FlowMonitor - a network monitoring framework for the Network Simulator 3 (NS-3)

Gustavo Carneiro
INESC Porto
Faculdade de Engenharia
Universidade do Porto
gjc@inescporto.pt

Pedro Fortuna
INESC Porto
Faculdade de Engenharia
Universidade do Porto
pedro.fortuna@inescporto.pt

Manuel Ricardo
INESC Porto
Faculdade de Engenharia
Universidade do Porto
mricardo@inescporto.pt

The authors would like to thank the support received from the Portuguese Fundação para a Ciência e Tecnologia under the fellowships SFRH/BD/23456/2005 and SFRH/BD/-22514/2005.

ABSTRACT

When networking researchers meet the task of doing simulations, there is always a need to evaluate the value of such models by measuring a set of well known network performance metrics. However, simulators in general and NS-3 in particular, require significant programming effort from the researcher in order to collect those metrics.

This paper reports a contribution for NS-3 consisting of a new flow monitoring module that makes it easier to collect and save to persistent storage a common set of network performance metrics. The module automatically detects all flows passing through the network and stores in a file most of the metrics that a researcher might need to analyze about the flow, such as bitrates, duration, delays, packet sizes, and packet loss ratio.

The value of this module is demonstrated using an easy to follow example. It is also validated by comparing the measurements of a simple scenario with the expected values. Finally, the performance of flow monitoring is characterized and shown to introduce small overheads.

1. INTRODUCTION

Network monitoring is accomplished by inspecting the traffic in a network; this technique can be employed for purposes such as: fault detection—to detect disruptions in the network connectivity, routing or services; performance evaluation—to measure the network utilization or the performance of network protocols; security monitoring—to detect possible security problems; and Service Level Agreements (SLA) monitoring—to verify if a given network service is performing as specified in contract.

NSTOOLS 2009, October 19, 2009 — Pisa, Italy.

Network monitoring may pose the following problems: 1) monitoring strategy—it can follow a passive approach, where the traffic present in the network is measured, or an active approach, where test traffic is injected in the network for testing purposes. In both cases the data collected can then be sent to a central point where a complete picture of the network is computed; 2) monitoring points—not every network element can be easily monitored because nodes may lack a monitoring interface such as the Simple Network Monitoring Protocol (SNMP), or nodes may not support the installation of additional software. Monitoring points also depend on the test objective which can be monitoring network links, monitoring network queue dynamics, or monitoring the traffic generated by specific server applications; 3) monitoring duration—it must be large enough to enable the gathering of statistically sound results, what implies that a relevant number of events must be captured; this duration may be difficult to define in the passive approach, since the rate of relevant events is, a priori, unknown; 4) synchronization—we may be interested in monitoring a sequence of events that might be difficult to synchronize in scenarios involving several nodes; 5) transparency—because network monitoring often uses the same resources to transmit regular traffic and monitoring control traffic, we may say that monitoring may affect the results.

In network simulation environments, network monitoring is used mostly for characterizing the performance of network protocols. Monitoring in a simulation environment differs from monitoring in real networks in a set of aspects: a) active monitoring is implicitly employed since the traffic injected in a simulated network is controlled and defined statistically; b) installing probing functions to inspect the traffic and queue dynamics is feasible and easy, and there is no need to use monitoring protocols such as SNMP, since the results are easily gathered as data resides in the same process; c) monitoring is less intrusive because the monitoring data needs not to traverse and use the network resources; d) events are easily synchronized because network simulators use the same simulation timer and scheduler; e) scenarios of lots of nodes can be easily addressed.

Network monitoring in simulated environments do present some problems: the simulation models may not be accurately designed and produce results that may diverge from the actual protocols. Gathering of results requires the researcher to develop a great deal of code and possibly to

know and use different scripting languages. This may be aggravated by the unwillingness of the researcher to dedicate more effort to programming tasks, rather than focusing on the research. This may also lead to lower quality simulation models.

This paper introduces the FlowMonitor, a network monitoring framework for the Network Simulator 3 (NS-3) which can be easily used to collect and store network performance data from a NS-3 simulation. The main goals behind the FlowMonitor development are to automate most of the tasks of dealing with results gathering in a simple way, to be easily extended, and to be efficient in the consumption of memory and CPU resources. NS-3 is a state of the art Network Simulation tool that eventually will replace NS-2, a network simulator used by many researchers.

The remainder of this paper is organized as follows: Section 2 gives an overview of the related work; then, in Section 3 an introduction to NS-3 is presented; Section 4 details the FlowMonitor NS-3 module, including its requirements and its architecture; Section 5 presents the validation and results; Section 6 provides the conclusions and future work.

2. RELATED WORK

There are a number of contributions that use trace files to gather information to produce network simulation statistics. Trace graph[8] is a NS-2 trace file analyzer based on Matlab that is able to process almost every type of NS2 trace file format; it produces a large amount of graphics that provide various views on the network such as throughput, delay, jitter and packet losses. In[1], a tracing framework is proposed, called XAV, which was built upon an XML Database Management System (DBMS). It was introduced to avoid storing redundant information, to speed up trace analysis and to simplify the access to trace files by post-processing tools. It requires simulators to produce XAV XML trace format files. These files are imported into the XML DBMS which provides an XQuery interface, an SQL-like querying mechanism. The authors compared the performance of XAV with an equivalent system using flat trace files and AWK for trace parsing, and the results show that XAV is always faster in situations where the user needs to extract non-multiple non-consecutive records.

The use of trace files can be optimized to a certain extent, but this approach usually requires a huge amount of disk space and the simulation performance is degraded by the extensive use of I/O operations. Even if sampling techniques are used, in large simulations the amount of data to store is still significant. Moreover, using samples introduces some precision error. In order to avoid the overhead of using trace files, statistics can be calculated during the simulation runs. In [5], an experiment was conducted which concluded that using trace files can make simulations take up to 6 or 7 times longer when compared to an approach where statistics are gathered during the simulation. The authors proposed a framework for NS-2 that integrates a data collection module with a separated graphical statistic analysis tool that reads result files produced by the former. The data collection module consists of a static C++ class named Stat. The Stat class provides a put method that can be called anywhere in the NS-2 code. It can collect abstract

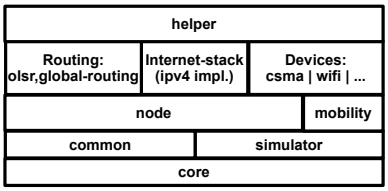

Figure 1: Overview or the main NS-3 modules

metrics, enabling the use of the framework to measure any variable needed by the user. It is also possible to select specific flows as targets for monitoring instead of considering all the flows. The collected data is used to calculate either mean values or probability density functions. The statistical analysis tool, named analyzer, uses a configuration file which defines the minimum and maximum number of simulation runs required and the specific metrics the user is interested in analyzing. The number of simulation runs can be defined by setting a desired confidence interval. The analyzer can be configured through command line or using a GUI developed with C++/GTK+. The generic nature of this framework provides enhanced flexibility, but at the cost of simplicity. It requires the programmer to explicitly include calls to Stat::put method in his code. Although the framework includes code to automate calls for the most common types of metrics, it still can require some integration effort in some cases. Its use is limited to NS-2. Porting the framework to NS-3 is not trivial due to the many differences between the two versions.

3. NS-3 INTRODUCTION

Network Simulator version 3, NS-3, is a new simulator that is intended to eventually replace the aging NS-2 simulator. NS-3 officially started around mid 2006, and the first stable version was released in June 2008, containing models for TCP/IP, WiFi, OLSR, CSMA (Ethernet), and point-to-point links, "GOD routing", among others. Additional stable versions have been subsequently released, including Python bindings, learning bridge, and real-time scheduler for version 3.2 (Sep. 2008), emulation, ICMP, and IPv6 addresses in NS-3.3 (Dec. 2008), WiFi improvements, object naming system, and "tap bridge" in NS-3.4 (Apr. 2009). Even though NS-2 still has a greater number of models included in the distribution, NS-3 has a good development momentum and is believed to have a better core architecture, better suited to receive community contributions. Core architecture features such as a COM-like interface aggregation and query model, automatic memory management, callback objects, and realistic packets, make for a healthier environment in which to develop new complex simulation models. In addition, it is reportedly[10] one of the better performing simulation tools available today.

NS-3 is organized as a set of modules, as shown in Fig. 1. The "core" module provides additional C++ syntactic sugar to make programming easier, such as smart pointers[6], rich dynamic type system, COM-like[2] interface query system, callback objects, tracing, runtime described object attributes, among others.

One of the unusual characteristics about NS-3 when com-

pared to other network simulators is its tracing architecture. Generally, tracing is a facility provided by a simulator by which the user can discover which significant events are happening inside the simulation and under which conditions. Tracing will allow the researcher to derive important metrics of a simulation that can be used to quantify the value of a simulated model relative to another module. In NS-2, as in most simulators, tracing consists in generating a text file describing a series of events, with associated time stamps and other properties, one event per line. Common events that are recorded to such files include MAC layer transmission, reception, and queue packet drop, among others. In NS-3, output of events to a text file is also provided, for a small subset of the events known by NS-3. Another possible way for storing events, in particular packet transmit/receive events, is via a PCAP files[1].

However, these are just alternative tracing systems; the main tracing system provided by NS-3 is callback based tracing. In NS-3 callback based tracing, *trace sources* are defined by NS-3 itself. Each possible trace source in NS-3 is associated with a specific object class and is identified by a name. The programmer may register a C++ function or method to be called when a certain (or a set of) trace source produces a new event. It is then the responsibility of the programmer to know what to do in the callback. Common uses for tracing include 1) writing raw event data to a file, 2) collect statistics for the occurrence of the event so that only the mean or other statistic moment is saved to a file, and 3) react to the event and change some parameter in the simulation in real time, for instance to experiment with cross-layer[4] optimizations.

Other modules in NS-3 include "common", containing data types related to the manipulation of packets and headers, and the "simulator" module, containing time manipulation primitives and the event scheduler. The "node" module sits conceptually above the previous modules and provides many fundamental features in a network simulator, such as a Node class, an abstract base class for a layer-2 interface (NetDevice), several address types, including IPv4/6 and MAC-48 (EUI-48 in IEEE 802 terminology) address classes, and abstract base classes for a TCP/IP stack. The "mobility" module contains an abstract base class for mobility models. A MobilityModel object may be aggregated with a Node object to provide the node with the ability to know its own position. Certain NetDevice implementations, such as WiFiNetDevice, need to know the physical position of a node in order to calculate interference and attenuation values for each transmitted packet, while others, like PointToPoint, do not need this information at all. Some common mobility models are included in NS-3, such as *static*, *constant velocity*, *constant acceleration*, *random walk*, *random waypoint*, and *random direction*[3].

NS-3 also contains a couple of routing models, "olsr" and "global-routing", a module "internet-stack" implementing a UDP/TCP/IPv4 stack, and few NetDevice implementations, including WiFi (infrastructure and adhoc 802.11), CSMA (Ethernet-like), and PointToPoint (very simple PPP-like link). Finally, sitting above all these modules is a "helper" module. This module provides a set of very simple C++ classes that do not use pointers (smart or otherwise) and wrap the existing lower level classes with a more programmer-friendly interface. In design pattern terminology[7], we might call this a *façade*.

[1]PCAP is a binary format for storing (usually live captured) packets, used by programs such as wireshark and tcpdump.

4. THE FLOWMONITOR NS-3 MODULE

4.1 Requirements

When designing the flow monitoring framework, FlowMonitor, a set of goals were taken into consideration, covering aspects such as usability, performance goals, and flexibility.

First and foremost the monitoring framework should be as easy to use as possible. Simulation is already very hard work, and researchers need to focus more on their research rather than spend time programming the simulator. The flow monitoring framework must be easy to activate with just a few lines of code. Ideally, the user should not have to configure any scenario-specific parameters, such as list of flows (e.g. via IP address and port src/dest tuples) that will be simulated, since these are likely to change for numerous reasons, including varying simulation script input parameters and random variable seed. The list of flows to measure should itself be detected by the flow monitor in runtime, without programmer intervention, much like the existing "ascii" and "pcap" trace output functions do already in NS-3.

Another important concern is regarding the perfect amount of data to capture. Clearly, too little data can be risky for a researcher. A complex series of simulations, including varying input parameters, and multiple runs for generating good confidence intervals, can take between a few minutes to a few days to complete. It can be very frustrating to wait for simulation results for days only to discover in the end that there was *something* that we forgot to measure and which is important, causing the researcher to code in the additional parameter measurement and wait a few more days for new simulations to be run. Ideally, the simulation framework should attempt to include a reasonably complete information set, even though most of the information may not be needed most of the time, as long as it does not consume too much memory. A large data set is also useful because, in this way, the researcher is able to run the same simulations once, but analyze the results multiple times using multiple views. The reverse is also true. We do not want to save too much information regarding flows. Too much information is difficult to store and transmit, or can significantly increase the memory footprint of the simulation process. For instance, it is preferable to use histograms whenever possible rather than per-packet data, since per-packet data does not scale well with the simulation time. Histograms, on the other hand, do not grow significantly with the number of packets, only with the number of flows, while still being very useful for determining reasonably good approximations of many statistical parameters.

It is also a goal of this framework that the produced data can be easily processed in order to obtain the final results, such as plots and high-level statistics. As mentioned earlier, researchers usually need to analyze the same data multiple times for the same simulations, which means that this data should end up on a persistent storage medium eventually. Prime candidates for storage are 1) binary files (e.g. HDF),

2) ASCII traces, 3) XML files, and 4) SQL database. It is not completely clear which one of these storage mediums is the best, since each one has its drawbacks. Binary files, for instance, can store information efficiently and allow fast data reading, but are difficult to programmatically read/write, and difficult to extend to accommodate new information once the format has been defined, jeopardizing future extensibility. ASCII traces (line-by-line textual representation of data) are verbose (high formatting overhead), difficult to extend, and potentially slow to read. XML files have excellent extensibility traits, but are also verbose, slow to read for large datasets, and requires everything to be read into memory before any data filtering can be performed. SQL databases, on the other hand, are very efficient reading and filtering data without requiring much process memory, but can also be difficult to manage (except file embedded ones, like SQLite), difficult to extend with new information, and more difficult than textual files to find out how to read the data, since the data format can only be discovered by reading the documentation of the software that produced it or by using a database access GUI tool.

We have opted to, initially, support only XML for data storage output, as well as provide access to in-memory data structures. Since the FlowMonitor collects reasonably summarized information, it is not expected that XML trees will be very large, and reading such XML trees into memory is not a problem with today's computing resources. XML presents an important advantage over any other format, which is the large set of programming libraries for reading XML, for almost every programming language, and almost any platform. Especially in scripting languages, such as Python, reading XML is relatively straightforward, and requires no additional programming language, such as SQL, is required knowledge by the researcher. However, this issue is highly debatable, and so all the FlowMonitor data structures are made available for those who wish to serialize data into another storage format.

Support for Python based simulations is also one of the main design goals. At the time of this writing, the NS-3 Python bindings lack support for connecting callbacks to trace sources (`Config::Connect`, `Object::TraceConnect` and related APIs). Although supporting trace source callbacks in Python is desired and planned, the main reason for the lack of interest in implementing this feature stems from the awareness that Python is a very slow scripting language and that using Python for per-packet tracing operations would just massively slow down the simulations to the point of not being practical. The reasons for this slowdown include the need to, on a per-call basis, acquire the Python GIL (Global Interpreter Lock), convert the C++ parameters into Python format, call the Python code, convert the return values from Python into C++ format, and finally release the GIL. In order to effectively collect data for Python based simulations we need a "configure and forget" approach, wherein a C++ class is made responsible for the actual tracing and reports back to Python just the final results, at the end of the simulation.

The FlowMonitor architecture is also designed with extensibility in mind. One use case of simulation is to research next-generation networks, which may not even use IPv4,

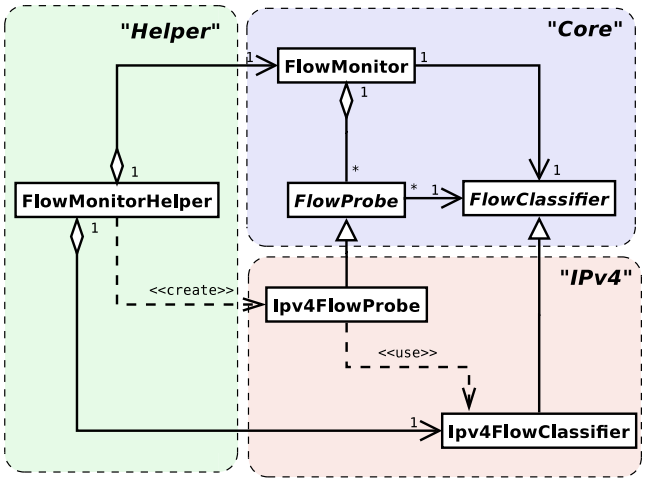

Figure 2: High level view of the FlowMonitor architecture

or IPv6. For instance, a researcher could be simulating an MPLS switching fabric, whose data plane encapsulates 802.3 frames, so we need extensibility at the packet acquisition level to accommodate different data planes. Alternatively, the concept of "flow" may differ from the usual *five-tuple* (source-ip, dest-ip, protocol, source-port, dest-port). For instance, someone may want to classify flows by a "flow label" or DSCP IPv4 field. Therefore we also need an extensible flow classification mechanism.

Finally, there is the obvious, but ever so important, requirement of low monitoring overhead. While it is true that some memory and computation overhead cannot be avoided when monitoring flows, this overhead should be as low as can be reasonably expected.

4.2 Architecture Overview

The FlowMonitor module is organized into three groups of classes, as shown in Fig. 2. The group of "core" classes comprises the *FlowMonitor* class, *FlowProbe*, and *FlowClassifier*. The FlowMonitor class is responsible for coordinating efforts regarding probes, and collects end-to-end flow statistics. The FlowProbe class is responsible for listening for packet events in a specific point of the simulated space, report those events to the global FlowMonitor, and collect its own flow statistics regarding only the packets that pass through that probe. Finally, the class FlowClassifier provides a method to translate raw packet data into abstract "flow identifier" and "packet identifier" parameters. These identifiers are unsigned 32-bit integers that uniquely identify a flow and a packet within that flow, respectively, for the whole simulation, regardless of the point in which the packet was captured. These abstract identifiers are used in the communication between FlowProbe and FlowMonitor, and all collected statistics reference only those abstract identifiers in order to keep the core architecture generic and not tied down to any particular flow capture method or classification system.

Another group of classes provides a "default" IPv4 flow monitoring implementation. The classes *Ipv4FlowProbe* and *Ipv4-*

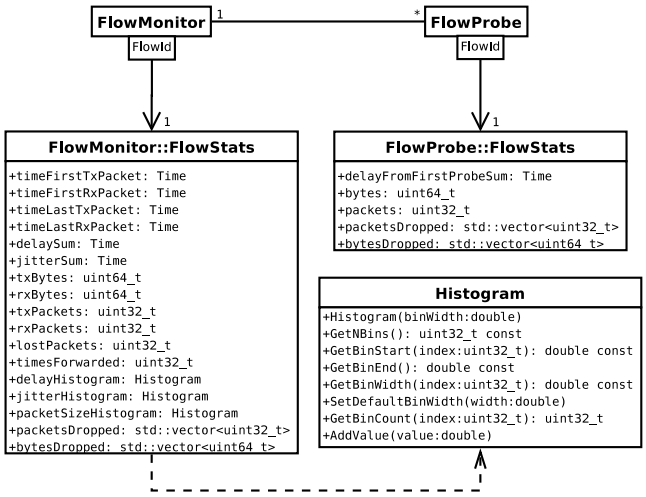

Figure 3: Data collected by the FlowMonitor

Classifier subclass the abstract core base classes FlowProbe and FlowClassifier, respectively. Ipv4FlowClassifier classifies packets by looking at their IP and TCP/UDP headers. From these packet headers, a tuple (source-ip, destination-ip, protocol, source-port, destination-port) is created, and a unique flow identifier is assigned for each different tuple combination. For each node in the simulation, one instance of the class Ipv4FlowProbe is created to monitor that node. Ipv4FlowProbe accomplishes this by connecting callbacks to trace sources in the Ipv4L3Protocol interface of the node. Some improvements were made to these trace sources in order to support the flow monitoring framework.

Finally, there is also a "helper" group consisting of the single class FlowMonitorHelper, which is modelled in the usual fashion of existing NS-3 helper classes. This helper class is designed to make the most common case of monitoring IPv4 flows for a set of nodes extremely simple. It takes care of all the details of creating the single classifier, creating one Ipv4FlowProbe per node, and creating the FlowMonitor instance.

To summarize this highlevel architecture view, a single simulation will typically contain one FlowMonitorHelper instance, one FlowMonitor, one Ipv4FlowClassifier, and several Ipv4FlowProbes, one per Node. Probes capture packets, then ask the classifier to assign identifiers to each packet, and report to the global FlowMonitor abstract flow events, which are finally used for statistical data gathering.

4.3 Flow Data Structures

The main result of the flow monitoring process is the collection of flow statistics. They are kept in memory data structures, and can be retrieved via simple "getter" methods. As seen Fig. 3, there are two distinct flow statistics data structures, *FlowMonitor::FlowStats* and *FlowProbe::FlowStats*. The former contains complete end-to-end flow statistics, while the latter contains only a small subset of statistics and from the point of view of each probe.

In order to understand the main utility of FlowProbe statistics, consider a simple three-node network, $A \leftrightarrow B \leftrightarrow C$. Consequently, we will have[2] three probes, P_A, P_B, and P_C. When a packet is transmitted from A to C passing through B, the probe P_A will notice the packet and create a FlowProbe::FlowStats structure for the flow, storing a $delayFromFirstProbeSum$ value of zero. Next, the probe P_B will detect the packet being forwarded, and will increment the value of $delayFromFirstProbeSum$ in its own FlowStats structure by the transmission delay from A to B. Finally, the packet arrives at C and will be detected by P_C, which then adds to its $delayFromFirstProbeSum$ the delay between A and C. In the end, we are able to extract not only end-to-end mean delay but also partial delays that the packet experiences along the path. This type of probe-specific information can be very helpful in ascertaining what part of the network is responsible for the majority of the delay, for instance. Such level of detail is missing from the FlowMonitor::FlowStats structure alone.

What follows is a more detailed description of the individual attributes in the flow data structures. In FlowMonitor::FlowStats, the following attributes can be found:

timeFirstTxPacket Contains the absolute time when the first packet in the flow was transmitted, i.e. the time when the flow transmission starts;

timeLastTxPacket Contains the absolute time when the last packet in the flow was transmitted, i.e. the time when the flow transmission ends;

timeFirstRxPacket Contains the absolute time when the first packet in the flow was received by an end node, i.e. the time when the flow reception starts;

timeLastRxPacket Contains the absolute time when the last packet in the flow was received, i.e. the time when the flow reception ends;

delaySum Contains the sum of all end-to-end delays for all received packets of the flow;

jitterSum Contains the sum of all end-to-end delay jitter (delay variation) values for all received packets of the flow. Here we define *jitter* of a packet as the delay variation relatively to the last packet of the stream, i.e. $Jitter\{P_N\} = |Delay\{P_N\} - Delay\{P_{N-1}\}|$. This definition is in accordance with the Type-P-One-way-ipdv as defined in IETF RFC 3393;

txBytes, txPackets Total number of transmitted bytes and packets, respectively, for the flow;

rxBytes, rxPackets Total number of received bytes and packets, respectively, for the flow;

lostPackets Total number of packets that are assumed to be lost, i.e. those that were transmitted but have not been reportedly received or forwarded for a long time. By default, packets missing for a period of over 10 seconds are assumed to be lost, although this value can be easily configured in runtime;

[2]Note: in the abstract base architecture it is not implied that there is one probe per node; however, for the sake of this example we will assume the IPv4 flow monitoring case, which does make such assumption.

timesForwarded Contains the number of times a packet has been reportedly forwarded, summed for all packets in the flow;

delayHistogram, jitterHistogram, packetSizeHistogram Histogram versions for the delay, jitter, and packet sizes, respectively;

packetsDropped, bytesDropped These attributes also track the number of lost packets and bytes, but discriminates the losses by a *reason code*. This reason code is usually an enumeration defined by the concrete FlowProbe class, and for each reason code there may be a vector entry indexed by that code and whose value is the number of packets or bytes lost due to this reason. For instance, in the Ipv4FlowProbe case the following reasons are currently defined: `DROP_NO_ROUTE` (no IPv4 route found for a packet), `DROP_TTL_EXPIRE` (a packet was dropped due to an IPv4 TTL field decremented and reaching zero), and `DROP_BAD_CHECKSUM` (a packet had bad IPv4 header checksum and had to be dropped).

Some interesting metrics can be derived from the above attributes. For instance:

mean delay: $\overline{delay} = \frac{delaySum}{rxPackets}$

mean jitter: $\overline{jitter} = \frac{jitterSum}{rxPackets-1}$

mean transmitted packet size (byte): $\overline{S_{tx}} = \frac{txBytes}{txPackets}$

mean received packet size (byte): $\overline{S_{rx}} = \frac{rxBytes}{rxPackets}$

mean transmitted bitrate (bit/s):

$$\overline{B_{tx}} = \frac{8 \cdot txBytes}{timeLastTxPacket - timeFirstTxPacket}$$

mean received bitrate (bit/s):

$$\overline{B_{rx}} = \frac{8 \cdot rxBytes}{timeLastRxPacket - timeFirstRxPacket}$$

mean hop count: $\overline{hopcount} = 1 + \frac{timesForwarded}{rxPackets}$

packet loss ratio: $q = \frac{lostPackets}{rxPackets+lostPackets}$

Some of the metrics, such as delay, jitter, and packet size, are too important to be summarized just by the sum, count and, indirectly, mean values. However, storing all individual samples for those metrics does not scale well and is too expensive in terms of memory/storage. Therefore, histograms are used instead, as a compromise solution that consumes limited memory but is rich enough to allow computation of reasonable approximations of important statistical properties. In FlowMonitor, the class Histogram is used to implement histograms. It offers a single method to count new samples, AddValue, a method to configure the bin width, SetBinWidth, and methods to retrieve the histogram data: GetNBins, GetBinStart, GetBinEnd, GetBinWidth, GetBinCount. From this data, estimated values for N (number of samples), μ (mean), and s (standard error) can be easily computed. From the equations found in [9] (Chapter 2), we can derive:

$$N = \sum_{i=0}^{M-1} H_i$$

$$\mu = \frac{1}{N} \sum_{i=0}^{M-1} C_i H_i$$

$$s^2 = \frac{1}{N-1} \sum_{i=0}^{M-1} (C_i - \mu)^2 H_i$$

In the above equations, M represents the number of bins, H_i represents the count of bin i, and C_i the center value of bin i.

In FlowProbe::FlowStats some additional attributes can be found on a per-probe/flow basis:

delayFromFirstProbeSum Tracks the sum of all delays the packet experienced since being reportedly transmitted. The value is always relative to the time the value was initially detected by the first probe;

bytes, packets number of bytes and packets, respectively, that this probe has detected belonging to the flow. No distinction is made here between first transmission, forwarding, and reception events;

bytesDropped, packetsDropped tracks bytes and packets lost qualified by reason code, similarly to the attributes with the same name in FlowMonitor::FlowStats.

4.4 Example

To demonstrate the programming interfaces for using FlowMonitor, we create a simple simulation scenario, illustrated in Fig. 4, with a grid topology of 3×3 WiFi adhoc nodes running the OLSR protocol. The nodes transmit CBR UDP flows with transmitted bitrate of 100 kbit/s (application data, excluding UPD/IP/MAC overheads), following a simple node numbering strategy: for i in 0..8, node N_i transmits to node N_{8-i}.

It is out of scope of this paper to present the full example program source code[3]. Suffice to say that enabling the flow monitor is just the matter of replacing the line `ns3.Simulator.Run()`, with something like this (using the Python language):

```
flowmon_helper = ns3.FlowMonitorHelper()
monitor = flowmon_helper.InstallAll()
monitor.SetAttribute("DelayBinWidth",
                     ns3.DoubleValue(0.001))
monitor.SetAttribute("JitterBinWidth",
                     ns3.DoubleValue(0.001))
monitor.SetAttribute("PacketSizeBinWidth",
                     ns3.DoubleValue(20))

ns3.Simulator.Run()

monitor.SerializeToXmlFile("results.xml", True, True)
```

[3]The complete source code is available online, at `http://code.nsnam.org/gjc/ns-3-flowmon`

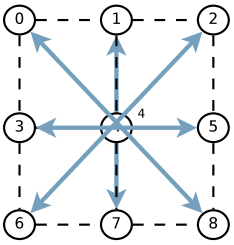

Figure 4: Example network topology

What the above code does is:

1. Create a new FlowMonitorHelper object;
2. Call the method InstallAll on this object. As a result, the flow monitor will be created and configured to monitor IPv4 in all the simulation nodes;
3. Configure some histogram attributes;
4. Run the simulation, as before, calling ns3.Simulator.Run();
5. Finally, write the flow monitored results to a XML file named "results.xml". The second parameter of SerializeToXmlFile indicates if we wish to also save the histogram contents, and the third parameter indicates if we wish to save per-probe flow statistics.

To present the results, one would have to write a program that reads the data from XML and creates plots. For instance, the Python program in List. 1 reads the XML file and plots the histograms of 1) received bitrates, 2) packet losses, and 3) delays, for all flows. The program uses the Python XML parsing module *ElementTree* for reading the XML file, and the *matplotlib* module for generating the plots. The results in Fig. 5 are obtained from this and show that most flows achieved an actual throughput of around 105 kbit/s, except for one flow that only transferred less than 86 kbit/s. Packet losses were generally low except for two flows. Finally, mean delays vary between ≈ 20 ms and ≈ 70 ms.

5. VALIDATION AND RESULTS

For validation and obtaining results, we begin by describing a simple network scenario, which is then used for validation purposes. Finally performance is evaluated using the same scenario and varying the network size.

5.1 Test scenario

Fig. 6 shows the network topology and flow configuration for the test scenario. It consists of a number of rows of nodes, each row containing a number of nodes, with each node connected to the next one via a point-to-point link. From left to right, a link is configured with 100 kbit/s, then the next link is configured with 50 kbit/s, the next with 100 kbit/s again, and so on. The configured delay is zero, and the drop tail queue maximum size is 100 packets. The rows are not vertically connected. The NS-3 GlobalRoutingManager is used to compute the routing tables at the beginning of the

Listing 1: Sample script to read and plot the results

```
et = ElementTree.parse(sys.argv[1])
bitrates = []
losses = []
delays = []
for flow in et.findall("FlowStats/Flow"):
    # filter out OLSR
    for tpl in et.findall("Ipv4FlowClassifier/Flow"):
        if tpl.get('flowId') == flow.get('flowId'):
            break
    if tpl.get("destinationPort") == '698':
        continue

    losses.append(int(flow.get('lostPackets')))

    rxPackets = int(flow.get('rxPackets'))
    if rxPackets == 0:
        bitrates.append(0)
    else:
        t0 = long(flow.get('timeFirstRxPacket')[:-2])
        t1 = long(flow.get('timeLastRxPacket')[:-2])
        duration = (t1 - t0)*1e-9
        bitrates.append(8*long(flow.get('rxBytes'))
                        / duration * 1e-3)

        delays.append(float(flow.get('delaySum')[:-2])
                      * 1e-9 / rxPackets)

pylab.subplot(311)
pylab.hist(bitrates, bins=40)
pylab.xlabel("Flow bitrate (bit/s)")
pylab.ylabel("Number of flows")

pylab.subplot(312)
pylab.hist(losses, bins=40)
pylab.xlabel("Number of lost packets")
pylab.ylabel("Number of flows")

pylab.subplot(313)
pylab.hist(delays, bins=10)
pylab.xlabel("Delay (s)")
pylab.ylabel("Number of flows")

pylab.subplots_adjust(hspace=0.4)
pylab.savefig("results.pdf")
```

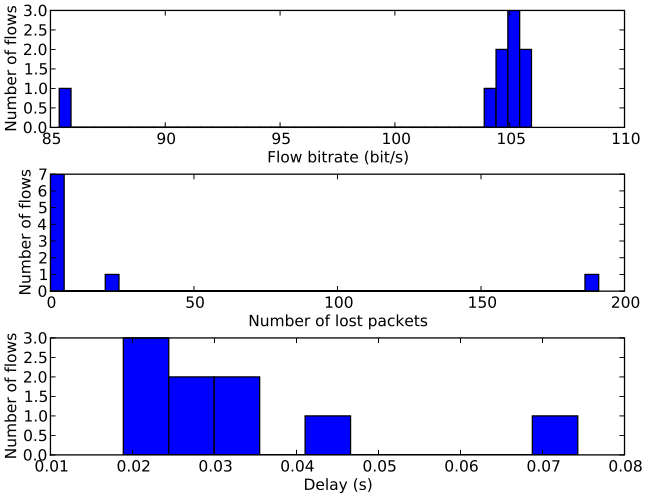

Figure 5: Example program results

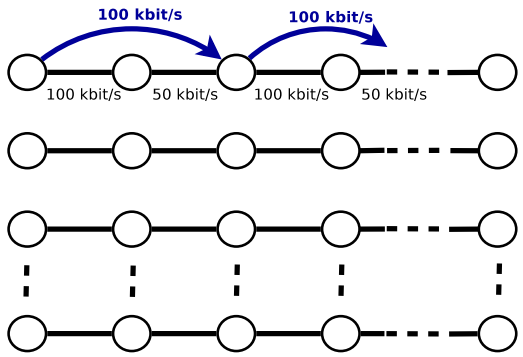

Figure 6: Test scenario

simulation. Flows are created at the beginning of the simulation. Every other node sends one 100 kbit/s UDP flow[4] to the node that is two hops to the right, and sends another flow to the node that is two hops to the left. Packet size is the default 512 byte UDP payload.

5.2 Validation

To validate the flow monitor, we begin by examining the test scenario and deriving some theoretical flow metrics, which are then compared to values obtained by the measurements done with the help of the FlowMonitor module.

The test topology is relatively simple to analyze. In the first link, between the first and second nodes, we have one 100 kbit/s UDP flow in each direction. For each flow, the first hop of the flow traverses a link whose capacity matches exactly the flow link-layer bitrate. Thus, packet loss rate will be zero, and queueing delay will also be null. However, when the flow reaches the second hop it will have to be transmitted by a link that is half the capacity of the flow bitrate. Consequently, after a few seconds the drop-tail queue will fill to maximum capacity, causing a queueing delay, and half the packets will have to be dropped due to the bottleneck link.

We can derive the estimated values for delays, losses, and bitrates of each flow. We define S as the number of bits in each packet, $S = (512 + 20 + 8 + 2) \times 8 = 4336$ bit, and C_1 the bitrate of the link layer of the 100 kbit/s link, $C_1 = 100000$ bit/s. Then, the delay in the first hop, where there is no queueing, is simply the transmission delay, $d_1 = \frac{S}{C_1} = 0.04336$ s. In steady state, the second hop drop-tail queue will be filled, and so packets will experience a delay corresponding to transmitting 99 packets ahead in the queue, plus the packet itself, plus the packet that the PPP device is currently transmitting. Since the second hop has lower bitrate, $C_2 = 50000$ bit/s, and so $d_2 = 101 \times \frac{S}{C_2} = 8.75872$ s. Thus, the total end-to-end delay experienced by the flow will be $d_1 + d_2 = 8.80208$ s. Regarding packet losses, each flow traverses two hops. As previously explained, packet drop probabilities will be 0 and 0.5 for the first and second hop, respectively. Thus, the end-to-end packet probability will be 0.5 for each flow. Consequently, the received bitrate should be half the transmitted bitrate.

[4]Actually, at application layer the flow bitrates are 94.465 kbit/s, so that with UDP, IPv4, and MAC header overhead the bitrate is exactly 100 kbit/s at link layer.

The validation results in Tab. 1 show that, for a scenario with 2704 nodes (i.e. 1352 flows), the measured results (for 10 simulations) match the theoretical values within a very small margin of error. The expected values for transmitted bitrate is slightly less than 100 kbit/s due to the translation from layer-2 to layer-3, taking into account the factor $\frac{512+20+8}{512+20+8+2}$ due to the PPP header not being present where packets are monitored by the FlowMonitor. Same reasoning applies to the received bitrate. The errors found between estimated and measured values are negligible for transmitted/received bitrates and delays, and are likely a result of sampling issues and/or numeric errors. The error in the packet loss ratio is slightly larger, but this error is not of great significance. This error can be explained by the method of measuring losses used by the FlowMonitor, wherein a packet is considered lost if it has not been reported by any probe to have been received or retransmitted for a certain period of time, i.e. only packets that are "missing in action" are considered lost. Naturally, at the end of simulation a number of packets are still in transit, some of which could have been lost, but the potential packet losses are not accounted for, hence the error. In Sec. 6.1 we mention a possible way to overcome this error, as future work.

5.3 Performance Results

To evaluate the overhead introduced by flow monitoring, we ran a series of simulations, increasing the network size between 16 and 2704 nodes, and measuring the time taken to simulate each scenario, and memory consumption (virtual memory size), 1) without collecting any results, 2) with flow monitoring, and 3) with ascii tracing to a file. We repeat each experiment 10 times with different random variable seeds. The performance results in Fig. 7 show the additional overhead, in terms of memory consumption and simulation wall-clock time, that is incurred by enabling the flow monitor or trace file. The top-left plot shows the total memory consumed by a simulation while varying the number of nodes. Three curves are shown: one represents the simulations without FlowMonitor or file tracing enabled, another other one represents the same simulations but with FlowMonitor enabled, and the remaining curve represents the simulations with ascii file tracing. The top-right plot shows an alternative view of the same information where instead the monitoring/tracing overhead is shown. The monitoring overhead, in percentage, is defined by the formula $100\frac{M_{monitor}-M_{base}}{M_{base}}$, where $M_{monitor}$ is the memory consumption with monitoring enabled, and M_{base} the memory without monitoring. Idem for file tracing. The memory overhead of enabling the FlowMonitor was 23.12% for 2704 nodes, corresponding to 45 MB of extra memory consumption, while the overhead with ascii file tracing was 5.82%. The bottom two plots show the impact in terms of simulation wallclock time of the FlowMonitor and trace file, the left plot showing the three curves separately, the right plot showing the relative overhead (using a formula similar to the memory overhead one). For the simulation time, which in case of FlowMonitor includes the time needed to serialize all data to an XML file, the overhead reaches a peak of about 55% for 500 nodes, but then gradually decreases with the network size until reaching 38.82% (about 90 seconds) for 2704 nodes, while the overhead of ascii trace file generation is al-

Metric	Measured Value (95% C. I.)	Expected Value	Mean Error
Tx. bitrate	$99646.06 \pm 2.68 \times 10^{-5}$	99631.00	+0.015 %
Rx. bitrate	$49832.11 \pm 7.83 \times 10^{-5}$	49815.50	+0.033 %
Delays	$8.8005 \pm 8.8 \times 10^{-9}$	8.80208	-0.018 %
Losses	$0.4978 \pm 1.5 \times 10^{-6}$	0.5000	-0.44 %

Table 1: Validation results

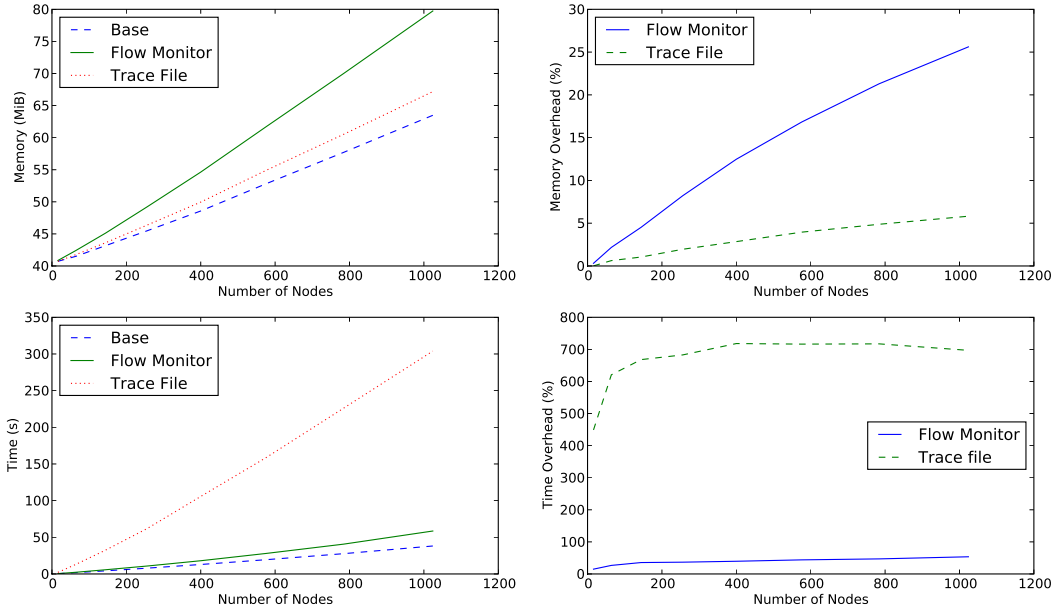

Figure 7: Performance results of the flow monitor

most always above 700%. In the course of all simulations, over 150 GiB of ascii trace files were generated.

It should be noted that these values are worst case scenario. Since the simulation scenario is very simple (PointToPointNetDevice is just about the simplest and fastest NetDevice implementation in NS-3), the additional overhead appears relatively significant. More complex simulations, for example WiFi and OLSR, should consume considerably more memory and CPU time, so that enabling the FlowMonitor on such simulations will add an overhead that will be smaller compared to the baseline simulation.

6. CONCLUSIONS

In this paper we have described a solution that solves a common problem for all researchers that need to conduct simulations in NS-3: how to easily extract flow metrics from arbitrary simulations? Existing solutions, some of which have been identified, do not solve this problem effectively, both for simulators in general but especially in NS-3. A set of general requirements have been identified, and a new flow monitoring solution was designed and implemented which meets those requirements. The simplicity of use of this new framework has been demonstrated via a very simple example. The implementation was validated by comparing measured flow metrics with theoretical results. Performance results show that flow monitoring introduces a relatively small overhead, even when used with a base simulation that is already very efficient to begin with.

6.1 Future Work

Although the FlowMonitor already covers the most important functionality needed from such a system, there is room for improvement. More data output methods, such as database and binary file, would of course be welcome. Another useful addition could be to add more options to better control what level of detail is stored in memory. For instance, we might not be interested in per-probe flow statistics, or in histograms. FlowMonitor support for multicast/broadcast flows is another features that could be useful for certain researchers. The FlowMonitor could benefit from a closer integration with NetDevices, e.g. so that it could directly monitor packet drop events from each NetDevice's transmission queue, as well as handle transmission errors from layer 2, instead of relying on the vaguer "packet missing in action" measurement method. In some circumstances a researcher might want to observe how flow metrics evolve over time, instead of just obtaining a summary of the results over the entire simulation. This would require saving a periodic snapshot of the flows metrics to a file. It can be done already by user code, but a convenience API for it would be interesting to have in FlowMonitor. Finally, we would like to add convenient methods to the Histogram class that would compute the values N, μ, and s that were mentioned in Sec. 4.3.

The FlowMonitor code for NS version 3.2 is available online at `http://code.nsnam.org/gjc/ns-3-flowmon/`. In the future, the authors plan to port this code to the latest version of NS-3 and propose it for inclusion in the main NS-3 source code repository.

7. REFERENCES

[1] R. Ben-El-Kezadri, F. Kamoun, and G. Pujolle. XAV: a fast and flexible tracing framework for network simulation. In *Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 47–53, Vancouver, British Columbia, Canada, 2008. ACM.

[2] D. Box. *Essential COM*. Addison-Wesley, 1998.

[3] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5):483–502, 2002.

[4] G. Carneiro, J. Ruela, and M. Ricardo. Cross-layer design in 4G wireless terminals. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 11(2):7–13, 2004.

[5] C. Cicconetti, E. Mingozzi, and G. Stea. An integrated framework for enabling effective data collection and statistical analysis with ns-2. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 11, Pisa, Italy, 2006. ACM.

[6] D. Edelson. *Smart pointers: They're smart, but they're not pointers*. University of California, Santa Cruz, Computer Research Laboratory, 1992.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[8] J. Malek and K. Nowak. Trace graph-data presentation system for network simulator ns. In *Proceedings of the Information Systems - Concepts, Tools and Applications (ISAT 2003)*, Poland, September 2003.

[9] S. W. Smith. *The Scientist & Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1st edition, 1997.

[10] E. Weingärtner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Proceedings of the IEEE International Conference on Communications 2009 (ICC 2009), Dresden, Germany, IEEE.*, June 2009.