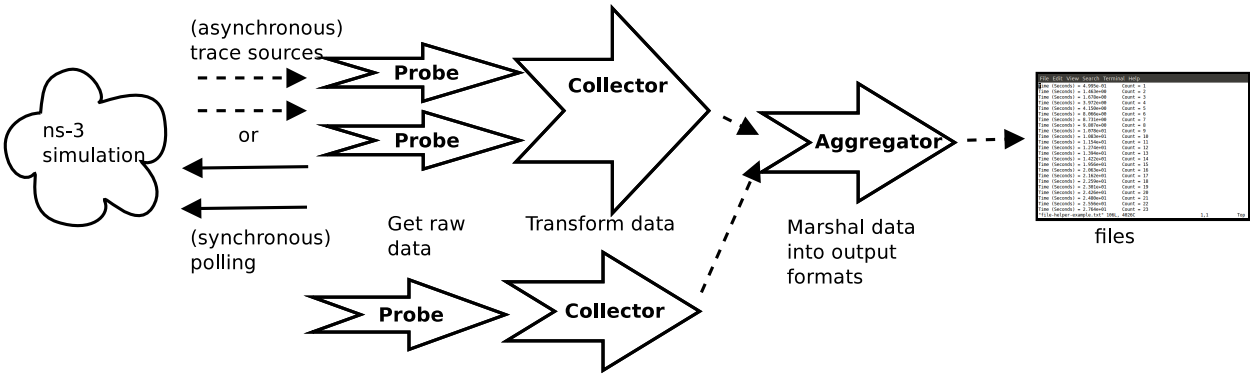

Figure 10.2: Data Collection Framework aggregation

A variation on the above figure is provided in *Data Collection Framework aggregation*. This second figure illustrates that the DCF objects may be chained together in a manner that downstream objects take inputs from multiple upstream objects. The figure conceptually shows that multiple probes may generate output that is fed into a single collector; as an example, a collector that outputs a ratio of two counters would typically acquire each counter data from separate probes. Multiple collectors can also feed into a single aggregator, which (as its name implies) may collect a number of data streams for inclusion into a single plot, file, or database.

10.2 Data Collection Helpers

The full flexibility of the data collection framework is provided by the interconnection of probes, collectors, and aggregators. Performing all of these interconnections leads to many configuration statements in user programs. For ease of use, some of the most common operations can be combined and encapsulated in helper functions. In addition, some statements involving *ns-3* trace sources do not have Python bindings, due to limitations in the bindings.

10.2.1 Data Collection Helpers Overview

In this section, we provide an overview of some helper classes that have been created to ease the configuration of the data collection framework for some common use cases. The helpers allow users to form common operations with only a few statements in their C++ or Python programs. But, this ease of use comes at the cost of significantly less flexibility than low-level configuration can provide, and the need to explicitly code support for new Probe types into the helpers (to work around an issue described below).

The emphasis on the current helpers is to marshal data out of *ns-3* trace sources into gnuplot plots or text files, without a high degree of output customization or statistical processing (initially). Also, the use is constrained to the available probe types in *ns-3*. Later sections of this documentation will go into more detail about creating new Probe types, as well as details about hooking together Probes, Collectors, and Aggregators in custom arrangements.

To date, two Data Collection helpers have been implemented:

- GnuplotHelper
- FileHelper

10.2.2 GnuplotHelper

The GnuplotHelper is a helper class for producing output files used to make gnuplots. The overall goal is to provide the ability for users to quickly make plots from data exported in *ns-3* trace sources. By default, a minimal amount of data transformation is performed; the objective is to generate plots with as few (default) configuration statements as possible.

GnuplotHelper Overview

The GnuplotHelper will create 3 different files at the end of the simulation:

- A space separated gnuplot data file
- A gnuplot control file
- A shell script to generate the gnuplot

There are two configuration statements that are needed to produce plots. The first statement configures the plot (filename, title, legends, and output type, where the output type defaults to PNG if unspecified):

```
void ConfigurePlot (const std::string &outputFileNameWithoutExtension,
                    const std::string &title,
                    const std::string &xLegend,
                    const std::string &yLegend,
                    const std::string &terminalType = ".png");
```

The second statement hooks the `Probe` of interest:

```
void PlotProbe (const std::string &typeId,
                const std::string &path,
                const std::string &probeTraceSource,
                const std::string &title);
```

The arguments are as follows:

- typeId: The *ns-3* TypeId of the Probe
- path: The path in the *ns-3* configuration namespace to one or more probes
- probeTraceSource: Which output of the probe should be connected to
- title: The title to associate with the dataset (in the gnuplot legend)

A variant on the PlotProbe above is to specify a fifth optional argument that controls where in the plot the key (legend) is placed.

A fully worked example (from `sixth-dcf.cc`) is shown below:

```
// Create the gnuplot helper.
GnuplotHelper plotHelper;

// Configure the plot.
plotHelper.ConfigurePlot ("sixth-dcf-packet-byte-count",
                          "Packet Byte Count vs. Time",
                          "Time (Seconds)",
                          "Packet Byte Count",
                          "png");

// Plot the values generated by the probe.
plotHelper.PlotProbe ("ns3::Ipv4PacketProbe",
                      "/NodeList/*/$ns3::Ipv4L3Protocol/Tx",
                      "OutputBytes",
                      "Packet Byte Count",
                      GnuplotAggregator::KEY_BELOW);
```

Note that the path specified may contain wildcards. In this case, multiple datasets are plotted on one plot; one for each matched path.

The main output produced will be three files:

```
sixth-dcf-packet-byte-count.dat
sixth-dcf-packet-byte-count.plt
sixth-dcf-packet-byte-count.sh
```

At this point, users can either hand edit the .plt file for further customizations, or just run it through gnuplot. Running *sh sixth-dcf-packet-byte-count.sh* simply runs the plot through gnuplot, as shown below.

It can be seen that the key elements (legend, title, legend placement, xlabel, ylabel, and path for the data) are all placed on the plot. Since there were two matches to the configuration path provided, the two data series are shown:

- Packet Byte Count-0 corresponds to /NodeList/0/$ns3::Ipv4L3Protocol/Tx
- Packet Byte Count-1 corresponds to /NodeList/1/$ns3::Ipv4L3Protocol/Tx

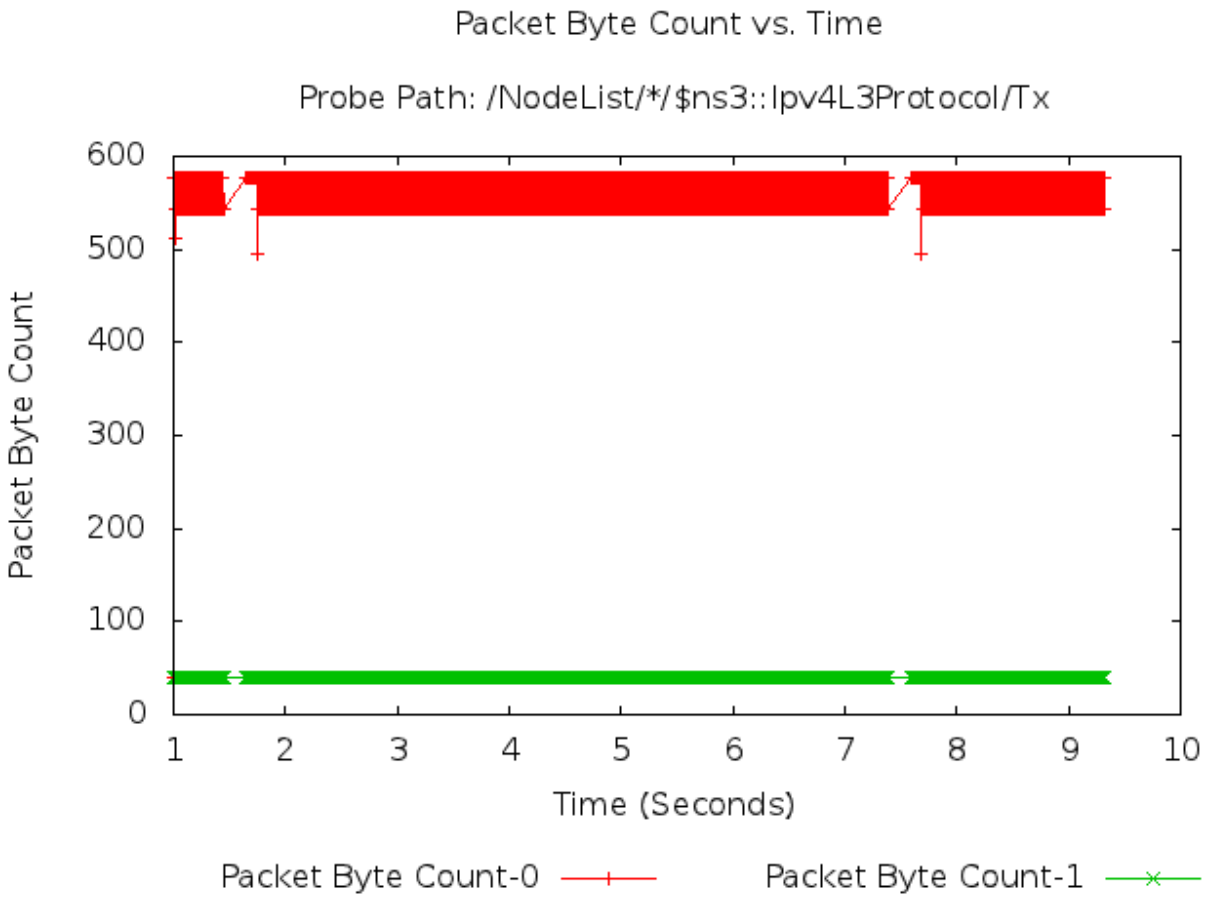

Figure 10.3: 2-D Gnuplot Created by sixth-dcf.cc Example.

GnuplotHelper ConfigurePlot

The GnuplotHelper's `ConfigurePlot()` function can be used to configure plots.

It has the following prototype:

```
void ConfigurePlot (const std::string &outputFileNameWithoutExtension,
                    const std::string &title,
                    const std::string &xLegend,
                    const std::string &yLegend,
                    const std::string &terminalType = ".png");
```

It has the following arguments:

Argument	Description
outputFileNameWithoutExtension	Name of gnuplot related files to write with no extension.
title	Plot title string to use for this plot.
xLegend	The legend for the x horizontal axis.
yLegend	The legend for the y vertical axis.
terminalType	Terminal type setting string for output. The default terminal type is "png".

The GnuplotHelper's `ConfigurePlot()` function configures plot related parameters for this gnuplot helper so that it will create a space separated gnuplot data file named outputFileNameWithoutExtension + ".dat", a gnuplot control file named outputFileNameWithoutExtension + ".plt", and a shell script to generate the gnuplot named outputFileNameWithoutExtension + ".sh".

An example of how to use this function can be seen in the `sixth-dcf.cc` code described above where it was used as follows:

```
plotHelper.ConfigurePlot ("sixth-dcf-packet-byte-count",
                          "Packet Byte Count vs. Time",
                          "Time (Seconds)",
                          "Packet Byte Count",
                          "png");
```

GnuplotHelper PlotProbe

The GnuplotHelper's `PlotProbe()` function can be used to plot values generated by probes.

It has the following prototype:

```
void PlotProbe (const std::string &typeId,
                const std::string &path,
                const std::string &probeTraceSource,
                const std::string &title,
                enum GnuplotAggregator::KeyLocation keyLocation = GnuplotAggregator::KEY_INSIDE);
```

It has the following arguments:

Argument	Description
typeId	The type ID for the probe used when it is created.
path	Config path to access the probe.
probeTraceSource	The probe trace source to access.
title	The title to be associated to this dataset
keyLocation	The location of the key in the plot. The default location is inside.

The GnuplotHelper's `PlotProbe()` function plots a dataset generated by hooking the *ns-3* trace source with a probe, and then plotting the values from the probeTraceSource. The dataset will have the provided title, and will consist of the 'newValue' at each timestamp.

If the config path has more than one match in the system because there is a wildcard, then one dataset for each match will be plotted. The dataset titles will be suffixed with the matched characters for each of the wildcards in the config path, separated by spaces. For example, if the proposed dataset title is the string "bytes", and there are two wildcards in the path, then dataset titles like "bytes-0 0" or "bytes-12 9" will be possible as labels for the datasets that are plotted.

An example of how to use this function can be seen in the `sixth-dcf.cc` code described above where it was used as follows:

```
plotHelper.PlotProbe ("ns3::Ipv4PacketProbe",
                      "/NodeList/*/$ns3::Ipv4L3Protocol/Tx",
                      "OutputBytes",
                      "Packet Byte Count",
                      GnuplotAggregator::KEY_BELOW);
```

Other Examples

Gnuplot Helper Example

A slightly simpler example than the `sixth-dcf.cc` example can be found in `src/stats/examples/gnuplot-helper-example.cc`. It is more of a toy example than `sixth-dcf.cc` because it has a made-up trace source created for demonstration purposes.

The following 2-D gnuplot was created using the example.

In this example, there is an Emitter object that increments its counter at various random times and then emits the counter's value as a trace source.

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

The following code is probing the Counter exported by the emitter object. This DoubleProbe 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> probe = CreateObject<DoubleProbe> ();
probe->SetName ("PathProbe");
Names::Add ("/Names/Probe", probe);

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

Note that because there are no wildcards in the path used below, only 1 datastream was drawn in the plot. This single datastream in the plot is simply labeled "Emitter Count", with no extra suffixes like you would see if there were wildcards in the path.

```
// Create the gnuplot helper.
GnuplotHelper plotHelper;

// Configure the plot.
plotHelper.ConfigurePlot ("gnuplot-helper-example",
                          "Emitter Counts vs. Time",
                          "Time (Seconds)",
                          "Emitter Count",
                          "png");

// Plot the values generated by the probe.  The path that we provide
// helps to disambiguate the source of the trace.
plotHelper.PlotProbe ("ns3::DoubleProbe",
```

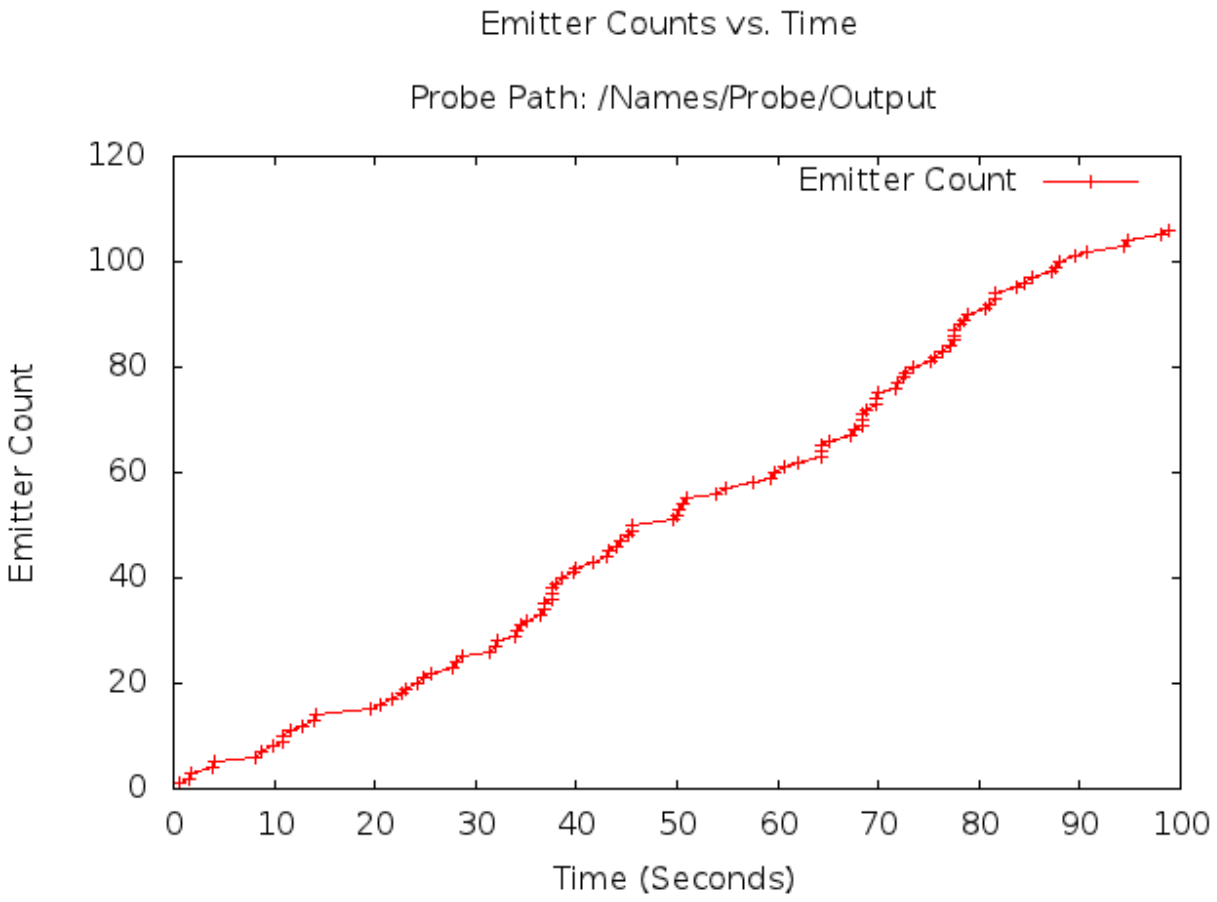

Figure 10.4: 2-D Gnuplot Created by gnuplot-helper-example.cc Example.

```
"/Names/Probe/Output",
"Output",
"Emitter Count",
GnuplotAggregator::KEY_INSIDE);
```

MANET SAFE Example

Another example that exercises the GnuplotHelper can be found in `src/stats/examples/manet-safe.cc`. This example is even better than the `sixth-dcf.cc` example in terms being a realistic simulation performed using *ns-3*. It is, therefore, more complicated and is doing more things with the data collection objects.

The following 2-D gnuplot was created using the example.

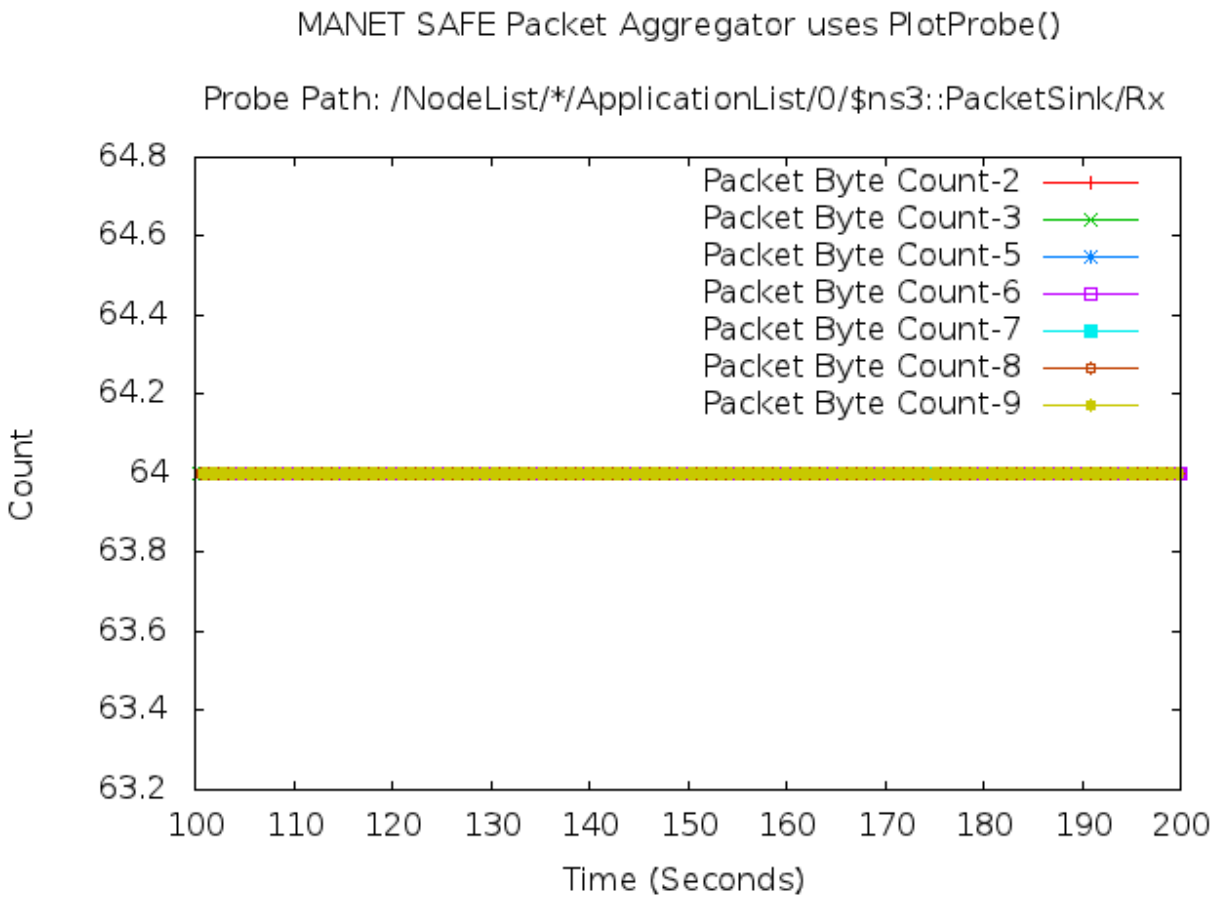

Figure 10.5: 2-D Gnuplot Created by manet-safe.cc Example.

The code that produced the gnuplot is below and shows how the GnuplotHelper is used to connect Probes directly to a GnuplotAggregator.

Note that because there were 10 matches for the wildcard in the path, 10 datastreams should have been drawn in the plot. But, 3 of the 10 datastreams did not receive any data values and were left out of the plot. Only the non-empty datastreams are shown above, which is why they are out of order.

The first datastream in the plot, which is labeled “Packet Byte Count-2”, corresponds to the wildcard match with the “*” replaced with “2”. The seventh datastream in the plot, which is labeled “Packet Byte Count-9”, corresponds to the

wildcard match with the "*" replaced with "9". Also, note that the function call to `PlotProbe()` will give an error message if there are no matches for a path that contains wildcards.

```
//============================================================
//
// An application packet probe will be hooked to the PacketSink Rx
// trace source using the new gnuplot helper API: PlotProbe() and
// the new file helper API: WriteProbe().
//
//============================================================

// Create the gnuplot helper.
GnuplotHelper packetPlotHelperNewAPI;

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

// Plot the values generated by the probe.
packetPlotHelperNewAPI.PlotProbe ("ns3::ApplicationPacketProbe",
                                  "/NodeList/*/ApplicationList/0/$ns3::PacketSink/Rx",
                                  "OutputBytes",
                                  "Packet Byte Count",
                                  GnuplotAggregator::KEY_INSIDE);
```

The following 2-D gnuplot was also created using the example.

The code that produced the gnuplot is below and also shows how the GnuplotHelper is used to connect Probes directly to a GnuplotAggregator.

Note that because there were 20 matches for the wildcard in the path, 20 datastreams were drawn in the plot. The first datastream in the plot, which is labeled "Routing Table Size-0", corresponds to the wildcard match with the "*" replaced with "0". The twentieth datastream in the plot, which is labeled "Routing Table Size-19", corresponds to the wildcard match with the "*" replaced with "19". Also, note that the function call to `PlotProbe()` will give an error message if there are no matches for a path that contains wildcards.

```
//============================================================
//
// An application packet probe will be hooked to the PacketSink Rx
// trace source using the new gnuplot helper API: PlotProbe() and
// the new file helper API: WriteProbe().
//
//============================================================

// Create the gnuplot helper.
GnuplotHelper routingTableSizePlotHelperNewAPI;

// Configure the plot.
routingTableSizePlotHelperNewAPI.ConfigurePlot ("manet-safe-aodv-routing-table-size-new-api",
                                                "MANET SAFE AODV Routing Table Size Aggregator uses P
                                                "Time (Seconds)",
                                                "Routing Table Size",
                                                "png");

// Plot the values generated by the probe.
routingTableSizePlotHelperNewAPI.PlotProbe ("ns3::Uinteger16Probe",
```

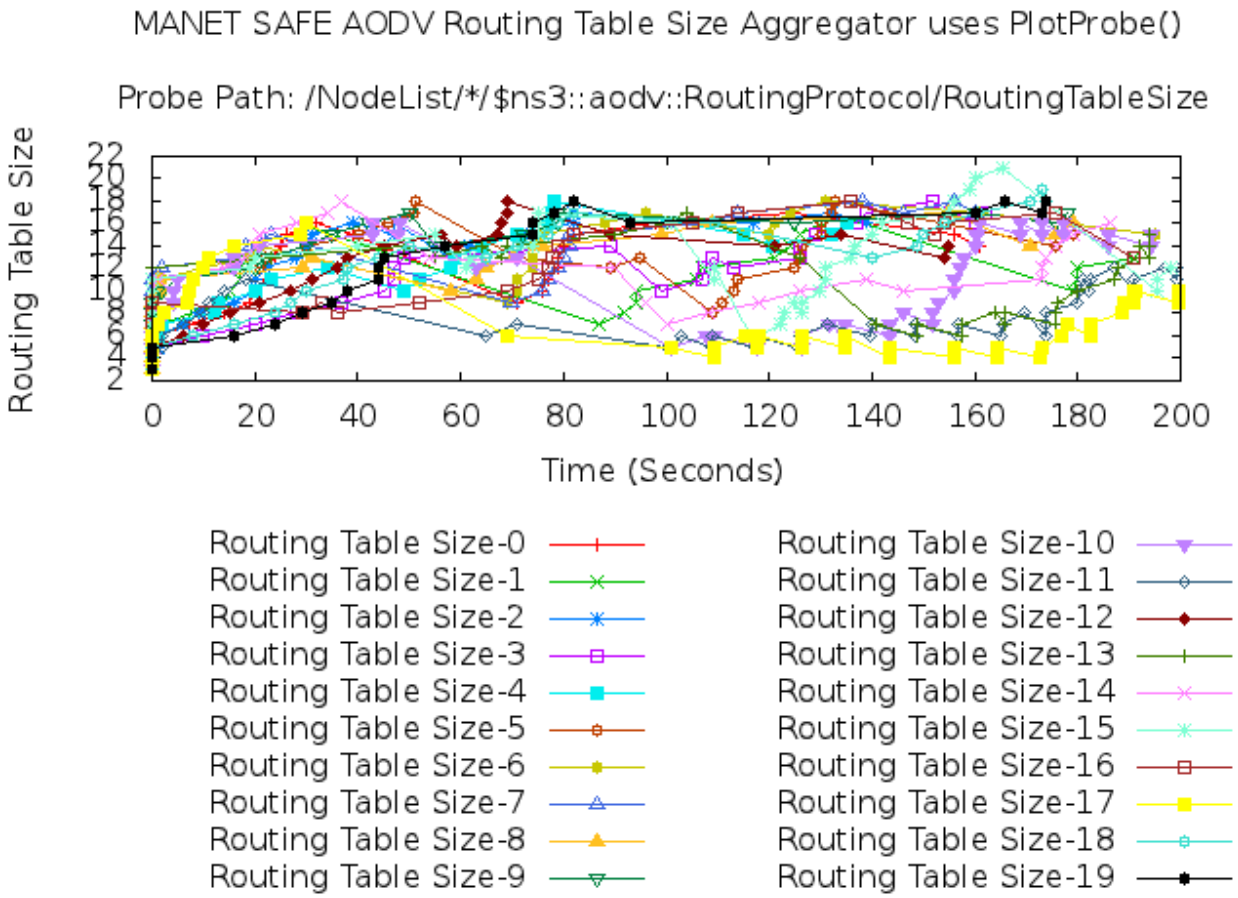

Figure 10.6: 2-D Gnuplot Created by manet-safe.cc Example.

```
                                    "/NodeList/*/$ns3::aodv::RoutingProtocol/RoutingTableSize
                                    "Output",
                                    "Routing Table Size",
                                    GnuplotAggregator::KEY_BELOW);
```

The following 2-D gnuplot was also created using the example.

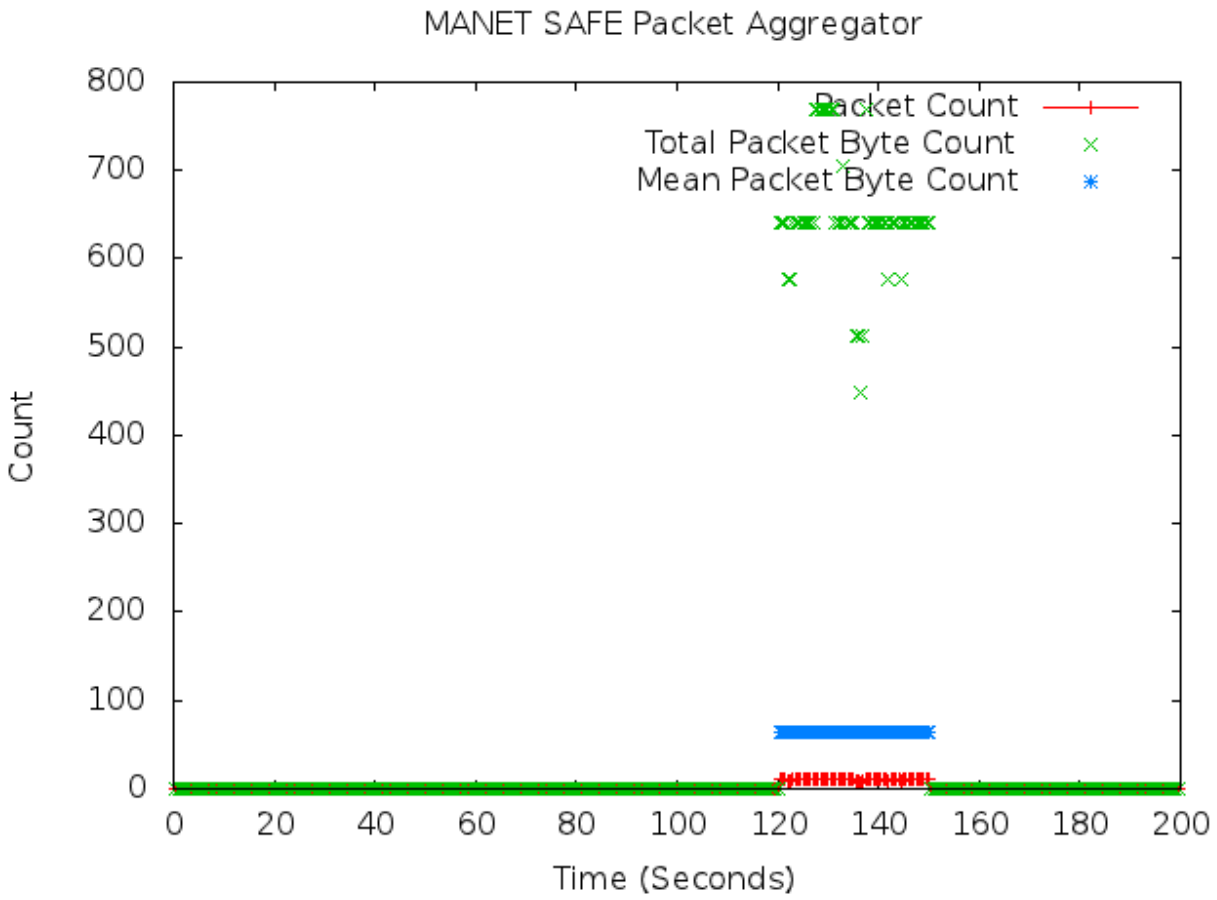

Figure 10.7: 2-D Gnuplot Created by manet-safe.cc Example.

The code that produced the gnuplot is below and shows how the GnuplotHelper is used to connect Probes to Collectors, which are then connected to a GnuplotAggregator.

```
//============================================================
//
// 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");
```

First, an ApplicationPacketProbe is added.

```
// 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> packetPlotProbe = packetPlotHelper.GetProbe ("PacketSinkRxProbe");
packetPlotProbe->SetAttribute ("Start", TimeValue (Seconds (120.0)));
packetPlotProbe->SetAttribute ("Stop", TimeValue (Seconds (150.0)));
```

Second, a BasicStatsCollector is added.

```
// 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> packetPlotCollector = packetPlotHelper.GetCollector ("PacketSinkRxCollector");
packetPlotCollector->SetPeriodic (Seconds (0.5));
```

Third, the 2-D datasets to be plotted are set up.

```
// Get a pointer to the helper's aggregator so that it can be configured.
Ptr<GnuplotAggregator> packetPlotAggregator = packetPlotHelper.GetAggregator ();
packetPlotAggregator->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.
packetPlotAggregator->Set2dDatasetStyle ("PacketSinkRxCollector/SampleCount",
                                         Gnuplot2dDataset::LINES_POINTS);
```

10.2.3 FileHelper

The FileHelper is a helper class used to put data values into a file. The overall goal is to provide the ability for users to quickly make formatted text files from data exported in *ns-3* trace sources. By default, a minimal amount of

data transformation is performed; the objective is to generate files with as few (default) configuration statements as possible.

FileHelper Overview

The FileHelper will create 1 or more text files at the end of the simulation.

The FileHelper can create 4 different types of text files:

- Formatted
- Space separated (the default)
- Comma separated
- Tab separated

Formatted files use C-style format strings and the sprintf() function to print their values in the file being written.

The following text file with 2 columns of formatted values named `sixth-dcf-packet-byte-count-0.txt` was created using more new code that was added to the original *ns-3* Tutorial example's code. Only the first 10 lines of this file are shown here for brevity.

```
Time (Seconds) = 1.000e+00    Packet Byte Count = 40
Time (Seconds) = 1.004e+00    Packet Byte Count = 40
Time (Seconds) = 1.004e+00    Packet Byte Count = 576
Time (Seconds) = 1.009e+00    Packet Byte Count = 576
Time (Seconds) = 1.009e+00    Packet Byte Count = 576
Time (Seconds) = 1.015e+00    Packet Byte Count = 512
Time (Seconds) = 1.017e+00    Packet Byte Count = 576
Time (Seconds) = 1.017e+00    Packet Byte Count = 544
Time (Seconds) = 1.025e+00    Packet Byte Count = 576
Time (Seconds) = 1.025e+00    Packet Byte Count = 544

...
```

The following different text file with 2 columns of formatted values named `sixth-dcf-packet-byte-count-1.txt` was also created using the same new code that was added to the original *ns-3* Tutorial example's code. Only the first 10 lines of this file are shown here for brevity.

```
Time (Seconds) = 1.002e+00    Packet Byte Count = 40
Time (Seconds) = 1.007e+00    Packet Byte Count = 40
Time (Seconds) = 1.013e+00    Packet Byte Count = 40
Time (Seconds) = 1.020e+00    Packet Byte Count = 40
Time (Seconds) = 1.028e+00    Packet Byte Count = 40
Time (Seconds) = 1.036e+00    Packet Byte Count = 40
Time (Seconds) = 1.045e+00    Packet Byte Count = 40
Time (Seconds) = 1.053e+00    Packet Byte Count = 40
Time (Seconds) = 1.061e+00    Packet Byte Count = 40
Time (Seconds) = 1.069e+00    Packet Byte Count = 40

...
```

The new code that was added to produce the two text files is below. More details about this API will be covered in a later section.

Note that because there were 2 matches for the wildcard in the path, 2 separate text files were created. The first text file, which is named "sixth-dcf-packet-byte-count-0.txt", corresponds to the wildcard match with the "*" replaced with "0". The second text file, which is named "sixth-dcf-packet-byte-count-1.txt", corresponds to the wildcard match with the "*" replaced with "1". Also, note that the function call to `WriteProbe()` will give an error message if there are no matches for a path that contains wildcards.

```
// Create the file helper.
FileHelper fileHelper;

// Configure the file to be written.
fileHelper.ConfigureFile ("sixth-dcf-packet-byte-count",
                          FileAggregator::FORMATTED);

// Set the labels for this formatted output file.
fileHelper.Set2dFormat ("Time (Seconds) = %.3e\tPacket Byte Count = %.0f");

// Write the values generated by the probe.
fileHelper.WriteProbe ("ns3::Ipv4PacketProbe",
                       "/NodeList/*/$ns3::Ipv4L3Protocol/Tx",
                       "OutputBytes");
```

FileHelper ConfigureFile

The FileHelper's `ConfigureFile()` function can be used to configure text files.

It has the following prototype:

```
void ConfigureFile (const std::string &outputFileNameWithoutExtension,
                    enum FileAggregator::FileType fileType = FileAggregator::SPACE_SEPARATED);
```

It has the following arguments:

Argument	Description
outputFileNameWithoutExtension	Name of output file to write with no extension.
fileType	Type of file to write. The default type of file is space separated.

The FileHelper's `ConfigureFile()` function configures text file related parameters for the file helper so that it will create a file named outputFileNameWithoutExtension plus possible extra information from wildcard matches plus ".txt" with values printed as specified by fileType. The default file type is space-separated.

An example of how to use this function can be seen in the `sixth-dcf.cc` code described above where it was used as follows:

```
fileHelper.ConfigureFile ("sixth-dcf-packet-byte-count",
                          FileAggregator::FORMATTED);
```

FileHelper WriteProbe

The FileHelper's `WriteProbe()` function can be used to write values generated by probes to text files.

It has the following prototype:

```
void WriteProbe (const std::string &typeId,
                 const std::string &path,
                 const std::string &probeTraceSource);
```

It has the following arguments:

Argument	Description
typeId	The type ID for the probe used when it is created.
path	Config path to access the probe.
probeTraceSource	The probe trace source to access.

The FileHelper's `WriteProbe()` function creates output text files generated by hooking the ns-3 trace source with a probe, and then writing the values from the probeTraceSource. The output file names will have the text stored in the member variable m_outputFileNameWithoutExtension plus ".txt", and will consist of the 'newValue' at each timestamp.

If the config path has more than one match in the system because there is a wildcard, then one output file for each match will be created. The output file names will contain the text in m_outputFileNameWithoutExtension plus the matched characters for each of the wildcards in the config path, separated by dashes, plus ".txt". For example, if the value in m_outputFileNameWithoutExtension is the string "packet-byte-count", and there are two wildcards in the path, then output file names like "packet-byte-count-0-0.txt" or "packet-byte-count-12-9.txt" will be possible as names for the files that will be created.

An example of how to use this function can be seen in the `sixth-dcf.cc` code described above where it was used as follows:

```
fileHelper.WriteProbe ("ns3::Ipv4PacketProbe",
                       "/NodeList/*/$ns3::Ipv4L3Protocol/Tx",
                       "OutputBytes");
```

Other Examples

File Helper Example

A slightly simpler example than the `sixth-dcf.cc` example can be found in `src/stats/examples/file-helper-example.cc`. This example only uses the FileHelper, not the FileHelper. It is also more of a toy example than `sixth-dcf.cc` because it has a made-up trace source created for demonstration purposes.

The following text file with 2 columns of formatted values named `file-helper-example.txt` was created using the example. Only the first 10 lines of this file are shown here for brevity.

```
Time (Seconds) = 4.995e-01    Count = 1
Time (Seconds) = 1.463e+00    Count = 2
Time (Seconds) = 1.678e+00    Count = 3
Time (Seconds) = 3.972e+00    Count = 4
Time (Seconds) = 4.150e+00    Count = 5
Time (Seconds) = 8.066e+00    Count = 6
Time (Seconds) = 8.731e+00    Count = 7
Time (Seconds) = 9.807e+00    Count = 8
Time (Seconds) = 1.078e+01    Count = 9
Time (Seconds) = 1.083e+01    Count = 10

...
```

In this example, there is an Emitter object that increments its counter at various random times and then emits the counter's value as a trace source.

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

The following code is probing the Counter exported by the emitter object. This DoubleProbe 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> probe = CreateObject<DoubleProbe> ();
probe->SetName ("PathProbe");
Names::Add ("/Names/Probe", probe);
```

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

Note that because there are no wildcards in the path used below, only 1 text file was created. This single text file is simply named "file-helper-example.txt", with no extra suffixes like you would see if there were wildcards in the path.

```
// Create the file helper.
FileHelper fileHelper;

// Configure the file to be written.
fileHelper.ConfigureFile ("file-helper-example",
                          FileAggregator::FORMATTED);

// Set the labels for this formatted output file.
fileHelper.Set2dFormat ("Time (Seconds) = %.3e\tCount = %.0f");

// Write the values generated by the probe.  The path that we
// provide helps to disambiguate the source of the trace.
fileHelper.WriteProbe ("ns3::DoubleProbe",
                       "/Names/Probe/Output",
                       "Output");
```

MANET SAFE Example

Another example that exercises the FileHelper can be found in `src/stats/examples/manet-safe.cc`. This example is even better than the `sixth-dcf.cc` example in terms being a realistic simulation performed using *ns-3*. It is, therefore, more complicated and is doing more things with the DCF objects.

The following text file with 2 columns of formatted values named `manet-safe-time-versus-packet-byte-count-new-ap` was created by the example. Only the first 10 lines of this file are shown here for brevity.

```
Time (Seconds) = 1.008e+02    Count = 64
Time (Seconds) = 1.011e+02    Count = 64
Time (Seconds) = 1.013e+02    Count = 64
Time (Seconds) = 1.027e+02    Count = 64
Time (Seconds) = 1.027e+02    Count = 64
Time (Seconds) = 1.028e+02    Count = 64
Time (Seconds) = 1.031e+02    Count = 64
Time (Seconds) = 1.033e+02    Count = 64
Time (Seconds) = 1.036e+02    Count = 64
Time (Seconds) = 1.038e+02    Count = 64

...
```

The following different text file with 2 columns of formatted values named `manet-safe-time-versus-packet-byte-count-new-api-9.txt` was also created by the example. Only the first 10 lines of this file are shown here for brevity.

```
Time (Seconds) = 1.008e+02    Count = 64
Time (Seconds) = 1.010e+02    Count = 64
Time (Seconds) = 1.013e+02    Count = 64
Time (Seconds) = 1.015e+02    Count = 64
Time (Seconds) = 1.018e+02    Count = 64
Time (Seconds) = 1.020e+02    Count = 64
Time (Seconds) = 1.023e+02    Count = 64
Time (Seconds) = 1.025e+02    Count = 64
Time (Seconds) = 1.028e+02    Count = 64
```

```
Time (Seconds) = 1.030e+02    Count = 64

...
```

The code that produced the two text files is below and shows how the FileHelper is used to connect Probes directly to a FileAggregators.

Note that because there were 10 matches for the wildcard in the path, 10 separate text files were created. But, 3 of the 10 text files did not receive any data values and were empty. Only the non-empty text files are shown above, which is why they are out of order.

The first non-empty text file shown, which is named "manet-safe-time-versus-packet-byte-count-new-api-2.txt", corresponds to the wildcard match with the "*" replaced with "2". The second non-empty text file, which is named "manet-safe-time-versus-packet-byte-count-new-api-9.txt", corresponds to the wildcard match with the "*" replaced with "9". Also, note that the function call to `WriteProbe()` will give an error message if there are no matches for a path that contains wildcards.

```
// Create the file helper.
FileHelper packetFileHelperNewAPIUsesWildcard;

// Configure the file to be written.
packetFileHelperNewAPIUsesWildcard.ConfigureFile ("manet-safe-time-versus-packet-byte-count-new-api",
                                                  FileAggregator::FORMATTED);

// Set the labels for this formatted output file.
packetFileHelperNewAPIUsesWildcard.Set2dFormat ("Time (Seconds) = %.3e\tCount = %.0f");

// Write the values generated by the probe.
packetFileHelperNewAPIUsesWildcard.WriteProbe ("ns3::ApplicationPacketProbe",
                                               "/NodeList/*/ApplicationList/0/$ns3::PacketSink/Rx",
                                               "OutputBytes");
```

The following text file with 2 columns of space separated values named `manet-safe-time-versus-aodv-routing-table-size-new-api-0.txt` was created by the example. Only the first 10 lines of this file are shown here for brevity.

```
Time    AODV Routing Table Size
----    -----------------------
0.00125458 3
0.0442549 4
1.05689 5
1.09395 6
7.04664 7
14.0383 8
19.0026 9

...
```

The following different text file with 2 columns of space separated values named `manet-safe-time-versus-aodv-routing-table-size-new-api-19.txt` was also created by the example. Only the first 10 lines of this file are shown here for brevity.

```
Time    AODV Routing Table Size
----    -----------------------
0.0202541 3
0.0222545 4
0.0752545 5
16.0386 6
24.0815 7
```

```
29.0351 8
34.9946 9
37.981 10

...
```

The code that produced the two text files is below and shows how the FileHelper is used to connect Probes directly to a FileAggregators.

Note that because there were 20 matches for the wildcard in the path, 20 separate text files were created. The first text file, which is named "manet-safe-time-versus-aodv-routing-table-size-new-api-0.txt", corresponds to the wildcard match with the "*" replaced with "0". The twentieth text file, which is named "manet-safe-time-versus-aodv-routing-table-size-new-api-19.txt", corresponds to the wildcard match with the "*" replaced with "19". Also, note that the function call to `WriteProbe()` will give an error message if there are no matches for a path that contains wildcards.

```
// Create the file helper.
FileHelper routingTableSizeFileHelperNewAPIUsesWildcard;

// Configure the file to be written.
routingTableSizeFileHelperNewAPIUsesWildcard.ConfigureFile ("manet-safe-time-versus-aodv-routing-tabl
                                                            FileAggregator::SPACE_SEPARATED);

// Set the heading for this output file.
routingTableSizeFileHelperNewAPIUsesWildcard.SetHeading ("Time    AODV Routing Table Size\n----    --

// Write the values generated by the probe.
routingTableSizeFileHelperNewAPIUsesWildcard.WriteProbe ("ns3::Uinteger16Probe",
                                                         "/NodeList/*/$ns3::aodv::RoutingProtocol/Rou
                                                         "Output");
```

The following text file with 2 columns of tab separated values named `manet-safe-time-versus-packet-count.txt` was also created by the example. Only the first 10 lines of this file are shown here for brevity.

```
0.5   0
1     0
1.5   0
2     0
2.5   0
3     0
3.5   0
4     0
4.5   0
5     0

  ...
```

The code that produced the text file is below and shows how the FileHelper is used to connect Probes to Collectors, which are then connected to a FileAggregator.

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

// Create the file helper.
FileHelper packetFileHelper;
```

```
// Configure the file.
packetFileHelper.ConfigureFile ("manet-safe-time-versus-packet-count",
                                FileAggregator::TAB_SEPARATED);
```

First, an ApplicationPacketProbe is added.

```
// Add a probe to the file helper.
packetFileHelper.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> packetFileProbe = packetFileHelper.GetProbe ("PacketSinkRxProbe");
packetFileProbe->SetAttribute ("Start", TimeValue (Seconds (120.0)));
packetFileProbe->SetAttribute ("Stop", TimeValue (Seconds (150.0)));
```

Second, a BasicStatsCollector is added.

```
// Add a collector to the file helper.
packetFileHelper.Add2dCollector ("ns3::BasicStatsCollector",
                                 "PacketSinkRxCollector",
                                 "PacketSinkRxProbe",
                                 "OutputBytes");

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

Third, the 2-D dataset to be written to the file is set up.

```
// Add a dataset to the file.  Note that the dataset context string,
// which is the third argument in this function call, must be unique
packetFileHelper.Add2dDataset ("PacketSinkRxCollector",
                               "SampleCount",
                               "PacketSinkRxCollector/SampleCount");
```

10.2.4 Scope and Limitations

Currently, only these Probes have been implemented and connected to the GnuplotHelper and to the FileHelper:

- DoubleProbe
- Uinteger8Probe
- Uinteger16Probe
- Uinteger32Probe
- PacketProbe
- ApplicationPacketProbe
- Ipv4PacketProbe

These Probes, therefore, are the only ones available to be used in `PlotProbe()` and `WriteProbe()`.

There is another structural limitation to these helpers, that has resulted in the need to hardcode some support for each probe and collector type into the helper.

Currently, support for connecting the collector to the probe must be explicitly coded in this helper based on the typeId passed in, because the current MakeCallback/TraceConnect API requires a function pointer of the right type. As new

collector types are added, this method should be extended, until a more general solution for working directly with the typeId strings is supported.

In summary, if one adds a new collector or probe type, and one wants to make use of it via the helper API, one must explicitly add support for these new types in the following functions:

- GnuplotHelper::AddCollector
- GnuplotHelper::ConnectProbeToAggregator
- FileHelper::Add2dCollector
- FileHelper::ConnectProbeToAggregator

In the next few sections, we cover each of the fundamental object types (Probe, Collector, and Aggregator) in more detail, and show how they can be connected together using lower-level API.

10.3 Probes

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.

10.3.1 Probe 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 [Cic06]

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

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 DataCollectionObject, and two in the Probe base class:

- "Name" (DataCollectionObject), a StringValue
- "Enabled" (DataCollectionObject), 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.

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.

10.3.2 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/stats/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 stats module header file was included in the example .cc file
2. the example was made dependent on the stats 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));
```

10.3.3 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.
- ApplicationPacketProbe connects to an *ns-3* trace source exporting a packet and a socket address.
- Ipv4PacketProbe connects to an *ns-3* trace source exporting a packet, an IPv4 object, and an interface.

10.3.4 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.

10.3.5 Examples

Two examples will be discussed in detail here:

- Double Probe Example
- IPv4 Packet Plot Example

Double Probe Example

The double probe example has been discussed previously. The example program can be found in `src/stats/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);
}
```

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/stats/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 thi
                    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;
    }
}
```

10.3.6 References

10.4 Collectors

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.

10.4.1 Collector 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.

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, DataCollectionObject, and four in the Collector base class:

- "Name" (DataCollectionObject), a StringValue
- "Enabled" (DataCollectionObject), 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);
```

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.

10.4.2 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/stats/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_collec
```

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 (&BasicStatsCollectorAsynchronousP
```

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()`.

Examples

There are currently no examples that exercise the BasicStatsCollector.

Tests

One test will be discussed in detail here:

- Basic Stats Collector Test

Basic Stats Collector Test

The basic stats collector test has been discussed previously. The test suite can be found in `src/stats/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 ();
}
```

10.5 Aggregators

This section details the functionalities provided by the Aggregator class to an *ns-3* simulation. 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.

10.5.1 Aggregator 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.

To date, two Aggregators have been implemented:

- GnuplotAggregator
- FileAggregator

10.5.2 GnuplotAggregator

The GnuplotAggregator produces output files used to make gnuplots.

The GnuplotAggregator will create 3 different files at the end of the simulation:

- A space separated gnuplot data file
- A gnuplot control file
- A shell script to generate the gnuplot

Creation

An object of type GnuplotAggregator will be created here to show what needs to be done.

One declares a GnuplotAggregator in dynamic memory by using the smart pointer class (Ptr<T>). To create a GnuplotAggregator in dynamic memory with smart pointers, one just needs to call the *ns-3* method `CreateObject()`. The following code from `src/stats/examples/gnuplot-aggregator-example.cc` shows how to do this:

```
string fileNameWithoutExtension = "gnuplot-aggregator";

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

The first argument for the constructor, fileNameWithoutExtension, is the name of the gnuplot related files to write with no extension. This GnuplotAggregator will create a space separated gnuplot data file named "gnuplot-aggregator.dat", a gnuplot control file named "gnuplot-aggregator.plt", and a shell script to generate the gnuplot named + "gnuplot-aggregator.sh".

The gnuplot that is created can have its key in 4 different locations:

- No key
- Key inside the plot (the default)
- Key above the plot
- Key below the plot

The following gnuplot key location enum values are allowed to specify the key's position:

```
enum KeyLocation {
  NO_KEY,
  KEY_INSIDE,
  KEY_ABOVE,
  KEY_BELOW
};
```

If it was desired to have the key below rather than the default position of inside, then you could do the following.

```
aggregator->SetKeyLocation(GnuplotAggregator::KEY_BELOW);
```

Examples

One example will be discussed in detail here:

- Gnuplot Aggregator Example

Gnuplot Aggregator Example

An example that exercises the GnuplotAggregator can be found in `src/stats/examples/gnuplot-aggregator-example.c`

The following 2-D gnuplot was created using the example.

This code from the example shows how to construct the GnuplotAggregator as was discussed above.

```
void Create2dPlot ()
{
  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);
```

Various GnuplotAggregator attributes are set including the 2-D dataset that will be plotted.

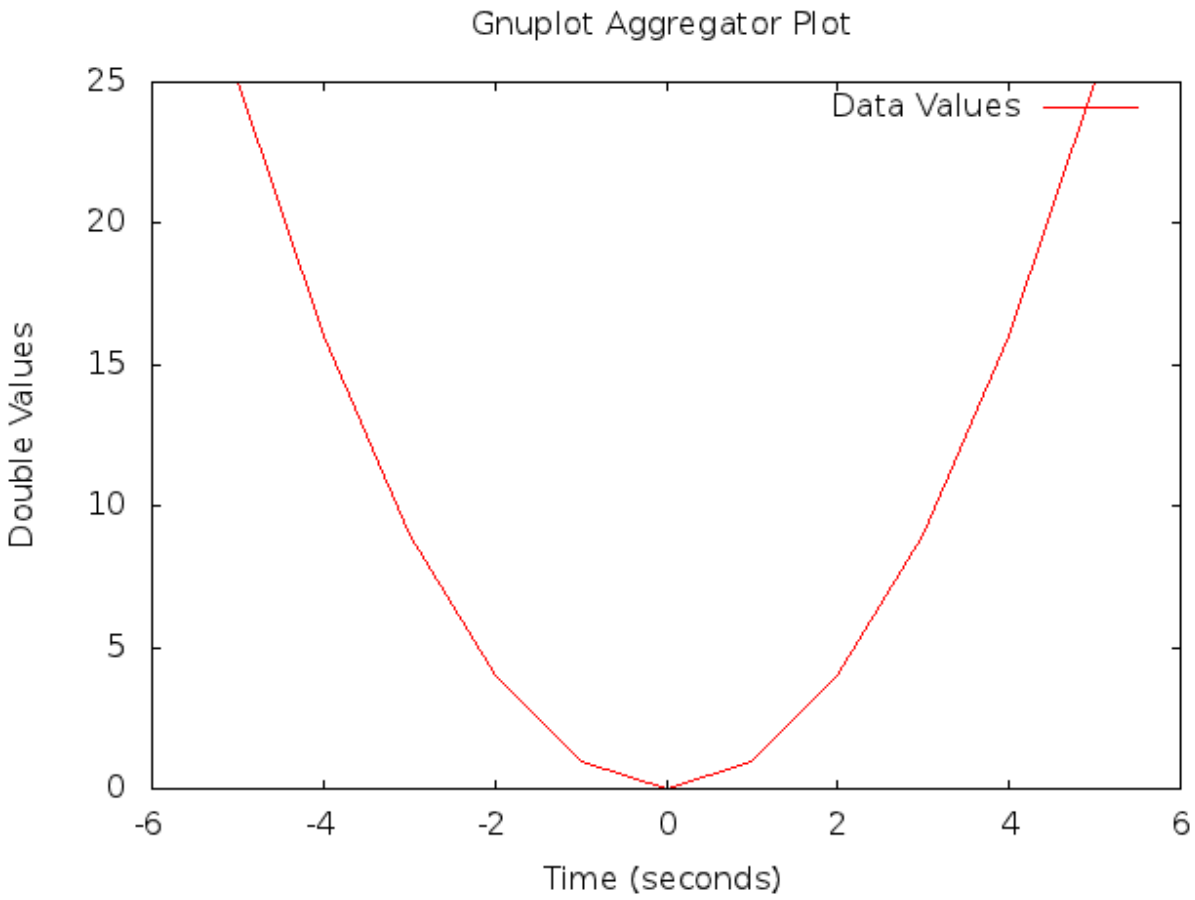

Figure 10.8: 2-D Gnuplot Created by gnuplot-aggregator-example.cc Example.

```
// 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 ();
```

Next, the 2-D values are calculated, and each one is individually written to the GnuplotAggregator using the `Write2d()` function.

```
  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 ();
}
```

10.5.3 FileAggregator

The FileAggregator sends the values it receives to a file.

The FileAggregator can create 4 different types of files:

- Formatted
- Space separated (the default)
- Comma separated
- Tab separated

Formatted files use C-style format strings and the sprintf() function to print their values in the file being written.

Creation

An object of type FileAggregator will be created here to show what needs to be done.

One declares a FileAggregator in dynamic memory by using the smart pointer class (Ptr<T>). To create a FileAggregator in dynamic memory with smart pointers, one just needs to call the *ns-3* method CreateObject. The following code from `src/stats/examples/file-aggregator-example.cc` shows how to do this:

```
string fileName       = "file-aggregator-formatted-values.txt";

// Create an aggregator that will have formatted values.
Ptr<FileAggregator> aggregator =
  CreateObject<FileAggregator> (fileName, FileAggregator::FORMATTED);
```

The first argument for the constructor, filename, is the name of the file to write; the second argument, fileType, is type of file to write. This FileAggregator will create a file named "file-aggregator-formatted-values.txt" with its values printed as specified by fileType, i.e., formatted in this case.

The following file type enum values are allowed:

```
enum FileType {
  FORMATTED,
  SPACE_SEPARATED,
  COMMA_SEPARATED,
  TAB_SEPARATED
};
```

Examples

One example will be discussed in detail here:

- File Aggregator Example

File Aggregator Example

An example that exercises the FileAggregator can be found in `src/stats/examples/file-aggregator-example.cc`.

The following text file with 2 columns of values separated by commas was created using the example.

```
-5,25
-4,16
-3,9
-2,4
-1,1
0,0
1,1
2,4
3,9
4,16
5,25
```

This code from the example shows how to construct the FileAggregator as was discussed above.

```
void CreateCommaSeparatedFile ()
{
  using namespace std;

  string fileName       = "file-aggregator-comma-separated.txt";
  string datasetContext = "Dataset/Context/String";

  // Create an aggregator.
  Ptr<FileAggregator> aggregator =
    CreateObject<FileAggregator> (fileName, FileAggregator::COMMA_SEPARATED);
```

FileAggregator attributes are set.

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

Next, the 2-D values are calculated, and each one is individually written to the FileAggregator using the `Write2d()` function.

```
  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 ();
}
```

The following text file with 2 columns of formatted values was also created using the example.

```
Time = -5.000e+00     Value = 25
Time = -4.000e+00     Value = 16
Time = -3.000e+00     Value = 9
Time = -2.000e+00     Value = 4
Time = -1.000e+00     Value = 1
Time = 0.000e+00      Value = 0
Time = 1.000e+00      Value = 1
Time = 2.000e+00      Value = 4
Time = 3.000e+00      Value = 9
Time = 4.000e+00      Value = 16
Time = 5.000e+00      Value = 25
```

This code from the example shows how to construct the FileAggregator as was discussed above.

```
void CreateFormattedFile ()
{
  using namespace std;

  string fileName       = "file-aggregator-formatted-values.txt";
  string datasetContext = "Dataset/Context/String";

  // Create an aggregator that will have formatted values.
  Ptr<FileAggregator> aggregator =
    CreateObject<FileAggregator> (fileName, FileAggregator::FORMATTED);
```

FileAggregator attributes are set, including the C-style format string to use.

```
// Set the format for the values.
aggregator->Set2dFormat ("Time = %.3e\tValue = %.0f");
```

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

Next, the 2-D values are calculated, and each one is individually written to the FileAggregator using the `Write2d()` function.

```
  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 ();
}
```

10.6 Adaptors

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

Note: the term 'adaptor' may also be spelled 'adapter'; we chose the spelling aligned with the C++ standard.

10.6.1 Adaptor Overview

An Adaptor is used to make connections between different types of DCF objects.

To date, one Adaptor has been implemented:

- TimeSeriesAdaptor

10.6.2 Time Series Adaptor

The TimeSeriesAdaptor lets Probes connect directly to Aggregators without needing any Collector in between.

Both of the implemented DCF helpers utilize TimeSeriesAdaptors in order to take probed values of different types and output the current time plus the value with both converted to doubles.

The role of the TimeSeriesAdaptor class is that of an adaptor, which takes raw-valued probe data of different types and outputs a tuple of two double values. The first is a timestamp, which may be set to different resolutions (e.g. Seconds, Milliseconds, etc.) but which defaults to seconds. The second is the conversion of a non-double value to a double value (possibly with loss of precision).

10.7 Examples that Use the Low-level DCF API Functions

The following examples that will be described in detail in this section do not use the DCF helpers. Instead, they all use the low-level DCF API functions.

- IPv4 Packet Plot No Helper Example

10.7.1 IPv4 Packet Plot No Helper Example

The low-level example `src/stats/examples/ipv4-packet-plot-no-helper-example.cc`. is based on the example `src/stats/examples/ipv4-packet-plot-example.cc`, which was discussed above. Both of these examples are based on the fifth.cc example from the *ns-3* Tutorial.

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

An IPv4 packet probe will be hooked to the Ipv4L3Protocol Tx trace source, and then three plots will be created.

```
//Create the packet probe
Ptr<Ipv4PacketProbe> packetProbe = CreateObject<Ipv4PacketProbe> ();
packetProbe->Enable();

//Create the collector
Ptr<BasicStatsCollector> collector = CreateObject<BasicStatsCollector> ();
collector->SetPeriodic (Seconds (0.5));
collector->Enable();
```

First, the packet count will be plotted.

```
//Create the gnuplot aggregator 1
Ptr<GnuplotAggregator> gnuplotAgg1 = CreateObject<GnuplotAggregator> ("IPv4_PacketCountPlot");
gnuplotAgg1->Set2dDatasetDefaultStyle (Gnuplot2dDataset::LINES);
gnuplotAgg1->SetTitle("Packet Count vs. Time");
gnuplotAgg1->SetLegend("Packet Count", "Time (Seconds)");
gnuplotAgg1->Add2dDataset("dataset", "Packet count");
gnuplotAgg1->SetTerminal("pdf");
gnuplotAgg1->Enable();
```

Second, the mean packet size will be plotted.

```
//Create the gnuplot aggregator 2
Ptr<GnuplotAggregator> gnuplotAgg2 = CreateObject<GnuplotAggregator> ("IPv4_MeanPacketSizePlot");
gnuplotAgg2->Set2dDatasetDefaultStyle (Gnuplot2dDataset::LINES);
gnuplotAgg2->SetTitle("Mean Packet Size vs. Time");
gnuplotAgg2->SetLegend("Mean Packet Size", "Time (Seconds)");
gnuplotAgg2->Add2dDataset("dataset", "Mean Packet Size");
gnuplotAgg2->SetTerminal("pdf");
gnuplotAgg2->Enable();
```

Third, the sum of the packet bytes will be plotted.

```
//Create the gnuplot aggregator 3
Ptr<GnuplotAggregator> gnuplotAgg3 = CreateObject<GnuplotAggregator> ("IPv4_PacketByteSumPlot");
gnuplotAgg3->Set2dDatasetDefaultStyle (Gnuplot2dDataset::LINES);
gnuplotAgg3->SetTitle("Packet Byte Sum vs. Time");
gnuplotAgg3->SetLegend("Packet Byte Sum", "Time (Seconds)");
gnuplotAgg3->Add2dDataset("dataset", "Packet Byte Sum");
gnuplotAgg3->SetTerminal("pdf");
gnuplotAgg3->Enable();
```

Next, the Emitter trace source will be connected to the Probe, and the Probe will be connected to the Collector.

```
//Emitter <--> probe
packetProbe->ConnectByPath("/NodeList/0/$ns3::Ipv4L3Protocol/Tx");

//Hook packet probe <--> collector
packetProbe->TraceConnectWithoutContext ("OutputBytes", MakeCallback (&BasicStatsCollector::TraceSink
//packetProbe->TraceConnectWithoutContext ("OutputBytes", MakeCallback (&echoInt));
```

Finally, the Collector will be connected to all of the plots.

```
//Hook collector <--> gnuplot 1
collector->TraceConnect ("SampleCount", "dataset", MakeCallback (&GnuplotAggregator::Write2d, gnuplot
//collector->TraceConnectWithoutContext ("SampleCount",  MakeCallback(&echoDouble));

//Hook collector <--> gnuplot 2
collector->TraceConnect ("SampleMean", "dataset", MakeCallback (&GnuplotAggregator::Write2d, gnuplotA


//Hook collector <--> gnuplot 3
collector->TraceConnect ("SampleSum", "dataset", MakeCallback (&GnuplotAggregator::Write2d, gnuplotAg
```

10.8 Scope/Limitations

This section discusses the scope and limitations of the Data Collection Framework.

Currently, only these Probes have been implemented in DCF:

- DoubleProbe
- Uinteger8Probe
- Uinteger16Probe
- Uinteger32Probe
- PacketProbe
- ApplicationPacketProbe
- Ipv4PacketProbe

Currently, only this Collector has been implemented in DCF:

- BasicStatsCollector

Currently, only these Aggregators have been implemented in DCF:

- GnuplotAggregator
- FileAggregator

Currently, only this Adaptor has been implemented in DCF:

Time-Series Adaptor.

10.8.1 Future Work

This section discusses the future work to be done on the Data Collection Framework.

Here are some things that still need to be done:

- Hook up more trace sources in *ns-3* code to get more values out of the simulator.
- Implement more types of Probes than there currently are.
- Implement more than just the single current 2-D Collector, BasicStatsCollector.
- Implement more Aggregators.
- Implement more than just Adaptors.