Modular Network Trace Analysis

Wolfgang Kiess Nadine Chmill Ulrich Wittelsbürger
Martin Mauve
Computer Networks Research Group
University of Düsseldorf, Germany
{kiess, chmill, wittelsbuerger, mauve}@cs.uni-duesseldorf.de

ABSTRACT

In this paper we present EDAT, a tool designed for the analysis of trace files from network simulations and experiments. The EDAT framework encapsulates analysis steps in extensible operators. These can be arbitrarily combined to a flow-based analysis. The core of EDAT is a library of these operators implemented as classes in the scripting language Ruby. For ease of use, these can be visually assembled by means of a graphical user interface that also allows to configure and execute these operators. The results can be plotted in graphs that are directly usable in scientific publications.

Categories and Subject Descriptors

C.2 [**Network Architecture and Design**]: Miscellaneous; I.6.6 [**Simulation Output Analysis**]

General Terms

Experimentation, Measurement, Performance, Verification

Keywords

Trace file analysis, wireless networks, ad hoc networks, experiments, simulation, visualization

1. INTRODUCTION

After a real-world experiment or a simulation is completed, it has to be interpreted based on the recorded trace files. The result of such an analysis are graphs usable e.g. for scientific publications. As an example for this process, consider the calculation of the packet delivery ratio between two stations A and B of a protocol X based on data from a real experiment. For this, the information in the two packet traces has to be processed by: 1) parsing the traces, 2) removing packets not sent by A or not belonging to protocol X, 3) counting the number of packets for each time interval in both files, 4) dividing the matching values through each other, and 5) producing a plot of these values.

Obviously this requires a significant, custom-made processing of the input data[1]. Therefore, such analyses are most often performed with custom-made software tools written in scripting languages like Perl, Python, Sed, or Ruby. When such a program is built from scratch, it can be crafted for the current analysis. However, these programs are only a by-product of an examination and their creation can be time consuming. As a result, such analysis programs are often "quick hacks" and lack a good software design, thus reducing maintainability and reusability. Furthermore, every small change in the analysis process requires the manual adaptation of the program source code and the subsequent rerunning of the whole analysis.

After examining our own programs written over the last years for the analysis of simulations and real experiments, it became obvious that a lot of simple as well as complex operations recur in modified form in nearly each program. Therefore, we have developed the *Extensible Data Analysis Toolkit* (EDAT) to encapsulate and reuse these recurring functionalities in so-called *operators.* EDAT treats the data to be analyzed as a flow that runs through a chain of concatenated operators. Each operator in the chain modifies the incoming data stream and hands the result over to the next operator. EDAT itself provides more than 40 operators and can be easily extended within a few minutes by either implementing new operator classes or combining existing operators to more complex ones. The tool can produce graphs in postscript-format, a functionality already used in a number of our own publications [1–3]. For fast analysis development, EDAT features a graphical user interface to combine operators by simple drag-and-drop. EDAT is designed for 1) the rapid development of analyses with 2) detailed control over the whole process of data manipulation with 3) the same power as directly programming it in a scripting language. EDAT has been created for the analysis of network simulations and experiments and the flow-based, sequential processing approach is especially suited to examine packet traces. However, this does not limit the tool to such analyses. As only a certain number of the provided operators (e.g., the parsers) is network-specific and flow-based processing is a general, powerful principle, the tool can be easily extended to also support non-network-specific analyses.

This paper is structured as follows. In Section 2 we review the related work. Section 3 then explains concept, architecture, and user interface of EDAT. Advanced features like caching are presented in Section 4 and the usage of EDAT

[1]According to our own experiences, simulation and experimental data can be treated similarly.

for the analysis outlined in the introduction is presented in Section 5. The conclusion can be found in Section 6.

2. RELATED WORK

The tasks in the post-experiment analysis based on recorded data can be divided into parsing, processing, data modeling/mining, and visualization. After extracting the raw data from a collection of files in the parsing step, this data is processed by combining, filtering, and transforming the information. Data modeling/mining is the application of statistical methods or clustering techniques to discover patterns and dependencies in the data, and visualization consists of producing a two or three-dimensional graphical representation of the result.

The processing on a per-packet basis is supported by network tracers like tcpdump [4] or wireshark [5] in an offline-mode. In this mode, packets in capture files can be filtered, and these tools also provide some basic manipulation of the files' content like cropping or altering packet timestamps. If the data has been parsed and inserted into a relational database, processing can also be performed by means of SQL with operations like joining or averaging. For more complex processing, the network data mining tool CoMo [6] can be used. CoMo manages the recording and storage of raw capture data as a data flow and provides callbacks in this flow. Via these callbacks, the user can insert custom functions written in C to implement the analysis.

The idea to combine generic elements to a processing pipeline can already be found in tools like the Unix shell bash or in dataflow programming languages. Here, we concentrate on tools that use this approach mainly for visualization or data mining. Huginn [7], a 3D visualizer for simulation trace files, allows to combine a predefined set of processing components within a fixed pipeline. The resulting information is used to alter the properties of the displayed network nodes. In the data mining tool KNIME [8], the necessary statistical calculations are created by graphically combining processing components. KNIME also supports simple (pre-)processing of the input data, e.g., via filtering or sorting. For more complex processing however, KNIME would have to be extended with new components. As their creation requires a lot of work [8] this may be too time consuming for an analysis process requiring such extensions on a regular basis. Such custom extensions would also be necessary for mathematical computation frameworks like Matlab [9] or Maple [10] which are designed for matrix manipulation or symbolic and numerical computations. OpenDX [11] is a data visualization tool for 2D or 3D plotting and also allows the production of animations. It provides a visual program editor to configure the plotting with predefined, connectable components. LabView [12] is a commercial visual programming language mainly used to build measurement and control applications. The analyses provided by LabView concentrate on the description of physical phenomena, e.g., by means of signal or image processing or wavelet transform.

Instead of performing custom analyses, it is also possible to use standard analysis. One example here are the metrics, e.g. for connectivity, loss, or delay defined by the IETF IPPM WG [13]. Another example is the trace file analyzer TraceGraph [14] that defines over 200 different standard evaluations for files produced by the network simulator ns-2. If the data has already been processed, plotting can be performed with gnuplot [15]. It requires an input file in a table style format and some configuration parameters and is able to plot the corresponding graph in a large number of output formats.

3. PHILOSOPHY, ARCHITECTURE AND IMPLEMENTATION

An examination of custom-made, handwritten programs for the analysis of network simulations and traces from real experiments shows that these programs share a lot of similar functionality. Instead of implementing each of these recurring operations from scratch upon design of a new analysis, EDAT provides a framework to encapsulate them in so called *operators*. These operators can be combined to form a data processing pipeline where the data is handed from one processing element to the next and successively transformed in each step. To adapt to the processing needs, each operator can be configured by means of certain parameters. With this approach, an analysis is a concatenated sequence of operations on the input data.

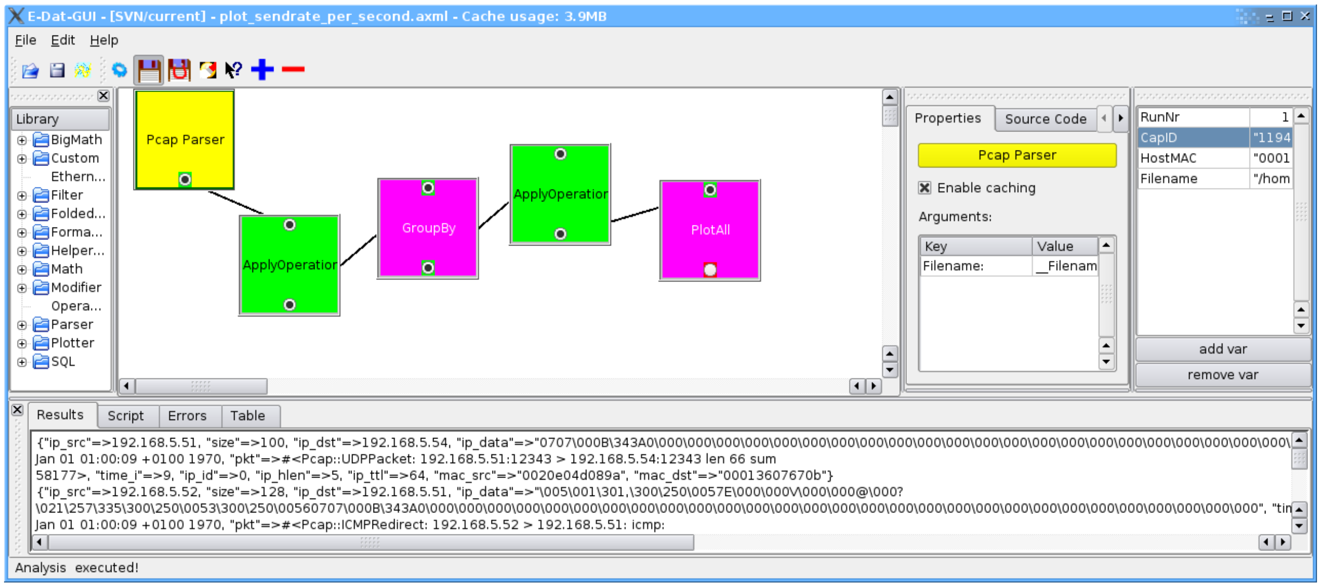
3.1 Graphical User Interface

As shown in Figure 1, a data processing pipeline is created with the EDAT *graphical user interface* (GUI). The GUI provides boxes as visual representations of the operators that can be added and combined by simple drag and drop. The user interface is divided into four main areas, the operator library on the left, the workbench in the center, the operator inspector on the right and a result and feedback view at the bottom. Once an operator is on the workbench, selecting it shows its current configuration in the inspector. To integrate an operator into the analysis, its input and output ports must be connected to other operators. Existing connections are represented by lines drawn from an output port at the bottom of one box to an input port at the top of another box. Each box provides a context menu to delete the operator or view and modify the operator's source code. The "execute" entry in this menu triggers the processing of this operator. The result is then shown in the text box at the bottom, allowing for an inspection of the data in the flow.

A noteworthy feature of the EDAT GUI is *operator folding* that allows to create a new operator by combining other operators. As a simple example, imagine that an analysis of all packets with a size between 500 and 1 000 bytes should be performed. To extract these packets, two consecutive filters can be inserted in the data flow, the first filtering out all packets below 500 bytes while the second discards all packets above 1 000 bytes. However, if such a range filter is needed in another analysis, the same two filters would have to be configured once again. Instead, these filters can be combined to a new operator: once they have been selected on the workbench, their context menus provide the "Create compound operator" option. This inserts a new operator into the library that internally uses the two filters to perform its tasks. In order to create a generic range filter, special tags can be used as configuration parameters. Each of these tags is used as argument of the newly created range filter and handed over as configuration option to the internal filter operators.

3.2 Operators and their Data Format

Under the hood, an EDAT operator is implemented as class in the scripting language Ruby. The input ports are



[✓ Enable caching]

Arguments:

Key	Value
Filename:	_Filenam

RunNr	1
CapID	"1194
HostMAC	"0001
Filename	"/hom

```
{"ip_src"=>192.168.5.51, "size"=>100, "ip_dst"=>192.168.5.54, "ip_data"=>"0707\000B\343A0\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\
Jan 01 01:00:09 +0100 1970, "pkt"=>#<Pcap::UDPPacket: 192.168.5.51:12343 > 192.168.5.54:12343 len 66 sum
58177>, "time_i"=>9, "ip_id"=>0, "ip_hlen"=>5, "ip_ttl"=>64, "mac_src"=>"0020e04d089a", "mac_dst"=>"00013607670b"}
{"ip_src"=>192.168.5.52, "size"=>128, "ip_dst"=>192.168.5.51, "ip_data"=>"\005\001\301,\300\250\0057E\000\000V\000\000@\000?
\021\257\335\300\250\0053\300\250\00560707\000B\343A0\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000", "tin
Jan 01 01:00:09 +0100 1970, "pkt"=>#<Pcap::ICMPRedirect: 192.168.5.52 > 192.168.5.51: icmp:
```

Analysis executed!

Figure 1: Screenshot of the EDAT GUI. The analysis shown on the workbench produces a plot of the number of packets a node has sent per second.

realized as references to other operator objects, and the remaining configuration is performed via additional constructor parameters. Between operators, data is exchanged in generic data containers. The format of these containers is related to the tables of relational databases but allows for "rows" with arbitrary structure. To this end, each row is implemented as an associative array of key-value pairs, and all rows together are stored in an array that preserves the input order. The processing in an operator is performed in three steps: 1) get the container with the output of the preceding operator; 2) modify the data and put the result in another container; 3) return the result container to the subsequent operator.

This mechanism is best explained by means of an example: consider an experiment where packets are transmitted over a network with lossy links. The goal of the analysis is to calculate the length of the occurring error bursts based on the receiver packet trace. The corresponding EDAT parser for this trace format produces a data container in which each row corresponds to one packet. The different header fields are stored as key-value pairs. A row from this container looks as follows:

```
{"ip_src"=>192.168.5.50,
"time"=>Wed Jul 25 09:03:02 +0200 2007,
"ip_dst"=>192.168.5.255, "id"=>46, ...}
```

To compute the length of the error bursts, we use the `CalculateInterrowDifference` operator. It calculates the difference of the same field between each consecutive pair of rows. The idea is to compute this for the packet-id field: if the id of consecutive packets (rows) differs by n, $n-1$ packets are missing. The processing is performed in the `process` method: by calling the `getResult` method of the preceding operator,this operators' `process` method is triggered that returns the input data container. The calculation of the difference is performed in a loop over all rows in this container. The result is stored in a new container and returned at the end of the method.

With the correct configuration, the operator `Calculate-InterrowDifference` transforms the input data into

```
{"ip_src"=>192.168.5.50, ...,
 "id"=>46, "id_delta"=>36, ...}
{"ip_src"=>192.168.5.50, ...,
 "id"=>54, "id_delta"=>8, ...}
```

where the new field `id_delta` contains the difference between consecutive ids. In the current example this difference is eight, thus a total of seven packets have been lost between these two successively recorded packets.

3.3 Creating an Analysis

From an implementation point of view, an analysis is a concatenation of operator instances in a stand-alone Ruby script. This script consists of commands to instantiate and configure the different operators and a call to the last operators' `getResult` method. If the GUI needs to execute an analysis, it generates the corresponding script and then executes it in a separate process. Due to this architecture, an analysis can also be invoked from shell or Ruby scripts to perform batch processing, or it can even be integrated into other programs.

3.4 Example Operators

Up to now, the general architecture of EDAT and the idea behind the operator concept have been described. In order to get an impression of the capabilities of this concept as well as on the different operations that are already supported, this section presents some of the existing operators.

In our experiments, it was often necessary to partition the input data into subsets and perform a certain operation on each subset. An example is the calculation of delivery ratio per second, where the packets sent in a certain second are the subset and the operation is counting these packets. This operation is implemented by the `GroupBy` operator that is inspired by the `group by` clause of SQL. It builds such subsets and executes an operator on each subset. However, it is not limited to simple aggregate functions but can also apply any operator to the data in the groups. Similar to `GroupBy`, the `Join` operator is inspired by the corresponding SQL expression. `Join` combines or matches the input from two flows to one flow based on a configurable key that serves as join-criterion. The problem of duplicate keys that occurs when two rows of data with the same content are joined is solved by appending "1" or "2" as a suffix to each key.

In order to interact with an SQLite database, the operator `SimpleSQL` can be used. It is configured with an SQL statement and returns the result in the well-known form of an array of hashes. Furthermore, there also exists an `Insert` operator to load the data in the flow into an SQL table. For this, the structure of the flow is analyzed and an appropriate table is created, then the single lines are inserted into this table. A concatenation of these two operators even allows arbitrary SQL statements to be executed on the data in the flow: `Insert` integrates the data in the database and `SimpleSQL` executes the target query.

The important task of visualizing analysis results is performed by the `PlotAll` operator. It is designed to produce two dimensional plots with one or more curves. To do this, the operator assumes that each row in the input flow contains the data for one point on the x-axis. The field representing this point can be configured as one of the input parameters. All other values in a row are plotted as data points that belong to the configured x-value. The plotting itself is performed by gnuplot: the preprocessed data is written to a file in appropriate format and then an instance of gnuplot is created for plotting. As the gnuplot files as well as the resulting postscript graphs are stored in the filesystem, `PlotAll` implicitly creates a chronologically sorted archive of all plots.

4. ADVANCED FEATURES

4.1 Automated Caching

When analyzing data and working with it, a user often just changes the parameter of one operator (e.g., the scope of a filter) and re-executes the whole analysis to examine the influence on the result. Furthermore, most analyses are developed in a step-by-step process in which the user adds a new operator to the end of the current analysis-flow and then re-executes it to examine the output. In both cases, the same calculations are repeated over and over again, a rather time consuming task for large amounts of data.

To cope with this, EDAT supports the caching of previous computations. This is implemented in the base class `Operator` from which all other operators are derived. Thus, newly created operators automatically inherit this feature. For the caching, the result of a computation is serialized and written to a file-system directory. If the configuration of an operator has changed, a recalculation is necessary, otherwise the result can be loaded from the cache. To determine whether a recalculation is necessary, EDAT uses a fingerprint of the operators' configuration that changes as soon as the configuration parameters change. Depending on their type, these parameters are treated differently when included in the fingerprint: 1) in order to determine whether the input provided by another operator has changed, its fingerprint is used. 2) If the parameter is a file, the modification timestamp is considered. 3) Parameters like strings or numbers can be directly included. All this different information is then concatenated. In order to avoid that the fingerprint becomes too large due to these concatenations, an md5sum [16] over all these values then represents the operators' fingerprint[2].

4.2 Executable Pieces of Code

As EDAT is implemented in a scripting language, it is possible to configure operators with pieces of code that are evaluated and executed at runtime. In contrast to a compiled language like C/C++ or Java in which the operations must be specified at compile time, this allows for more generic operators. With EDAT, the data can be modified in ways that would normally require a significant amount of manual programming. For example, in one of our experiments packets of varying size have been sent over a multihop network. To allow for a unique identification as a packet travels from node to node, each packet carries a consecutive number. This number is wrapped in a UDP packet that is itself wrapped in an IP packet. Extracting this number requires to take the payload of the IP packet (i.e. the UDP packet), strip the eight UDP header bytes, and convert the rest to an integer. For this task, the operator `ApplyOperation` is configured with an executable piece of code as third argument:

```
ApplyOperation.new(output, "ip_data",
                  "[8..-1].to_i()")
```

The code snipped from `ApplyOperation` that performs the modifications in the `process` method looks as follows:

```
lines.each do |line|
   line[@key] = eval("line[@key]" + @operation)
   result.push(line)
end
```

The `eval` method provided by Ruby evaluates the expression that it gets as argument. Instance variables start in Ruby with the at sign "@". In the current example `@key` is the second and `@operation` the third argument of the `ApplyOperation` constructor. Due to the above configuration, the performed operation is thus

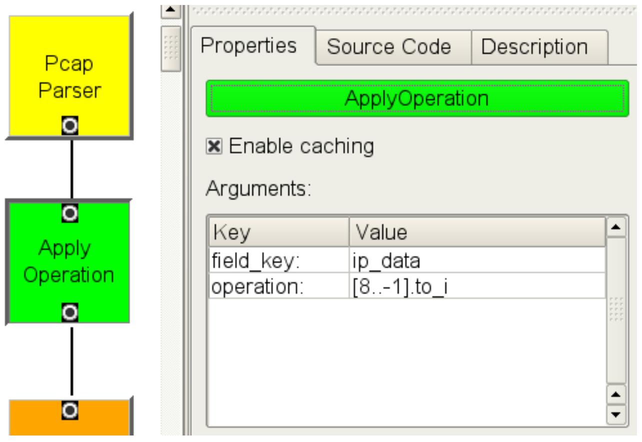
```
line[ip_data] = (line[ip_data])[8..-1].to_i()
```

Thereby the bytes starting at position eight until the end are extracted from the payload and the "to_integer" method is applied to the resulting substring. The configuration of this transformation via the GUI is shown in Figure 2. After the data flow has passed the corresponding operator, the `ip_data` field contains the decoded packet sequence number instead of the IP payload.

5. CASE STUDY

In the introduction, the calculation and plotting of delivery ratio has been used as an example for a standard analysis. How this analysis is conducted with EDAT will be demonstrated with data from one of our experiments. Here, we use the libpcap trace files from two laptops equipped

[2]Note that a harmful fingerprint collision is very unlikely. For this to happen, 1) an md5sum collision has to occur and 2) the wrongly loaded result must not lead to a crash of the analysis. However, to avoid this, caching can be switched off for the final analysis or the "detect fingerprint collision" option that verifies the fingerprint can be activated.



☒ Enable caching

Arguments:

Key	Value
field_key:	ip_data
operation:	[8..-1].to_i

Figure 2: Configuration of ApplyOperation.

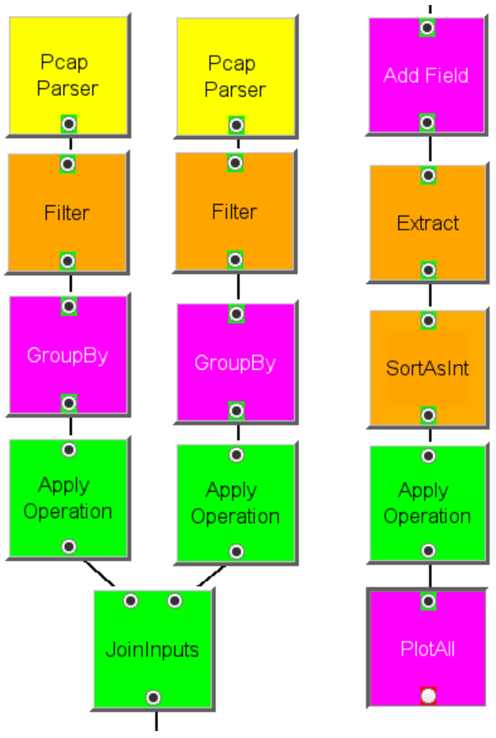
with 802.11b network interfaces. The first laptop broadcasted 80 packets/s and the second recorded these packets with tcpdump. A screenshot of this analysis on the EDAT workbench can be found in Figure 3.

The first half of the analysis shown in Figure 3(a) starts with parsing the files and filtering the packets according to the correct sender address. In the `GroupBy` operator, the packets are first sorted according to their timestamps in 5-second buckets, followed by counting the number of packets in these buckets. After the results are extracted from the special "GroupBy" data structure, they are joined according to their bucket id. The data flow now contains lines of the form `[value_input1 = 416, value_input2 = 220, time_i = 1185347050]` where the first value represents the sent packets, the second the received packets, and the third the timestamp of the bucket. The second half of the processing can now be found in Figure 3(b). After the ratio between the two input-values has been added as additional field to each line of the flow, only the fields required for the plot are extracted. The following operator sorts the rows according to their timestamp which is then normalized to start at time zero. The result of this analysis as it is plotted when executing `PlotAll` can be found in Figure 4.

To get an impression of the impact of caching, the runtime of the above analysis has been measured and averaged over 100 repetitions. Without caching, analyzing the 14 000 packets in the two files took 3.2 s on a 2 GHz Opteron. If caching is activated, the runtime with empty cache is 5.5 s as new results must be written to the cache files. This improves if cached data can be reused, e.g., if the bucket size in the `GroupBy`-operators is changed from five to ten seconds. Here, the results of the two previous `Filter`-operators are reused, resulting in a duration of 2.5 s. This shows the trade-off: although twice as fast as the analysis with empty cache, the improvement is only moderate compared to the execution without caching. This changes if more of the previous calculations can be reused, e.g., if the delta calculation in the first operator of Figure 3(b) is changed to a subtraction instead of a division. Here the analysis only takes 0.02 s.

6. CONCLUSIONS

In this paper, we have presented the extensible data analysis toolkit EDAT that is designed for the evaluation of net-



(a) Part 1 (b) Part 2

Figure 3: Example analysis for plotting the throughput between two nodes.

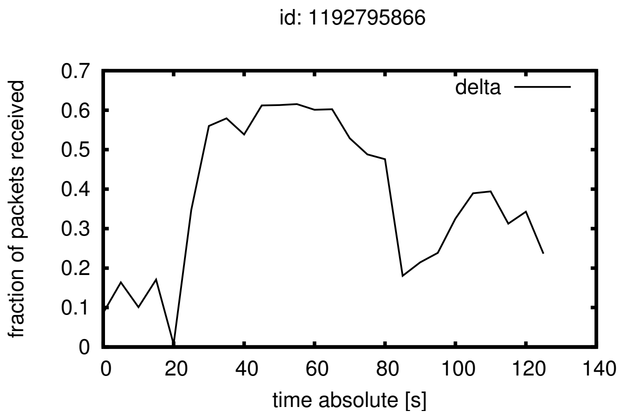


Figure 4: Graph resulting from the analysis shown in Figure 3.

work simulations and experiments. It follows a flow-based approach where modular operators are combined to form the analysis. The tool is easily extensible, comes with a rich set of operators, and has already been used to produce the graphics for a number of our own scientific publications. Although extending the tools' capabilities requires some programming knowledge, analyses that use existing components can be created by non-programmers with its graphical user interface. EDAT eases the task of evaluating network simulations and experiments, allowing researchers to concentrate on the original problems rather than on the development of tools to analyze them.

The extensible data analysis toolkit EDAT is available on `www.cn.uni-duesseldorf.de/projects/EDAT` under the terms of the GPL.

7. REFERENCES

[1] W. Kiess, A. Tarp, and M. Mauve, "Real-World Evaluation of Ring Flooding," *Mobile Computing and Communications Review*, vol. 10, no. 4, pp. 11–12, Oct. 2006.

[2] W. Kiess, S. Zalewski, and M. Mauve, "Improving System Clock Precision With NTP Offline Skew Correction," in *MedHocNet '07: Proceedings of the 6th Annual Mediterranean Ad Hoc Networking Workshop*, June 2007, pp. 159–164.

[3] B. Scheuermann, W. Kiess, M. Roos, F. Jarre, and M. Mauve, "On the time synchronization of distributed log files in networks with local broadcast media," *IEEE/ACM Transactions on Networking*, 2008, accepted.

[4] "Tcpdump: a tool for network monitoring," http://www.tcpdump.org.

[5] "The wireshark network protocol analyzer," http://www.wireshark.org/.

[6] G. Iannaccone, "Fast prototyping of network data mining applications," in *PAM '06: Proceedings of the Passive and Active Measurement Conference*, Mar. 2006.

[7] B. Scheuermann, H. Füßler, M. Transier, M. Busse, M. Mauve, and W. Effelsberg, "Huginn: A 3D visualizer for wireless ns-2 traces," in *MSWiM '05: Proceedings of the 8th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Oct. 2005, pp. 143–150.

[8] "KNIME, The Konstanz Information Miner," http://www.knime.org/.

[9] "Matlab," http://www.mathworks.com/products/matlab/.

[10] "Maple," http://www.maplesoft.com/.

[11] "OpenDX - Open Data Explorer," http://www.opendx.org/.

[12] "LabVIEW for Measurement and Data Analysis," http://zone.ni.com/devzone/cda/tut/p/id/3566.

[13] "IETF IP Performance Metrics Working Group," http://www.ietf.org/html.charters/ippm-charter.html.

[14] "Trace graph - Network Simulator NS-2 trace files analyser," http://www.tracegraph.com/.

[15] "Gnuplot, a portable command-line driven interactive data and function plotting utility," http://www.gnuplot.info/.

[16] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321 (Informational), Apr. 1992, http://www.ietf.org/rfc/rfc1321.txt.