An open source network traffic performance monitoring and diagnostics tool.

www.trafmon.org

# Architectural Design

Thomas Grootaers, Luc Lechien

Software Release 1.0

2020-10

# COPYRIGHT, LICENSE AND TRADEMARKS

Original text is © 2020 AETHIS sa/nv Belgium, Thomas Grootaers, Luc Lechien

This material is based upon work funded and supported by the European Space Agency and the Belgian Federal Authorities (BELSPO) under GSTP Contract Nr ESRIN 4000128964/19/I-EF with AETHIS sa/nv, Belgium.

The view, opinions, and/or findings contained in this material are those of the authors and subsequent free contributors and should not be construed as an official ESA, Government or AETHIS position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favouring by ESA or AETHIS.

NO WARRANTY. THIS AETHIS MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. AETHIS MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. AETHIS DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT] This material is for approved for public release and unlimited distribution under the terms and conditions of Open Source Apache License v2.0 (https://www.apache.org/licenses/LICENSE-2.0.txt, OSI Approved https://opensource.org/licenses/Apache-2.0), which governs its use, distribution, modification and re-publication.

Adobe is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.
AngularJS is a trademark of Google, Inc., https://angularjs.org/
CentOS Marks and JBoss are trademarks of Red Hat, Inc. ("Red Hat").
CERT is a registered trademark owned by Carnegie Mellon University
Eclipse and BIRT are registered trademarks of the Eclipse Foundation, Inc. in the United States, other countries, or both.
JQuery and JQuery UI are trademark of OpenJS Foundation, https://openjsf.org/
Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.
MaxMind, GeoIP, GeoLite, and related trademarks are the trademarks of MaxMind, Inc.
Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.
OpenSSL is a registered trademark of the OpenSSL Software Foundation in the U.S. and other countries.
Oracle, Java, MySQL, WebSphere and Solaris are registered trademarks of Oracle and/or its affiliates in the United States and other countries.
Python is a registered trademark of the Python Software Foundation.
Tomcat® and Apache HTTP Server™ are (registered) **trademarks** of the Apache Software Foundation.
UNIX is a registered trademark of The Open Group.
WebLogic is a registered trademark of IBM Corp. in the United States, other countries, or both
Wireshark is a registered trademark of the Wireshark Foundation.
All other trademarks are the property of their respective owners.

# DOCUMENT HISTORY

| Release | Date | Change |
|---|---|---|
| 1.0 | Oct 2020 | First issue |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# ACKNOWLEDGEMENTS

The authors wish to acknowledge the valuable contributions of all ancient employees of the AETHIS® Company in Belgium, who have worked on the successive versions of the base software and its documentation from which the open source trafMon software is derived.

In particular, special recognition is given to Jacques Maes, David Orban, Jonathan Van den Schrieck, Benoît Liétaer, Julien Denis, Thomas Soupart, Fabien Coenegrachts, who have more specifically participated to its elaboration. Also a thought is given in memory the authors' deceased associate, Luc Steenput, who has heavily promoted the initial idea and subsequent enhancements of the tool, within the European Space Agency and elsewhere.

Lastly, the authors wish to acknowledge the strong support of ESA staff members: Manfred Lugert, Erling Kristiansen, Johan Stjernevi, Manfred Bertelsmeier, Gioacchino Buscemi, Michele Iapaolo, Andrea Cogliandro and Claudia Neroni, as well as of officers of the Belgian BELSPO Federal Service, Jacques Nijskens, Agnès Grandjean and Hendrick Verbeelen.

# TABLE OF CONTENT

# TABLE OF FIGURES

# ACRONYMS AND ABBREVIATIONS

# 1. TRAFMON DESIGN DRIVERS

## 1.1 ORIGIN

The trafMon platform has its origin from two developments implemented for the European Space Agency (ESA).
A first version was designed in the frame of the Galileo Programme, for the qualification of worldwide ground segment links, which are supporting time-critical traffic flows.
A second version was developed for the Earth Observation Programme, for the measurements, auditing and troubleshooting of communications over the ESA's Earth Observation Payload Data Network. The provided services include the observations of the files transferred during FTP sessions and the underlying TCP connections efficiency.

The initial open source version is based on a further enhancement of the platform.

## 1.2 PRECISE MEASUREMENT OF ONE-WAY FLOWS

One of the target environments for which the trafMon tool has been designed is that of a (start-shaped) multilink network transporting traffic under stringent performance requirements, in terms of latency (one-way delays) and continuity (nearly no risk that the guaranteed latency is not respected during operational conditions).

A good candidate for such rigorous network service level is the communication network that supports remote surgery activities, being imposed safety-of-life working conditions.

Some link could be long distance, realised by a combination of a terrestrial path and a VSAT satellite hop. More generally, the communication links is made on a series of segments, where the quality of each of them would be measured.

Thanks to Global Navigation Satellite Systems (GPS, Glonass, Galileo, Beidou, QZSS, NAVIC), it is possible to have a precise and stable time reference at all sites spread worldwide. Its distribution, over the LAN, via NTP protocol permits to keep the millisecond accuracy.

By spreading probe devices at every site being source or destination (or both) of real-time sensitive flows, it is possible to capture a copy of every target flows packet and measure its timestamp of occurrence there. Provided the packet (or a part of) isn't modified over its network travel, it is possible to identify it uniquely by its unchanged (part of) content at different hops over its path. Three or four bytes of hash function applied to this packet content produces a rather reliable signature.

The disseminated probes can then centralise their per-packet pairs of observations *{packet signature, capture timestamp}* to permit the measurement of one-way latencies and the detection of too-late or lost packets. Ideally, a *flow identifier* should be added, for permitting to distinguish network performance behaviours among different flows of different priorities.

Of course, the probe observations would typically be injected as an additional auxiliary data flow, mixed with the real-time one. Therefore the protocol and its encoding mechanism, used for this probe-to-collector transmission of observations must be

- quite compact (~1 % of the observed traffic volume or less),
- fully controlled and regulated,
- reliable with acknowledges and retries at slow pace.

At the central site, the per-packet partial observations are gradually reconciled to derive the latency measurements.

After a certain timeout, defined on the basis of maximum transfer time, incomplete per-packet records are detected to identify and measure the lost or too-late packets.

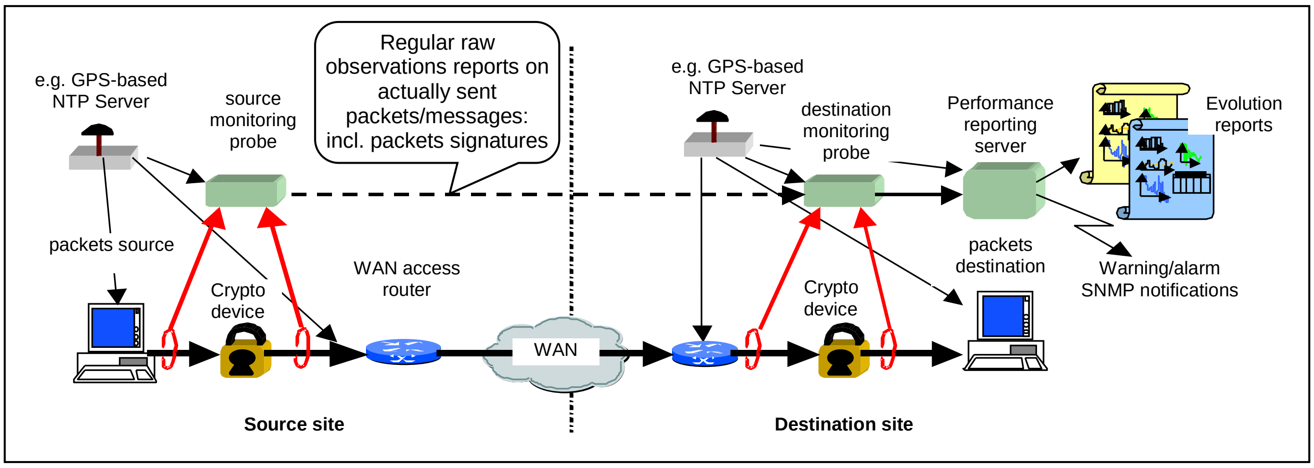
e.g. GPS-based NTP Server
source monitoring probe
Regular raw observations reports on actually sent packets/messages: incl. packets signatures
e.g. GPS-based NTP Server
destination monitoring probe
Performance reporting server
Evolution reports
packets source
Crypto device
WAN access router
WAN
Crypto device
packets destination
Warning/alarm SNMP notifications
Source site
Destination site

**Figure 1 Principle of End-to-End One-Way Per Packet Measurement**

Network/Port Address Translation (NAT/PAT) does still permit to sign the same way each packet before and after the modification by selectively masking the header field(s) subject to modification. Encryption, however, does only permit to compare flow measurement on the basis of the Quality of Service (QoS) classes, as the IPv4 Type-of-Service (TOS) byte is copied in the (readable) outer header of VPN tunnelled packets, although, in some cases, a heuristic based on close timestamps before and after the crypto device could permit the per packet match.

In the more general case, the deployment of probing points at several places over the network path, with one or multiple packet capture interfaces per probe device, is illustrated below.

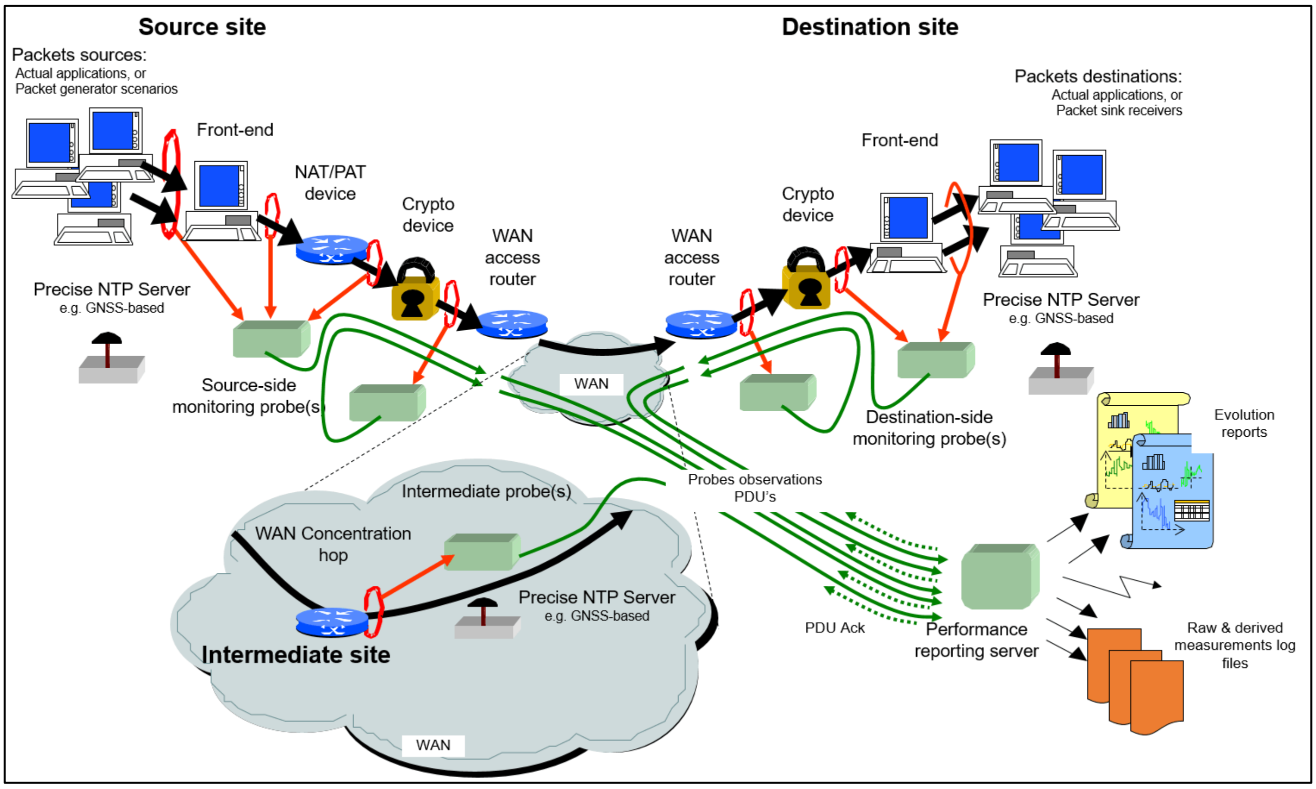

**Figure 2 General Case for One-Way Performance Measurement**

Another way to also get intermediate timestamps during the network traversal of the packets is the exploitation of the IPv4 Timestamp Option. This does however place constraints on the way the network travel path is implemented: intermediate timestamping network devices must also be precisely time-synchronised,, most firewall on the market block (silently drop) packets with IPv4 options and the IP Timestamp fields are hidden and unmodified during their travel inside encrypted VPN tunnel.

Anyway, all above depicted interesting capabilities must be supported by the flow matching, the packet signing and the timestamps extraction and merging trafMon mechanisms.

In order to implement a resilient measurement architecture, the trafMon central processing system can be duplicated. Each probe is then assigned two or more destination collectors for its measurements. The observations PDU maximum size and minimum inter-PDU delay can even differ between collector instances. Because every collector receives the same set of probe observations, each can independently build the same database content and generate the same report figures. When one fails or is (partly) unreachable, metrics can still be available at the other(s) trafMon central site(s).

## 1.3 QUITE COMPACT ONE-WAY OBSERVATIONS

When monitoring a network environment supporting time-critical traffic, the transmission of observations made by the distributed probes must not interfere with the performances of the monitored communications.

Therefore, the encoding of the one-way partial observations sent by the respective probes, and probably injected with the time-critical traffic itself, must be negligible in volume. Hence the need to implement the most compact possible encoding mechanism for the one-way related data, where every bit is meaningful.

## 1.4 FULL-STACK PROTOCOL ANALYSIS AT WIRE SPEED

Looking at every packet simply for flow identification, packet signing and timestamping is rather limited in scope. First of all, the probe can also measure the packet sizes and derive volumes and duration metrics similar to what classical NetFlow information provides.

Although this size information could be added by the probes to their respective per packet observations, this would produce a gigantic amount of unprocessed data flowing through the central collector for simply aggregating them into basic metrics (*foreseen in the configuration DTD, but never implemented*). Hence it is by far more efficient that the probe itself is able to produce higher level measurements, delivered at slower pace (with aggregation interval varying between 10 seconds to 1 hour).

Even more, the computing resources of the distributed probes, combined with all the amount of information available in the stack of protocol headers of the inspected packets, allow to produce far richer indicators for network performance monitoring and communications troubleshooting.

Not only does the probe need to aggregate raw observations from individual packets, but it must be able to reconstitute the protocol exchanges of layered communications and perform stateful analysis of some peer-to-peer dialogs to produce behavioural interpretations.

The top-level currently achieved in the probe implementation is the complete follow-on of FTP session exchanges and automatic learning and detection of data transfer connections, established in active or passive mode, for the provision of directory content listing and for the GET or PUT transfer of individual files. Especially, FTP passive data connections occur between totally random TCP ports, and are therefore never qualified as FTP communications by other existing *Network performance monitoring and* diagnostics (NPMD) tools.

This being said, we are facing a big performance challenge. A tool like Wireshark provides support for full analysis of nearly all existing communication protocols, even the more exotic ones. But at the cost of multiple interpretation passes, which can only be conducted offline or on low data rate exchanges.

The Linux kernel implementation of the packet capture capability has already responded, in part, to the need of efficiently buffering packets during traffic peak, and permitting to inspect their respective content, without copy to the user space of the probe program.

The architecture of the probe software has been split as a pipeline of two processes:

- The father is in charge of the stateless inspection – in situ – of captured packets, one by one, and on the as-efficient-as-possible matching of all flow classes combined conditions (in a single traversal) to detect the (list of) flow class(es) the packet belongs to, or whether the packet is of no interest and can be skipped.

  For TCP connections transporting actual volumes of data (e.g. file transfers over FTP or HTTP), it was initially thought to perform 1-out-of-N sampling of packets to conduct the further analysis, in order to cope with high data rate. Finally, this has not been implemented, and replaced by the choice (per flow class) of retaining every packet of the TCP connection, or only the SYN (start), FIN or RST (end) marking the boundaries of the data transfer. This allows to measure the transfer total duration and, provided less than 4 GBytes, the actual size of the transferred payload. This way, most (nearly all) of the captured network packets do not require further analyse, and can be skipped by the father probe.

- The probe child process is fed by the father with records of relevant data extracted only from the useful packets and optionally, for those under one-way precise monitoring, with the actual packet content (for signature hash).

  It is in charge of IP reassembly, of matching the datagrams with the discovered flow instance(s) – as per <*GranularFlow*> specification applicable to its flow class(es) – of stateful protocol analysis (TCP connection and FTP session exchanges, request/response matching for ICMP Echo and UDP SNMP/DNS/NTP transactions), and of aggregation of numerous protocol counters.

  Finally, the child implements the trafMon PDU custom UDP-based controlled protocol to deliver its various types of raw observations to the central collector(s).

This is illustrated in Figure 3 below.

On multi-CPU/multi-core current processor architectures, the father probe process locks itself on the processor #0, while the child process assigns an affinity for all processors but the first one. This way, the two processes are really running independently, in parallel.

Note also that, for proper measurements, the probe must deactivate the offloading in the NIC card of the Ethernet reassembly and checksum verification and discard, for every of its packet capture interfaces.

The use of the portable *libpcap* library also permits to mimic the actual real-time packet capture through the replay of a previously captured recording. The probe supports the replay simulation, where packets are handled by respecting their original inter-packet delay. More useful, the packet capture file can be processed at full speed. This permits to measure the maximum performance that the probe can achieve on a given hardware.

## 1.5 FULLY CONTROLLED, ALTHOUGH RELIABLE, PROBE OBSERVATIONS PROTOCOL

In section 1.3 above, it is highlighted that the observations conducted in a time critical environment may not create a perturbation of the performances under precise monitoring.

Not also those precise observations (i.e. the one-way measurements) must be compact, but the protocol used by the trafMon probes to deliver (all) their measurements to the central collector must be controlled and reliable.

Reliability implies that every probe protocol data unit (trafMon PDU) correctly received by the collector must be acknowledged or retransmitted up to a reasonable amount of time.

Control of the injected data rate has several implications on the way the PDU units are injected:

- Maximum PDU size and maximum build duration

  Only small-size packets (200-300 bytes) would have a limited impact on the transmission duration of following time critical packet(s), when serialised over a low bandwidth WAN access link. Inversely, exploiting the largest possible packets (1500 bytes), where bandwidth capacity is not a problem, permits to limit the number of probe packets that could clash with the measured ones.

  But, during quiet traffic periods, it is necessary to limit the time a PDU stays "under construction", so as to avoid delays in refreshing central measurements or, worse, that part of the observations are considered too old or already missed by the collector. Hence the need to flush any type of PDU data after its maximum build time.

- Minimum inter-PDU time gap

  Probe PDU are sent as UDP unreliable datagram units. Where a big burst of PDU units are ready for (re-)transmission at the same time, pushing them all in burst could have two effects: impact the forwarding of monitored network traffic and overflow gateway buffers that could drop some datagram units. Hence the need to implement the probe sending time into a series of time slots, distant from each other by the specified minimum time gap, where at most one PDU sending (initial or retransmission) is scheduled. This way, the probe output is kept smooth, preventing any burst.

- Maximum retries, initial timeout, timeout increment and timeout multiplier

  When the acknowledge response to a given PDU is not received within a running timeout period, it must be re-sent. The timeout for its next period could be increased, possibly multiplied (doubtful usefulness) and incremented by specified values.

  When the number of retries is reached, after a sequence of retries + 1 successive timeout increased delays, the PDU must be destroyed by the probe.

- Long retry mode, final timeout

  When the timeout after maximum retry of a PDU exhausts, the probe is entering into the so-called "long retry mode". The connectivity with the collector is, at least, bad or probably broken for a while. Hence no need to continue to bomb the communication channel with resent of all PDU units.

Only the one PDU that has exhausted its retry count is regularly sent at a period corresponding to the initial timeout value.

Newly generated PDU units continue to be retried at the maximum timeout value (slow pace) until they are older than a given final timeout.

- Break border window

  When the probe PDU units are travelling over the same path as the (maybe time critical) monitored traffic, a break of connectivity between a probe and the collector (leading to the long retry mode) is also influencing the monitored traffic. Interesting observations are therefore those that are made during a time window just before the loss of connectivity. Hence those PDU units that belong to the break border window are preserved, and continue to be perpetually sent at the maximum timeout period. Those observations are hopefully centralised after the network connectivity is restored, permitting later analysis of the traffic behaviour just before the break. Younger observations are them destroyed after a while.

- Key definitions and important (low volume) observations

  Some types of PDU information are however permanently retried (at maximum timeout period): those observations are not necessary for the correct handling of subsequent measurements (definitions of discovered flow instances, definitions of buckets for histogram distributions), or those observations whose volume is limited and which are considered really useful to preserve (the individual records of FTP file transfers).

## 1.6 SINGLE CONFIGURATION FILE FOR DISTRIBUTED ONLINE FUNCTIONS AND SYNCHRONISED UPDATE

All trafMon probe and collector instances share a same version of a single XML configuration file. Tuning parameters are specified as XML attributes of the definitional tags.

Updating the configuration may concern one or more probes and the collector(s), and influence the observations made after the change. Hence it is useful that a switch over to a new version of the configuration file is done at the same time by all trafMon online components altogether.

Furthermore, a warn restart is nearly impossible to implement, due to the complexity of internal configuration-driven data structures, and due to the probe child inheritance of those structures initialised by the father before forking. Hence a configuration update means the probable loss of observations under building. This creates a glitch in the raw measurements, sufficiently short in time to be negligible.

If the configuration update (process stop and re-start) isn’t performed in synchronism, partial data glitches would be spread over a significant duration. So the synchronisation of the automatic restart with the new configuration causes a short glitch, general but limited in time, and accompanied with the generation of warning events that are kept in the database together with the performance measurements, allowing a proper interpretation of the discontinuity during offline interpretation of the performance figures any time later.

## 1.7 DETECTION, LOGGING AND NOTIFICATION OF REMARKABLE EVENTS BY TRAFMON COMPONENTS

The probe and the collector programs can detect several types of remarkable event occurrence, e.g.

- stop and start of a trafMon online component, configuration switch over
- lack of traffic or saturation of the kernel-resident packet capture buffer
- saturation of the probe processing and restoration of nominal capacity
- lack of connectivity between probe and collector
- loss of partial observations in one-way measurements, detected clash (collision) of signature hash (same signature for two or more “simultaneous” packets)
- [threshold detected on raw metrics – not implemented yet]

Those events explain possible abnormalities reported by the performance indicators. Hence, these are handled as a specific type of observations that are gathered by the probes and collector, and logged together with the other raw observations for regular loading in the database.

## 1.8 CONDITIONALLY COMPILED EMBEDDED SNMP SUB-AGENT

The probe and the collector programs optionally implement an embedded Net-SNMP sub-agent.

The above-mentioned event occurrences can then be notified, in real-time, to a network management tool as Traps or retry able Inform requests.

Furthermore, a custom MIB is defined, respectively for the trafMon probe and the trafMon collector programs, allowing the SNMP manager to get statistics information about the behaviour of the distributed online components.

## 1.9 TUNEABLE LOGGING OF SYSTEMATIC PRODUCTION OF TRACE AND ERROR MESSAGES

Every software module (C code file) can be assigned a given level of verbosity and destination log for the diagnostics messages that systematically decorate the program code.

Those messages do not only permit to verify, validate, debug the program code, but are also useful during operations to get insights on the monitored traffic and thei handling by the trafMon probe and collector.

## 1.10 PRODUCTION OF RAW OBSERVATIONS AS A COLLECTION OF ASCII LOG FILES

Each type of raw observations made by the probe as well as the one-way metrics produced by the collector and all trafMon online processing events result in dedicated ASCII logs written by the collector in a spool directory.

This gives the user total freedom for the further processing of the raw data.

## 1.11 DATA AGGREGATION IN A SQL DATABASE

trafMon implements off-line functions for the aggregation and flexible report generation of the protocol detail metrics.

A regularly run Python/SQL script loads the next available chunk of ASCII data logs, in part as update of pre-aggregated tables (at 1 minute, 1 hour and 1 day), in part as new rows of independent data records (events, TCP connections, FTP transfers) in a relational database.

Previous trafMon implementation was using PostgreSQL and stored procedures written in Perl (the write-only, even write-once language – I won't make only friends here ;-).

But having faced some performance issues during operations have lead us to choose MySQL for the current implementation.

In order to permit further aggregation of data without loss of statistical information, every counter is represented, for a given time interval, by a population (number of observed samples), a <u>sum</u>, a <u>sum of squares</u> (for deriving the standard deviation), a <u>minimum</u> value and a <u>maximum</u> value.

Care has been taken to support multiple instances of databases. The name of the runtime database can be changed in configuration file. The database instance is initially created with the first loading of observations. All table templates (for temporary working tables as well as for persistent tables) are provided in a separate `*trafMon_template*` database instance with no actual data. This database also holds all the necessary stored procedures, most of which take the actual database name as first argument.

The reporting menu bar let you first select the database to use as reporting source (by convention, all names would start with '*trafMon'* prefix.

Extensive use of physical partitioning of data tables, based on time intervals, permits to destroy ancient fine grain data at no cost. doing the same via DELETE FROM queries would kill the database server process, already rather busy in loading and aggregating potentially large volumes of observations.

## 1.12 FLEXIBLE DATABASE REPORT WRITER WITH CHARTING

Although this is a bit outdated mechanism, without exploiting all the dynamic capabilities offered by HTML5, we have chosen Eclipse/BIRT. It is an open source tool that gives full flexibility and offer ease of quickly designing report templates. It that can extract and further process database values, and present them as a mix of text, tabular data and business charts. It can generate on-demand Web based reports and can also produce PDF documents in batch mode.

Furthermore, is extensive tuning capability, thanks to JavaScript, allowed us to layout some meaningful reports responding to specific requirements.

A set of pre-designed reports has been drawn. But the users can adapt them through the Eclipse/BIRT Designer, and can design their own.

## 1.13 NETWORK MODELLING PER ACTIVITY, PER LOCATION AND PER REMOTE INTERNET COUNTRIES

One of the requirements for trafMon was to exploit the architectural model of a private network belonging to an Organisation. Their IP nodes (workstations and servers) are lying in several *Locations* (rooms, floors, buildings, sites, internal and DMZ LAN segments …). Their IP nodes are also partitioned into different Activities (can be business units, departments, missions, functions). Their network is also interfaced with the general Internet, where peers (being clients or remote servers) can be qualified, thanks to GeoIP™ free or commercial data files, into different countries (cities and ASN numbers).

Further aggregation of flow volumes data, possibly extended with NetFlow records, can be conducted. Daily batch preparation (at night) produce tables (hourly and daily) with dat flows qualified peers and protocol service name. The user can then select a specific report layout (manager, operator or conversation views), make a choice of Activity (or all), of Location (or all) or of a specific host IP address, and specify a time span (day, week month) to let the BIRT Engine proceed to the further aggregation of target data and the population of the meaningful report charts.

In order to assist the user in making dynamic choices for the report parameters, based on actual data present in the database, a JavaScript/AngularJS/PHP Web application has been drafted. It presents a selection menu-bar for synthesis reports and an alternate version suited to the protocol details reports.

## 1.14 POSSIBLE FUTURE ENHANCEMENTS

This tool is provided as open source under permissive Apache2 license. The idea is to permit interested contributors to join their forces and respective expertise in order to transform this initial trafMon tool as flagship free product. Furthermore, nothing prevent that trafMon is also later embedded in commercial products.

Hence, below is a collection of enhancements that have already been thought about while developing and making use of the initial tool.

### 1.14.1 Support of Passive Tap Devices: Pair of Linked Interfaces

The Linux capture ring buffer and the *libpcap* interface provides independent buffering and next packet access function, independently for each capture interface.

Currently, the probe packet inspection loops through all pending packets at a given capture interface, before looking at the next one.

This is not an issue when using mirror port of a switch device or active network tap device, where the packets are delivered in proper sequence of occurrence, whatever their direction. This way, the traffic reported by the different capture interfaces of a same probe can be processed asynchronously from each other.

With purely passive network tap devices, that are inserted in a trunk link, the packets in one direction are replicated at one monitor port, while the packets in the other direction are presented at the other mirror port. Each monitor port is connected to one capture interface of the probe. The capture buffering operated by the Linux kernel, combined with its scheduling of the probe (father) process, implies that successive one-way packets can accumulate in each of the two-direction probing interfaces. By handling all those in one direction, then only those of the other direction, the correct “request/reply” handshake sequence is not followed in chronological order. So the protocol stateful follow-on of the exchanges (UDP-based transactions, TCP handshakes, FTP command/response dialogs) isn’t correct anymore.

By implementing the possibility to declare pairing of interfaces in the trafMon probe configuration, and by implementing a lookup mechanism where only the oldest packet from the top of the two paired interfaces capture queues is next inspected, the respective packet sequencing of the bi-directional traffic will be respected. This would add proper support of passive tap devices to the trafMon probe.

### 1.14.2 Further Exploiting the Multi-processing Capabilities

A slight change in the probe code can lead to specify (configure) which CPU core ID the father process should lock itself on (and maybe also for the child). This way, several instances of a probe can co-exist on a same computer, each processing its own set of capture interfaces, without competing (too much) for common hardware resources.

This is the simplest way to increase the probe performance capacity.

But for handling 10GB network interfaces, a rework of the software architecture would be required. The “father” process (stateless packet decoding and flow matching) could be multiplied, for a same interface, each being assigned a partition of the traffic to capture, thanks to specially crafted *libpcap BSD Packet Filters (BPF)* implemented in the Linux kernel.

Similarly, selection criteria could be added, for instance, to the trafMon Flow Class definitions, to direct the decoded packet observations to different instances of the “child” process performing stateful protocol exchanges analysis and data aggregation.

### 1.14.3 Support of IPv6

Although foreseen initially, it has been decided not to take into account of IPv6 when developing the initial trafMon probe.

One key issue here is the way that efficiency consideration imposed to the probe coding has over exploited the fact that an IPv4 address can be held in a 32bit unsigned integer. This is especially used in flow class/instance matching and in observations encoding.

Adding such support afterwards will imply to revisit the entire C source code.

### 1.14.4 Online Threshold Detection on Raw Measurements

A possible extension to the trafMon online functions – probably best implemented in the collector – would be to compare specified raw real-time metrics with given thresholds (static of dynamic derived baseline) and to generate online notifications and event log records upon threshold trespassing and restoration.

### 1.14.5 Plug-in Interface for Additional Protocol Analysers and Observations

One capability that had been implemented in the more basic initial trafMon implementation (one-way only per packet individual observations) was to design a generic plug-in interface in the probe and in the collector code, where additional observations could be extracted from the packets under inspection, could be encoded the probe PDU units, then decoded and handled by the collector for producing complementary data logs. Each pluggable module was event implemented as a dynamically loaded shared object, as specified in the configuration.

The incremental approach, under time pressure, for re-implementing the trafMon software from scratch and with full-stack protocol decoding and stateful analysis, has not permitted to re-design a comparable generic plug-in interface.

It is however possible to design such an approach, to implement a new skeleton of the C software and to import the existing code in the new modular structure, taking inspiration from how the wireshark code is architected.

Nevertheless, this would be a challenge and … a significant work!

### 1.14.6 Maybe Numerical FlowID’s and Table Join are More Efficient

The choice has been made to replace, for the database persistent data tables, the numerical flow identifiers by a meaningful string describing the peer addresses and ports, the flow direction, and the probe interface.

This way, all data tables are independent. But the same flow ID string is repeated numerous times.

It could be more efficient that the table refer to flow instances via a unique numeric ID, part of the table key, with search index; and the flow table can be joined when selecting per flow or per field constituting the flow description.

This way the database tables would be kept more compact (efficient disk I/O), and the search and join could be quicker when producing the reports.

This must be tested before concluding.

### 1.14.7 Find and Use a More Modern Report Writer, Exploiting the HTML5 Dynamic Graphics Capabilities

BIRT is rather powerful and flexible. But is a bit outdated and limited in what concerns the charting. Many reporting and charting programs that people are using today are quite more flashy.

Exploiting the capabilities of a charting library like D3.js (d3js.org) would be welcome, but combined with easy to draw report template designer and SQL interface.

### 1.14.8 Big Data Database Technology and Handling Techniques

The volume of observation can grow quite quickly, and the production of each new report view would require to produce optimized SQL procedure.

Even though, some synthesis reports still take too much time for interactive generation.

Modern technology and data handling techniques used for the “Big Data” discipline would be welcome to replace the rather basic trafMon SQL schema.

# 2. TRAFMON COMPONENTS ARCHITECTURE

The trafMon software is conceptually split in two parts:

- Online functions: corresponding software is in charge of real-time and near real-time monitoring of the traffic and of producing the basic performance observations.
  - One or more **probes** are reading (capturing in read-only stealth mode) the operational traffic
  - The probe or probes are sending their observations though custom formatted UDP packets that are sent via their computer dedicated data port
  - The **collector** receives the probe's observations and sends back PDU acknowledges
  - The collector regularly output the basic observation into flat log files
- Offline functions: consists in **SQL database** regular batch loading and aggregation of basic observations logs into prepared metrics and in offline querying and report presentation of those performance metrics.
  - Regularly run **scripts** and associated **database stored procedures** upload the latest collector output files in working input tables of the relational database.
  - Regularly run scripts, invoking the **BIRT reporting** java runtime on pre-defined report templates, perform SQL data retrieval queries on the database and produce pre-built report files **in batch**, e.g. PDF documents or HTML formatted reports.
  - **On-demand interactive access to the BIRT** java runtime on Apache Tomcat server let a human user (e.g. network operator) to instantiate, through his Web browser, a selected BIRT report template based on custom provided parameters. This interactive generation of BIRT reports is facilitated by a **dynamic menu bar Web application**, based on AngulaJS/PHP, for letting the user select his desired report attributes.

The online functions are optimised C programs, with a XML configuration file that follows a DTD syntax definition.

The offline functions are implemented by a mix of

- Python scripts (and some shell scripts),
- MySQL statements and stored procedures,
- JavaScript embedded in BIRT report templates,
- JavaScript and PHP code to build the AngularJS menu bar Web application,
- Pre-designed BIRT report templates,
- crontab and logrotate configuration

# 3. SOFTWARE ARCHITECTURE OF THE PROBE PROGRAM

In order for the probe to be able to process captured packets at wire speed, the trafMon probe has been designed as a pipeline of two processes linked by a circular buffer implemented in shared memory:

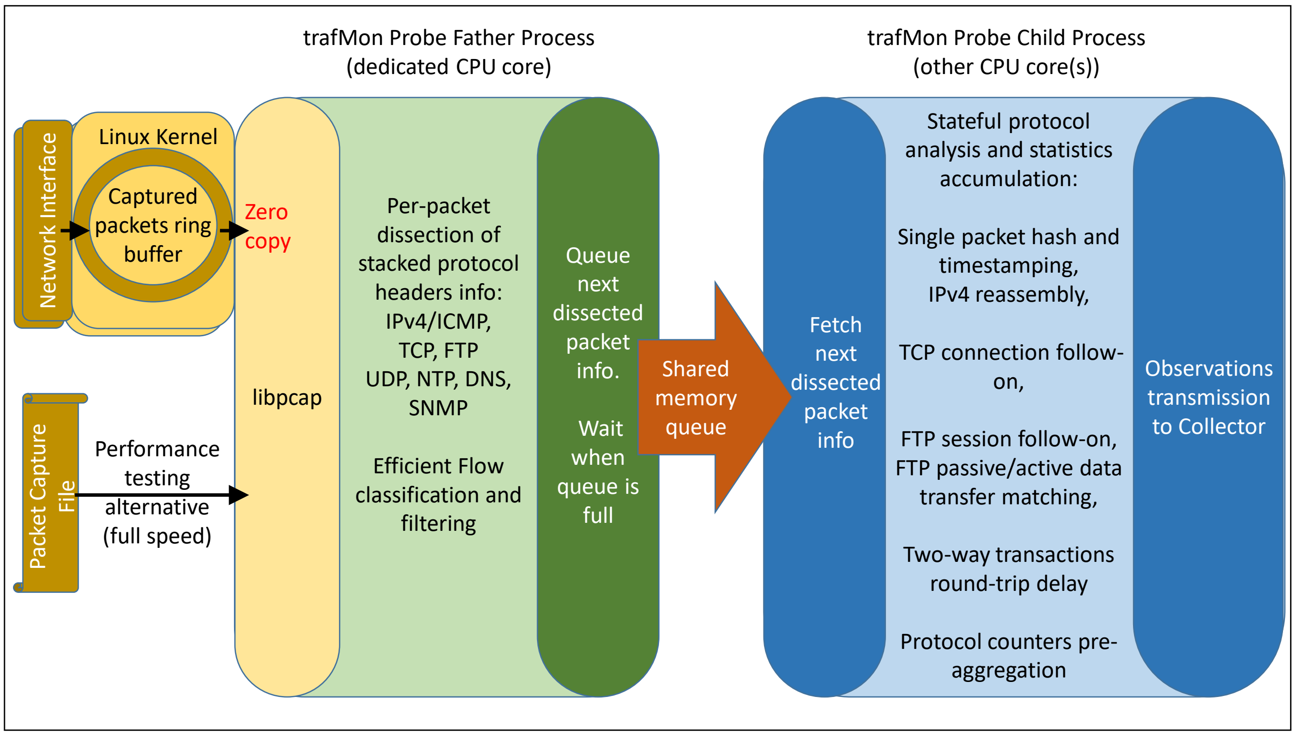
trafMon Probe Father Process
(dedicated CPU core)
trafMon Probe Child Process
(other CPU core(s))
Network Interface
Linux Kernel
Captured packets ring buffer
Zero copy
Packet Capture File
Performance testing alternative (full speed)
libpcap
Per-packet dissection of stacked protocol headers info: IPv4/ICMP, TCP, FTP UDP, NTP, DNS, SNMP
Efficient Flow classification and filtering
Queue next dissected packet info.
Wait when queue is full
Shared memory queue
Fetch next dissected packet info
Stateful protocol analysis and statistics accumulation:
Single packet hash and timestamping, IPv4 reassembly,
TCP connection follow-on,
FTP session follow-on, FTP passive/active data transfer matching,
Two-way transactions round-trip delay
Protocol counters pre-aggregation
Observations transmission to Collector

**Figure 3 trafMon Probe Structure: Pipeline of Two Processes**

The packet capture probe makes use of the portable API of the public domain *libpcap* library. This allows directing the operating system kernel to capture packets at one interface and, according to a user-specified filter, to selectively provide their content to upper layer software. A libpcap context will be given to each capture interface.

*Libpcap* permits to fetch only a portion of the packet, alleviating further processing. And, on Linux, capture filter can run inside the kernel, avoiding to handle irrelevant Ethernet frames.

*Libpcap* also stamps the captured packets as soon as they enter the system. This gives the best possible precision of their respective time of occurrence.

And *libpcap* gathers statistics on its activity; especially the counter of packets dropped due to resource limitations (packets in excess such that the system cannot actually get them all). This information is one source for trafMon health notification.

Modern version of *libpcap*, especially on Linux, permits to memory map a buffer of custom specified size between the kernel and user space, thereby avoiding a copy of the capture portion of the packets. Correct sizing of this internal buffer permits to efficiently absorb traffic bursts during one or a few seconds.

Packet analysis starts by applying custom defined flow classification. Matching a flow class identifies which analyses are to be applied to the packet: as governed by the custom configuration. At first this indicates which protocol information fields are to be dynamically learned to identify a granular flow: the unit for which observations are measured and centralised.

The analysers are of two types:

- Those that extract information from a packet are stateless. Flow identification (flow class and granular instance) form a particular case in that it has to be conducted as a first step, and governs further dispatching of the packet through the analysers, for producing the user-requested statistics. The protocol header inspector, at different layers, are part of this group.
- Those that encompass multiple packets, either seen in sequence (e.g. fragments), or seen across probe interfaces (e.g. Network/Port Address Translation matching), or monitoring bi-directional exchanges, from simple two-way (ICMP Echo, DNS query/response, NTP query/response and SNMP request/response) to complex sessions (TCP connection, FTP session). All impose to preserve specific state information in searchable ad hoc data structures.

Output of one analyser is often required by another. For instance:

- An upper layer packet analyser needs the offset in packet where its protocol header starts; which is computed at lower layer.
- The packet signature cannot be blindly applied to the packet content: some varying protocol fields need to be masked, not only at IP layer but above; moreover, obtaining the same signature for corresponding pairs of a two-way exchange interacts with the corresponding transaction-specialised analyser; even, such signature is not required at all when only global statistics are requested for a flow, and no per-packet observations are to be reconciled by the collector; but it must uniquely identify a datagram or segment that undergoes fragmentation between probing points: signature of the fragment set must be computed the same at every point.
- TCP window scaling option, present in the SYN/SYN+ACK must be taken into account by the TCP connection follower over the entire sequence of further exchanged segments.

State information maintained by the probe plays a key role in the trafMon software tool. Protocol behaviour analysis implies to retain a large variety of information items across subsequent packets. And the reduction of the volume of observations to be centralised implies that the probe maintains numerous counters and global statistics to be regularly transmitted to the collector. For most flows, these global measurements replace the central gathering of per-data unit detailed data, for some others both are transmitted.

Therefore, state data structures permit the probe to organise its measurements. These are organised in simple containers such as the dichotomy search table of the standard C library (qsort/bsearch), custom hash table, or by the efficient home-made dictionary, combining a balanced “btree” with doubly linked sorted list.

In order to adapt to potential large variations of the probe load, following the volume of traffic, the probe processing chain is split in slices with interposed buffers. Some of them are implemented on the basis of a generic circular buffer object with optional copy-in and/or copy-out:

- A large libpcap internal buffer is assigned for kernel mapped passing of pre-filtered captured packets;
- An additional probe circular buffer, built in shared memory, acts as a separator between light input pre-processing of packets (performed by the father process) and its further more in-depth analyses, in the child process;
- Observations accumulated on a per-packet or per-datagram/segment unit granularity, the regular samples of per-protocol statistics variables and those accumulated about individual TCP sessions, FTP sessions and FTP file transfers are then gradually encoded into their respective types of probe observations PDU under construction, to be regularly sent to the collector;
- Prepared PDU units are put in output queue, each assigned to a free time slot, for regulated sending and re-transmission to the central collector. This last queue requires careful handling for implementing the necessary regulation control as presented in section 1.5 above.

Buffer full warning and alerts are important health indicators. Protocol analysers will also detect abnormalities in traffic. These events can be reported in two ways:

- They are transmitted with the recent observations, in order to be stored in the trafMon database, together with the corresponding performance measurements allowing at least an expert user to interpret the measurements in their context.
- They can also be transmitted to an overall health monitoring tool under the form of SNMP traps.

Health monitoring of the probe can be complemented through SNMP polling of its embedded SNMP agent implementing a simple MIB with representative traffic performance counters and statistics on the probe processing activity.

The sequencing of the processing slices as well as the regular timeout checkers are triggered by a fine grain timer utility, with avoidance of pre-emptive interrupts.

The main configuration of the probe lies in an XML file shared with the collector. It is loaded thanks to *libxml2* public domain package. Each software module then learns its own set of parameters and compiles the ad hoc data structures. The father process creates Flow Classes and related Granular Flow data structures corresponding to the XML configuration before forking its child, which inherits those pre-initialised information. An update of the configuration file is detected and induces a cold restart of the probe at the exact time at which the new version becomes valid. Because all online trafMon involved systems need to be precisely time synchronised, the switch of configuration can happen in synchronism over the distributed trafMon components.

## 3.1 INPUT PROCESSING AND BUFFERING

Packets are captured independently at each network probing interface, according to custom-defined packet capture filter. This is achieved through the constant non-blocking polling of *libpcap*.

Modern libpcap version running on current Linux kernel makes use of a kernel ring buffer directly mapped into the user space for avoiding one copy of the captured packets. Although very efficient and with a custom specified size, the internal capture buffer is unfortunately not directly accessible by the client program. The kernel resident packet copies can only be accessed one after the other: each is passed to a callback routine and its buffer slot is released upon callback return.

All stateless protocol decoding work of the probe can be conducted directly on the kernel mapped packet, and the Flow Class matching is also operated in an optimized single traversal of filter predicates. Only the pre-processing data results, for those packets relevant for further analysis and observations reporting, need to be buffered in probe inter-process queue.

## 3.2 FLOW MEMBERSHIP AND GRANULAR SUB-FLOW DISCOVERY

The probe is fed by Ethernet frames, possibly truncated to speed-up the capture: length of actually captured content is custom specified. Only IP packets are subject to capture. The capture is indifferent to the presence or not of VLAN tag. Besides the upper network layer protocol number (IP), the Ethernet header content is ignored.

After protocol headers identification, the packet enters actual analysis chain. This is commonly applied to the input queues of all capture interfaces, which are inspected in full before going to the next one.

The observations and analyses to be conducted is custom specified for user-defined classes of data flows.

Theoretically, the first operation to be conducted on the packet is determining its membership of one (maybe several) user-defined flow class(es) and therein, to which granular flow instance it belongs:

- Custom implementation of criteria testing on protocol headers fields has been preferred to the use of applying a sequence of per-Flow Class BPF packet filters provided by *libpcap*.

  Because a packet can match several Flow Classes criteria, it has been decided to conduct first the full stack protocol dissection, gathering all relevant fields of information about the different layers in a relevant data structure. Then this structure is “sieved”, in a single quite efficient pass, through the combination of all predicates taken altogether from the set of configured Flow Classes respective criteria. This global sieve is organised in such a way that the different protocols fields fully checked one after the other, so as to ensure that the current field value lies in a CPU register as long as it is tested. As soon as a predicate is invalidated, the corresponding Flow Classes are excluded from the list of remaining candidates. After the single traversal, the packet is declared belonging to all the remaining Classes.

- All IP fragments must be mapped to the datagram or segment they belong to.

  IP header fields permit a first screening of flow class and granular flow membership. But UDP or TCP header is not known by the second and subsequent fragments, preventing full discrimination.

  However, the capture interface, the pair of IP addresses and the IP identifier together provide a key for retrieving a short-life reassembly context, dynamically created by the first detected fragment, not necessarily the one containing the transport layer header.

  Individual packet analysis of a second or further fragment can have to be suspended and delayed until the first fragment (containing the transport header) is analysed: determining full granular flow identification.

  Such per fragment analysis is only needed where per-packet individual measurement is requested.

  Complete datagram/segment analysis is, in any case, delayed until either the reassembled data unit is complete or a short-life timeout occurs.

  Such timeout conducts to an incomplete data unit event and, when requested, to the processing continuation of every pending fragment:

  - normal per-packet measurement as soon as the first fragment, with UDP or TCP header, has been received,
  - IP-level only processing (custom selected) otherwise

  In practice, the datagram key, common to all fragments, serves to early skip the further handling of a packet in the father process: as soon as a first fragment occurs, and its Flow matching reveals that other fragments won’t be needed, a short-life corresponding entry is registered in a lookup dictionary. Those subsequent fragments that have been captured before this first will be kept for analysis by the child (and then skipped during stateful reassembly attempt). But most often, the first fragment occurs before, and all following fragments can then be dropped early by the probe father process. This way, only relevant fragments are preserved for reassembly by the child process.

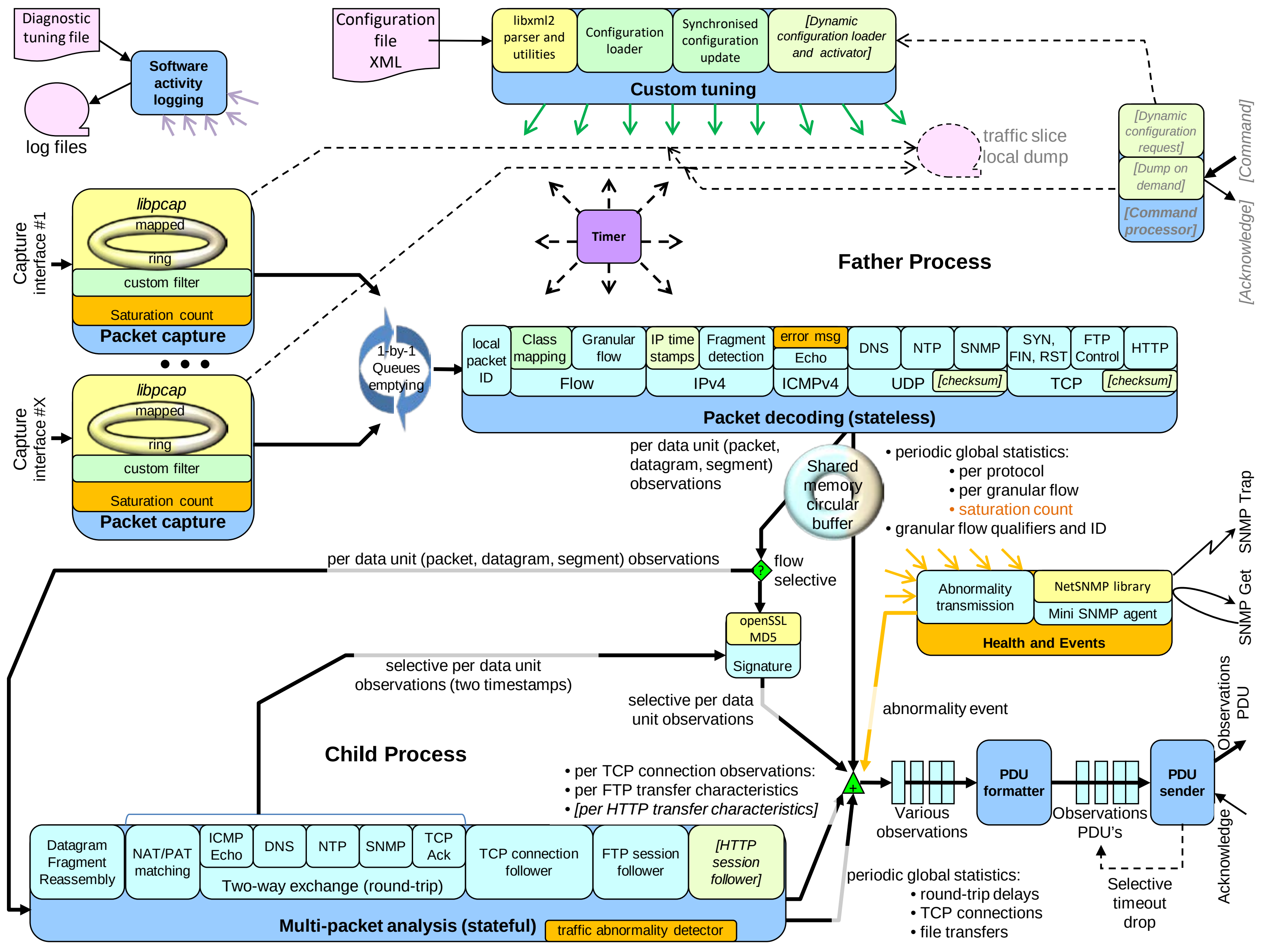

**Figure 4 trafMon Probe Architecture**

- Auto-discovery of granular flow instances is a flexible feature:
  - Each Flow Class defines what identifying fields of the packets are used to define a flow instance.
  - Even for same set of flow qualifiers values, granular flows must be kept distinct between capture interfaces. This allows to avoid over counting a same flow instance seen multiple times. Only those one-way flows for which more than one probe provides partial observations (reconciled in the collector) won't have a probe interface in their identifier.
  - A particular context has to be created under the flow class context. The first occurring member will assign values to the set of custom identified flow qualifiers. Efficient matching mechanism permits to map subsequent packets to their granular flow. No more used granular flow contexts will have to be cleaned after custom-tuned obsolescence duration.
  - The characteristics of a granular flow (values for qualifiers) have to be sent once to the collector, together with a probe-defined identifier, independently of transmission of, possibly intermediate, related observations referring to this identifier.
  - The collector maps the per probe flow identifier to a unique value it generates itself. So that the measurements from all probes end-up with the same identifying number. After implementation*, this choice has revealed to be counterproductive*: Indeed, most of the observed flows are distinct per probe interface (see above); hence their per-probe unique ID is enough. And as long as the flow definition (values of the retained protocol fields) PDU is not delivered to the collector, the observations for the unknown flow must be disregarded by the collector, which cannot map them to its own generated flow ID. Because the mapping ID-to-definition are to be redone within the database (when the collector restarts it restarts assigning new numeric IDs to the re-discovered flows), preserving the per-probe assigned flow ID would limit the data glitch due to collector restart.

## 3.3 SYSTEMATIC DISSECTION OF IPV4/UDP, IPV4/TCP OR IPV4/ICMP PROTOCOL HEADERS

For first (or only) fragments, a systematic dissection of the IPv4 protocol header and of its upper protocol header is conducted, filling a specific dissection data structure.

For ICMP error packets, the embedded packet headers are also dissected (the ICMP error belongs to the data flow at the origin of the error.

This forms the subject to the Flow Class single pass sieve traversal.

## 3.4 EFFICIENT FLOW CLASS SINGLE PASS SIEVE TRAVERSAL

Without limitation of capability, any Flow Class applicability filter can be expressed in up to three layers of Boolean connectives:

- top level 1: **AND or NAND**: as in [NOT] ( P1 AND S2 AND S3 AND P4 )
- intermediate level 2: **OR or NOR** as in S2::= [NOT] ( p5 OR p6 OR s7 )
- bottom level 3: **AND or NAND**: as in s7::= [NOT] ( pr8 AND pr9 )

Predicates are conditions on fields of information extracted from the IPv4/ICMP/TCP/UDP headers.

Thanks to this limitation in depth of the expressions, a sieve structure with all predicates of all filters specified by all configured Flow Classes can be built. A single traversal of this structure where every possible protocol fields are checked one after the other, permits to invalidate candidate Flow Classes as early as possible. Those Flow Classes whose filter remain valid after the traversal are applicable to the packet.

## 3.5 FURTHER DISSECTION OF SERVICE PROTOCOL

The type of analyses defined for the set of applicable Flow Classes implies the application protocol: ICMP Echo, DNS, SNMP, NTP, FTP.

Based on the applicable Flow Classes, the further dissection of the remaining service protocol header is conducted.

And such packet is systematically queue for analysis by the probe child process

## 3.6 CACHE OF FTP SESSION PEERS

When a packet is considered member of an FTP session, its pair of IP addresses is remembered in a cache.

Later, when a TCP packet has been dissected for an unknown service protocol, a check is made to see whether its IP address pair is registered in the cache, together with an important Boolean specifying how to monitor/measure the FTP data connection.

- If only initial and final TCP connection packets are needed for measuring the duration and estimating the payload, as the difference between sequence number at TCP connection boundaries,
    - the cache matching current packet is elected for probe child processing only if it is a TCP SYN, TCP FIN or TCP RST type packet,
- Otherwise
    - the cache matching current packet is systematically elected for child analysis.

## 3.7 ONE-WAY FLOWS EARLY SIGNATURE HASH

When a packet is not fragmented and is subject to one-way measurement by two or more probes, its full datagram content can already been hashed from the kernel-resident data.

In such case, only the signature hash must be transferred to the probe child process, together with its header dissection results.

For one-way (incomplete) fragments, however, the full packet will unfortunately have to be transferred to the child process, in addition with its dissected information.

## 3.8 IPV4 RE-ASSEMBLY IN THE PROBE CHILD PROCESS

The probe child process handles retained per packet information one after the other.

Fragments are saved in their re-assembly context until the entire datagram is re-constructed (or a timeout occurs).

Further analyses are delayed until this happens.

Note that the one-way individual observations can be applied to individual fragments, but also to re-assembled datagram units. In this case, two timestamps are produced: those of the first and last fragments to occur (oldest and youngest timestamps of the collection of IPv4 fragments).

## 3.9 IDENTIFICATION OF NECESSARY STATELESS AND STATEFUL ANALYSES

For each dissected (and reassembled) data unit, the probe child process identifies the upper layer stateless and stateful protocol analyses required by one or more of its Flow Classes: TCP follow-on, FTP session or FTP data connection matching, one-way hop timestamping, round-trip delay, counters statistics

## 3.10 GRANULAR FLOW INSTANCES DISCOVERY AND MATCHING

Then the child process loops over all Granular Flow specifications induced by the set of Flow Classes the packet belongs to. It creates a new flow instance entry in its registry or retrieves the probe flow ID already registered.

## 3.11 PERFORMING STATELESS AND STATEFUL ANALYSES

According to the outcome of the Flow Classes traversal (see 3.9 above), the child then applies the current data unit to the concerned set of analysis in progress:

- Round-trip delay: TCP SYN/ACK or TCP RTTM option, UDP/SNMP, UDP TCP, UDP DNS, ICMP Echo,
- TCP connection stateful follow-on,
- FTP session follow-on of the control connection or FTP data connection follow-on (directory listing or file transfer)

## 3.12 PROBE DATA AGGREGATION

The probe child process makes use of timer to publish aggregated measurements ar the required frequency. These aggregated measurements could be

- all accumulated protocol counters, for the set of discovered flow instances
- the round-trip delays

## 3.13 PROBE EVENTS

When the father probe detects the occurrence of a remarkable event, it passes its record to its child, via the shared memory circular buffer.

The child process gradually builds a PDU for events with those events reported by the father and those detected by the child process itself.

## 3.14 OBSERVATIONS PDU PUBLICATION

There is one “PDU under construction” for each of the following types; more precisely, there is one per PDU type and per destination collector (multiple collectors can be used, that receives the same probe observations independently, in order to support deployment of disaster resistant architecture)

- Flow Instances Description
- Metrics Histogram Slices description: for instance, round-trip delay <2ms, 2-10 ms, 10-50 ms, >50 ms
- Metrics Histogram Slices data
- TCP Connections observations
- FTP File Transfers observations
- Flow Instance protocol counters (delta during one time interval)

- Per Packet/Datagram observations
- Per Packet/Datagram Delay observations
- Probe Events

New observations are gradually filling up the PDU, until it is detected full compared to the configured limit. The PDU then undergoes final encoding and is passed to the transmission module for regulated sending to the collector(s).

This empties the storage for building the PDU, where new data record can then be appended.

Conceptually, to time for sending successive probe PDU units is discrete: cut in successive slots separated by the minimal inter-PDU delay. When a new PDU is to be transmitted, it is placed in the next available slot, or sent immediately if the previous one was already delivered since a long enough time.

PDU's are saved in a data structure associated by a timer, waiting for collector acknowledgement (very small UDP packet containing only two times the PDU ID, to detect transmission errors). An acknowledge PDU is then released.

When a timeout timer triggers for a given PDU, if its maximum retry count isn't yet reached, the new timeout value is computed, the PDU is then assigned the next transmission slot (may be immediately), and the new acknowledge timer is started.

During nominal conditions, the first occurrence of a retry count exhaustion, without acknowledge, induces the entrance of the *long retry mode*. The PDU detected exhausted is then reties at its maximal rate (initial timeout value). Any PDU that has not yet exhausted its retry count is also retried, but at minimal rate (maximum timeout derived from maximum retry count, increment and multiplier attributes). The PDU with important definitions (flow, metrics histogram slices) or observations (individual FTP file transfers) are similarly slowly retried at maximum timeout. And same slow retries occur for PDU units whose observations content belongs to a fixed time window (*break border time* parameter) just before the start of the long retry mode (just before the border of the break of connectivity with the remote collector). This last permits fine grain traffic behavioural analysis leading to a potential problem of the network link.

A first reception of collector acknowledge cancels its long retry mode condition. and nominal transmission behaviour is re-installed.

When no PDU have to be delivered for a while, empty heartbeat PDU is sent, so as to indicate to the collector that the probe is still alive and accessible.

# 4. SOFTWARE ARCHITECTURE OF THE COLLECTOR

For all but the one-way observations PDU types, the current collector barely acts as a PDU decoder and value printer to the corresponding output ADCII log file. However, it is structured in a way allowing interposition of further processing (e.g. computation of specific metrics KPI, threshold comparison for real-time detection and warning on traffic abnormalities).

An early implementation choice has been made for the flow description by the probe, which reveal in fact counterproductive. Each flow instance is assigned its unique numeric flow ID by the originating probe. The collector assigns a collector-unique numeric flow ID to replace that from the probe, in order to keep distinct IDs across all probes. But all probe observations, besides the one-way measurements, are individual, per probe interface. So the probe flow ID is already unique per probe ID. The drawback of the current implementation is that, when restarting a collector, all probes must be restarted, or must resend their flow descripts as soon as possible. all observations data from a probe, pertaining to a collector-unknown flow ID, must be dropped until the collector receives the corresponding flow description.

## 4.1 PDU RECEPTION PROCESSING

The PDU reception mechanism adapts to variable length UDP data gram units. First, a non-blocking check is made on the socket, to see whether there are pending data or to learn the size to be read all at once after dynamic memory allocation.

Every PDU has a CRC code. Sanity checking is made, then the PDU is acknowledged (except for empty heartbeat). The probe alive time is refreshed.

When the PDU hasn't been already received, it is queued in a collector input circular buffer.

For one-way partial observations, it was initially thought to implement a special mechanism to handle arrival of late PDU from a probe whose connectivity was broken during a while. Indeed, when partial observations are still missing after a timeout, if this probe is detected silent at that time, the validity of the partly reconciled packet observations could be enlarged, waiting for late PDU. Those could then bypass input queueing for immediate processing. Finally, the queueing doesn't really lapse, so such a queue bypass wouldn't reveal useful.

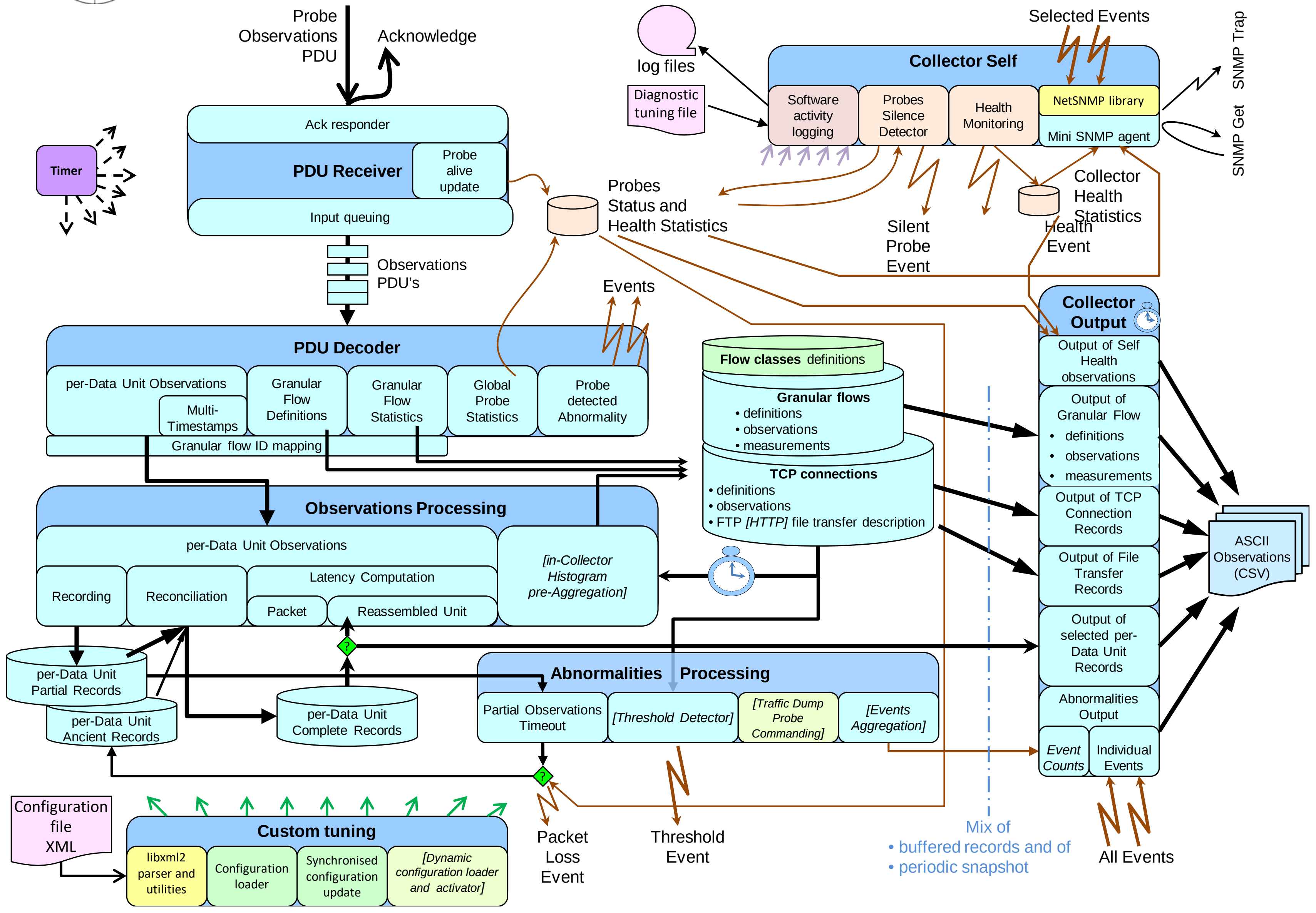

Figure 5 trafMon Collector Architecture

## 4.2 PDU DECODING AND OUTPUT OF SINGLE-PROBE OBSERVATIONS

As said above, all specific collector processing of observations, except for one-way flow measurements, hasn't been implemented (bracketed italics modules in the figure).

So, after PDU decoding, all types of single probe observations are simply appended to the corresponding type of observations output ASCII log file.

## 4.3 ONE-WAY RECONCILIATION OF PROBE PARTIAL PER-PACKET OBSERVATIONS

For partial observations, the handling is quite more sophisticated.

Due to the quite compact PDU encoding, the decoding algorithm is rather intricate (see the Detailed design document or the fully commented source code for explanations).

Per packet records (with flowID, packet signature, size and timestamp(s)) are reconstituted one by one from the decoding.

An attempt is made to retrieve the collector record for the corresponding packet, which serves to gather the set of timestamps reports by the several concerned probes. Because the few bytes of content signature risks to collide with that of another packaet for the same Flow Class/Flow Instance, we need to find the best match in terms of timestamps: the packet whose reference time (seen by another probe) is the closest to that of the new partial observation record.

For this, a special kind of dictionary has been implemented. It is made of BTree (balanced 2-3 tree, where each tree node has 2 or 3 children) whose leafs are anchors of a doubly linked list of the dictionary elements, maintained in increasing key order. This list allows easy travel, from the head/tail or from any element, and easy insertion/extraction.

For our per-packet (per datagram) observations, the key also embeds the oldest timestamp known timestamp of the record to search. Hence, most of the search won't lead to an exact match (same packet timestamp at different locations of the network traversal path). But the BTree search routine produces either the matched leaf, or the one either at right or at left. So, in case of an approximate match, we can get the element of the dictionary which is the closed to the search key (also embedding a timestamp value).

Upon a signature clash, we can then match the records with the closest known timestamp. Not only do we then detect the signature clashes, we are tolerant to them and continue to consolidate individual sets of per packet observations for different packets that exhibit the same signature hash value.

Per packet (per datagram) observations are gradually complemented with partial records from the several probes on the path. Note that a partial observation can consist in one or several timestamps, and the packet/datagram size. Timestamps can be

- the capture time at a probe interface,
- the value of an entry in the IP Timestamp Option from the IPv4 header,
- the value of a specified (type of) NTP timestamp extracted from the NTP packet content.

Alternative paths (routes) in the network travel of a data flow are supported by giving the same hop name to different timestamps (typically different probe capture interfaces at alternative nodes in the network.

When all specified hop timestamps have received a value, the packet observations is completed. Either the record, with a list of timestamps, is produced as-is in the output, available for further custom processing (database or other data handling), or the latency is requested between two identified hops. In this case, the collector directly computes the one-way delay and maintains a set of latency histogram slices, which are output and reset at regular time intervals.

The partial observations dictionary is also regularly traversed for finding ancient incomplete records.

- If a missing observation has lapsed the maximum lifetime inside a probe, the probe would have already dropped it from its PDU retry queue. Hence the collector declares the one-way record has "*dropped*".
- If the expected probe for missing observation has been silent (maybe due to loss of connectivity) around the time it would have produced the timestamp(s) and sent them to the collector, the observations is kept waiting for late arrival of still missing observations.
- If the missing timestamp(s) is(are) at the end of the ordered hop list, the obsolete record induces the declaration of a "*packet lost*".
- Otherwise, for a reason or another, we have observations for the packet (datagram) at the end of its travel but not at the start or an intermediate hop. The record is then declared "*incomplete*" or "*partly missed*".

Either each of those three type of one-way incomplete hops are output individually, each in their log file corresponding to the exception type (lost, dropped or missed), or these exception counters are aggregated by the collector.

In case of collector aggregation, because observations can finally become complete after a delay longer than the period of reporting aggregated one-way statistics (latency histograms and/or exception counters), successive slots of aggregation are maintained in the collector: each slot represent one reporting interval. Those slots form a time window that ends with the still open interval containing “now”. So the consolidated one-way observations are output with a delay of several slots.

# 5. REGULAR DATABASE LOADING AND AGGREGATION

The output of the collector, as well as the extracted NetFlow records from the optional SiLK add-on, consists of ASCII log files forming a flexible boundary between the online and offline trafMon function.

## 5.1 REGULAR DATABASE UPDATES (EVERY 10 MINUTES)

However, due to the continuous generation of observations, among others, pre-aggregated counters at 1 minute granularity or below, there is a need of quite regularly

- loading the individual description records (TCP connections, FTP transfers),
- loading counters in temporary database tables, visible only by the loading process which also determine the end of their lifetime; for those records that participate to the same one minute interval for which aggregate values already exist in the permanent table, the one minute records are update, and for those new values, the one-minute records are simply  appended,
- splitting the long duration optional NetFlow records into successive slots of one minute intervals, which can update the table as described here above.

Before first data for a new day are loaded in a persistent table, a new partition is created for that day. Normally, the last partition, called *pFuture*, is always kept empty.

The structure of temporary and persistent tables is defined in the `*trafMon_template*` MySQL database, which also contains the trafMon stored procedures and the protocol table defining the known application service TCP and UDP port numbers, and their optional precedence.

When the target MySQL database (named `*trafMon*` by default) doesn't yet exist, it is created, then each persistent destination table is also created when initial data are available for it.

After the persistent table at 1 minute is updated, it is used to update that at 1 hour. Then the 1 hour persistent table is used for updating the head of the table at 1 day granularity.

After such process (*trafMon_loader.py*), the non-resolved IP addresses (including the newly discovered ones) are subject to mapping, either in terms or Activity/Location, or through GeoIP, and a reverse DNS lookup is attempt (*trafMon_updateIpInfo.py*).

Following tables are aggregated at 1 minute, 1 hour and 1 day:

- IPv4 counters,
- IPv4 sizes distribution (histograms),
- ICMPv4 counters,
- UDP counters,
- TCP counters,
- FTP counters,
- Two-way round-trip delays (histograms),
- One-way latencies (histograms),
- One-way counters

Following are definitional tables:

- Flow instance descriptions,
- histogram slices definitions of metric instances,
- or one-way hops definitions (optional)
- trafMon probes and collector events

Following tables contain individual data records:

- TCP connections,
- FTP transfers,
- one-way hops timestamps (optional)
- and, separately, one-way lost, one-way missed, one-way dropped counters (optional)

After merging, the original data log files are archived in a compressed tar file.

## 5.2 PREPARATION OF TRAFFIC VOLUMES DATA WITH ACTIVITY/LOCATION/GEOIP AND DNS NAMES

Once a day, the volumes of passive FTP data connections are merged with those of corresponding FTP session flows. Then the aggregation per Activity/Location/peer Location or Country is conducted.

Once a day, but later, the same aggregation is performed on NetFlow data.

So the synthesis reports are up-to-date up to yesterday.

## 5.3 FULL UPDATE OF IP ADDRESSES (WEEKLY)

Once a week or less, preferably at a quiet period at night, the full set of DNS names are queried again. This is done in successive batches, but generates bursts of DNS traffic by the trafMon central server (*trafMon_updateIpInfo.py --all*).

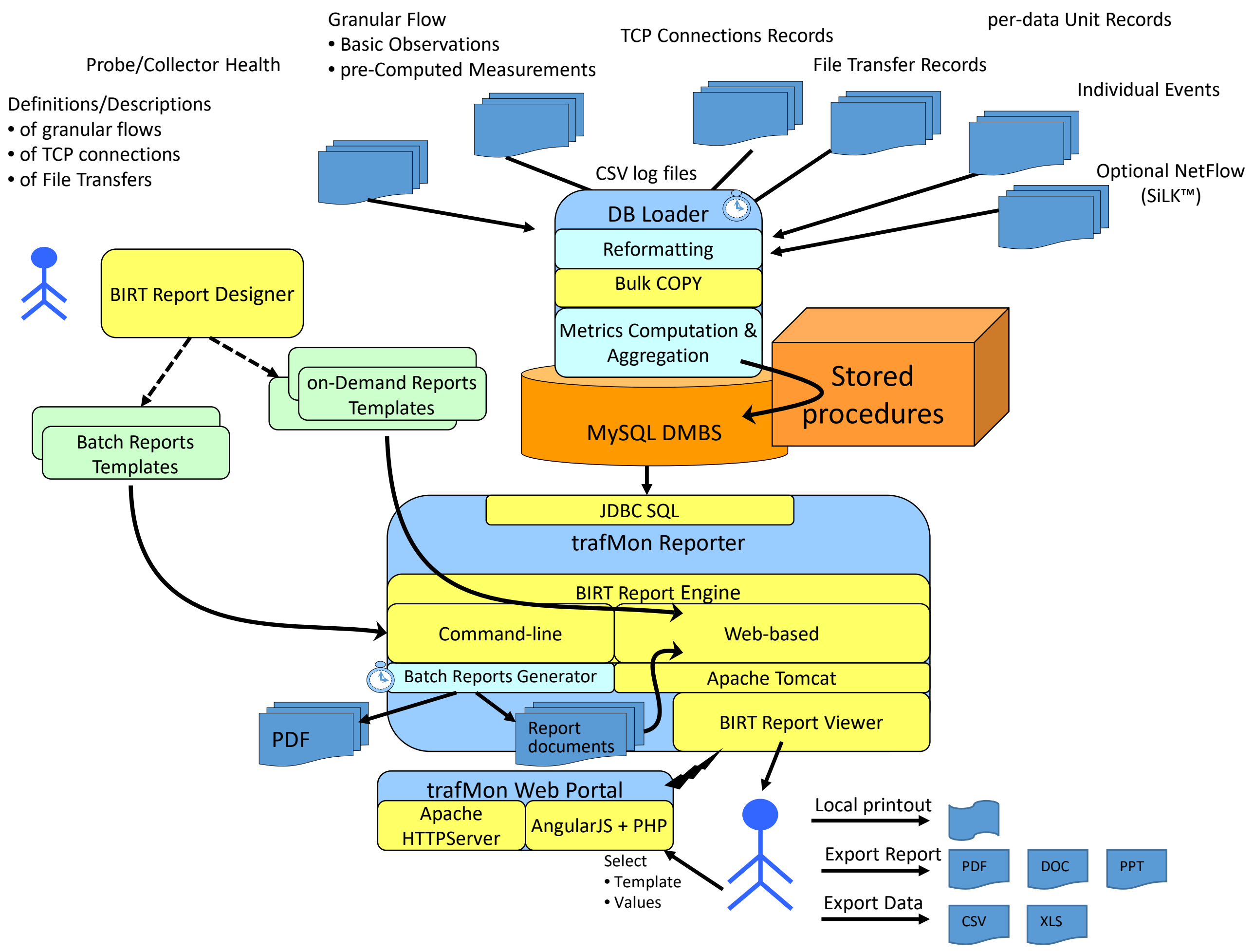

**Figure 6 Database and Reporting Architecture**

# 6. REPORTING FUNCTIONS

Once the data are available in persistent tables, any reporting tool could be used for presenting views on the data.

Based on HTML5, there exist quite powerful data visualising libraries, such as the d3.js open source JavaScript graphical library. But we wanted to rely on a free tool that permits easy and flexible design of report templates, mixing textual, tabular and charting representations, whose output would be suited for Web browser for on-demand drill-down as well as supporting the automatic generation of electronic report instances.

For these reasons, we selected to rely on Eclipse/BIRT (https://www.eclipse.org/birt/).

The BIRT Designer, run as a perspective in the Eclipse Integrated Development Environment, or available as a standalone application, allows a non-expert user to draw or update report templates through a relatively intuitive GUI, although sometimes surprising in what concerns the WISIWIG mapping from design view and laid down report, especially when pagination (for PDF documents) is applied.

Anyway, reports can be designed on Linux or Windows systems (with some issues concerning font equivalence e.g. Helvetica vs Arial).

A BIRT Runtime Engine is then able to apply a custom report template, with assigned attribute values for its parameters, to SQL queries retrieving the underlying data from the database and laying down an actual report instance, either as a multi-page PDF document, or in HTML. The BIRT Engine is a Java EE application that can either run as a command, producing report in batch mode, or interactively as part of the Java EE application server. In this case, either the report is produced as a continuous single Web page, with attribute values assigned to report parameters via the URL encoding. Or it can involve the BIRT Viewer Web application, which produce and displays the paginated report and allows printing or exporting the PDF or some of the underlying data sets.

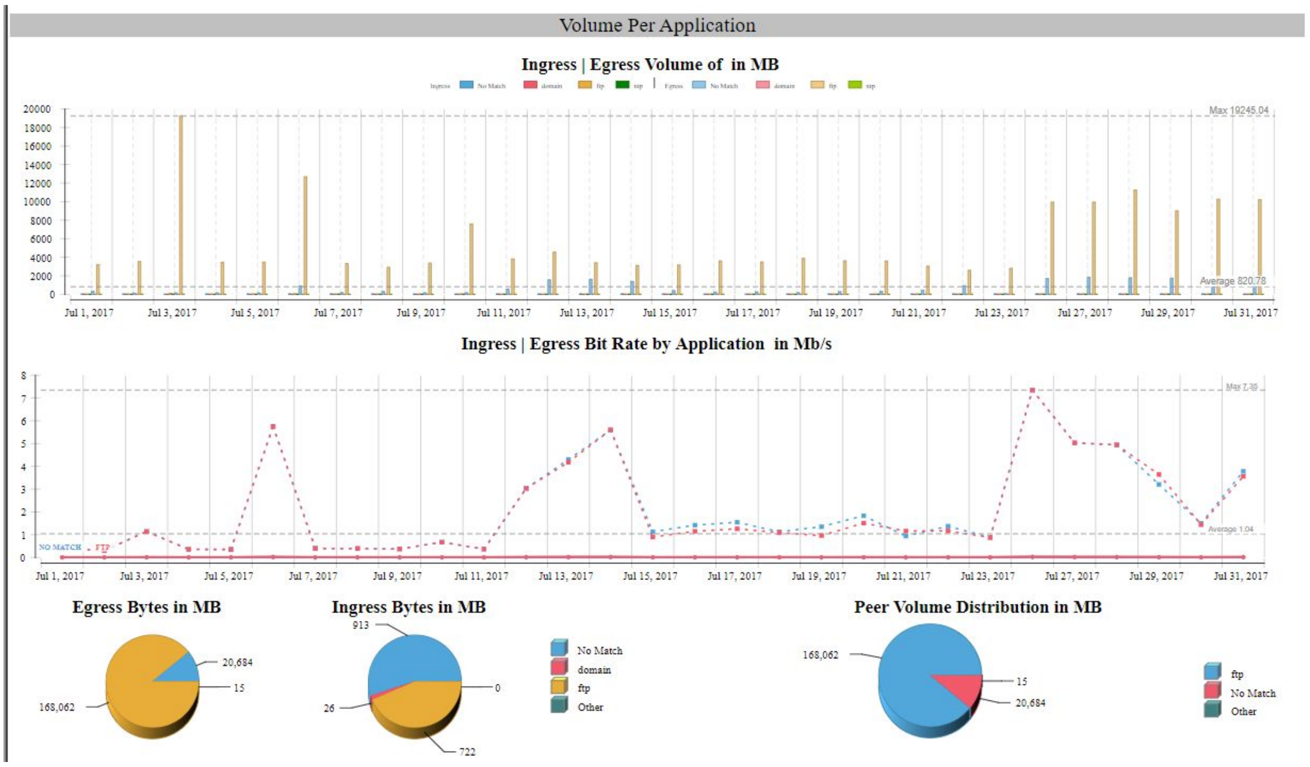

**Figure 7 Excerpt of a trafMon *Manager* BIRT Report**

A set of pre-designed BIRT report templates have been produced, with standardised arguments and some tricky JavaScript. Anyway, the user is able to adapt them or to create new ones though the Eclipse BIRT open-source Designer.

Those reports are foreseen for installation within the BIRT Runtime Engine and Report Viewer J2EE application in Apache Tomcat server environment.

For simplifying the selection of parameters for on-demand report generation, a basic JavaScript/AngularJS/JQuery and PHP Web application has been produced for the Apache HTTP Server, which displays a dynamic top-screen menu bar and launches the Tomcat/BIRT report generation for display in the frame below.

Two example scripts, respectively for synthesis reports and for protocol details reports, are also provided that automatically generate PDF report files in batch mode.

# 7. DEPLOYMENT ARCHITECTURE

The figure below depicts a typical deployment architecture of the trafMon components.

The underlying model is that of an Organisation being spread over two or more production sites (could be buildings or even rooms), where the Local Area Network segments are structured as internal LAN and Demilitarised Zone (DMZ), with server typically providing services to external users, over the internet, and to staff at other sites. With two sites, this typically leads to four know ‘*Locations*‘:

- Site 1 Internal,
- Site 1 DMZ,
- Site 2 internal,
- Site 2 DMZ

Also part of the model, while not visible in the topology diagram, is the partitions of the computer systems according to the different ‘*Activities*’ which structure the Organisation, e.g.: Administration (ADM), Engineering (ENG) and Production (PROD). For instance, this leads to the following partitioning of IP addresses:

- @ Site 1 Internal: ADM hosts and ENG hosts,
- @ Site 1 DMZ: ADM servers and PROD servers,
- @ Site 2 Internal: ADM hosts, ENG hosts , and PROD hosts ,
- @ Site 2 DMZ: ENG servers and PROD servers.

In order to cover the monitoring of all data flows, we need to capture the incoming/outgoing traffic over the connection of all four LAN segments with their local firewall. This is represented by the probe “lasso”, which is materialised by the plug of a probe computer monitoring port (typically without IP address) to a span port (or mirror port) of the LAN switch, where the required packets are replicated. It could also be achieved through the use of Ethernet tap devices, but note that supporting pure passive tap devices (where IN and OUT traffic are replicated to two separate physical mirror ports) would require a slight change in the probe software, to preserve the respective order of occurrences of packets in both directions while analysing the protocol exchanges.

A same traffic probe can capture traffic via two or more monitoring interfaces, as for probe3 below. But due to the processing load and computer hardware dimensioning constraints, it can be sometimes more appropriate to use one probe device per probing interface, as with probe1 and probe 2 in our example.

For observations derived from protocol analysis, the probe unique interface name is part of the name of each detected data flow instance. This avoid problems when a same flow is seen at different probing points, which is then appropriately handled upon aggregation of observation in the central database.

Some flows can also be monitored for their one-way latency and packet loss metrics. For instance regular SNMP requests from a Site 1 internal computer towards the switch of Site 2 DMZ. In this case, probe1 reports the source timestamp of the flow packets, together with their per-packet identifying MD5 signature hash; and probe2 reports the destination timestamps and packet signature. Both probes refer to the same one-way flow identified by '*srcIP:high>dstIP:161*', without probe interface identifier in the name.

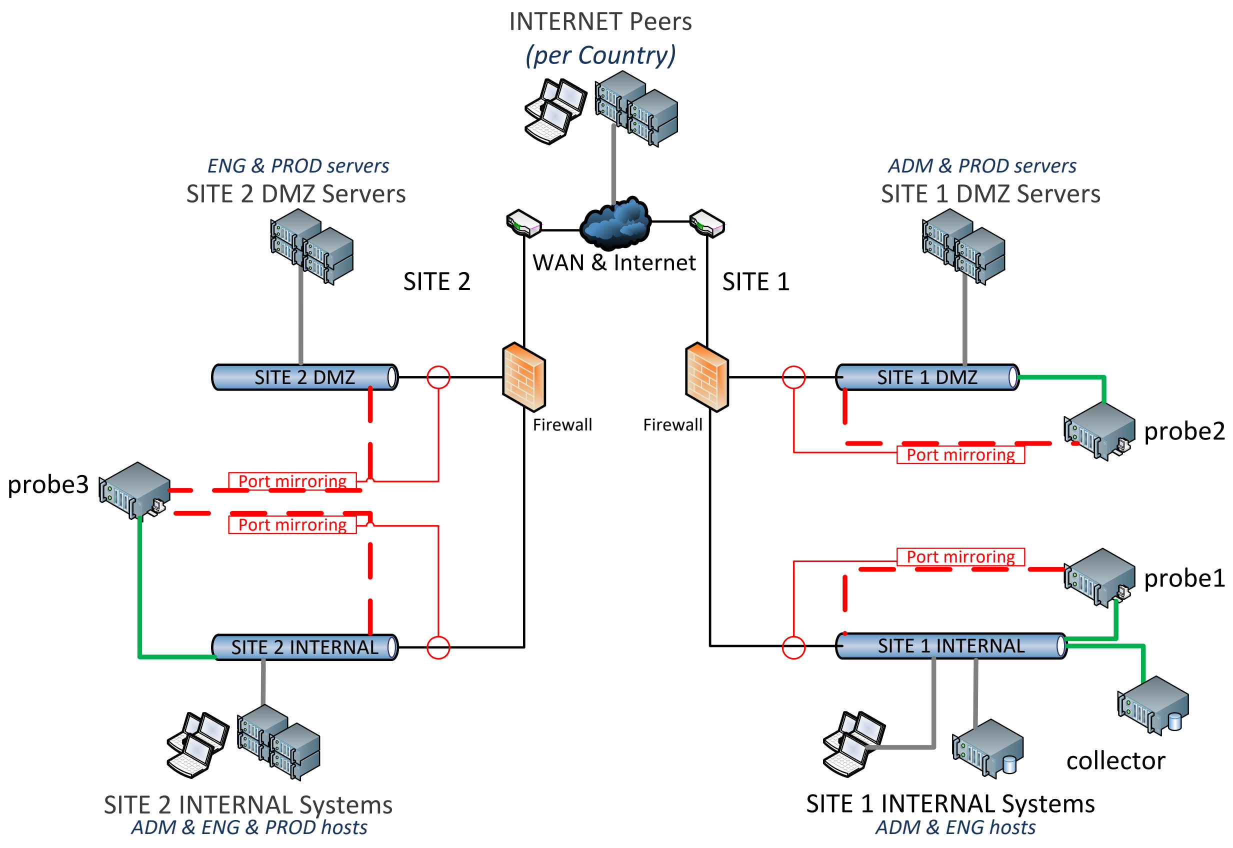

**Figure 8 Typical Deployment Architecture**

At a convenient place in the network, typically on one side of the WAN data-link or at the centre of a star shaped topology, a Central Processing system gathers the probe observations. It runs the online trafMon collector.

The MySQL Database is typically co-located on this central server, where all the raw and pre-aggregated data are regularly stored and updated. Stored procedures are also scheduled on a daily basis for the further aggregation implied by synthesis views, in terms of Activity/Location/Host and peer Internet Countries

The Eclipse BIRT database reporting tool, with its runtime Engine for batch PDF reports generation, and its Apache Tomcat interactive runtime Engine can also run on the same central server, together with the Apache HTTP server and the trafMon menu bar Web application.

# 8. ADMINISTRATION AND MAINTENANCE

## 8.1 UNPRIVILEGED TRAFMON ACCOUNT

Thanks to Linux setcap, it is possible to run the `tmon_probe` under an unprivileged account (e.g. `trafmon`) member of the group (e.g. `trafmon`) for which the capabilities are applied.

The `tmon_collector` does even not need privileged capabilities, provided it has write access to its data logs output directory.

Both components generates diagnostic trace logs (under `/var/log/trafMon/` or elsewhere) and both must be able to rename their new update of XML configuration file to overwrite the previous one (under `/etc/tmon/xml/`).

## 8.2 TWO ACCOUNTS FOR MYSQL TRAFMON DATABASE(S)

A MySQL account (e.g. `tmon_db`) has full permission on all databases whose names are starting with ‘`trafMon`’ as a prefix.

Another (e.g. `tmon_birt`) has only read-only and execute-only rights. For this user, the MySQL password should be implicit when executing `mysql` command-line locally. This is used by the `trafmon` Linux account for scheduling database aggregation or clean-up maintenance tasks: thanks to a read-only ~trafmon/.my.cnf protected file containing the password for the `tmon_birt` database user.

## 8.3 LOGROTATE

By convention, all log files generated by the trafMon programs and scipts are names with the ‘`.log`’ suffix and placed in a same directory (e.g. `/var/log/trafMon/*.log`).

A sample logrotate file for trafMon is provided with the distribution.

It must be invoked every hour through the `root` account `crontab` file <u>on probe and central servers</u>. It ensures that the Linux account:group `trafmon:trafmon` stays owner of every log files (archived and runtime).

## 8.4 MYSQL SERVER AND TOMCAT DAEMON

On the central server, the `root` account `crontab` file should run a script, every minute, that detects the absence (crashed or not yet started) of either `mysqld` or `tomcat` service and automatically relaunch them upon need.

Also, Tomcat could quickly become thick in virtual and resident memory, after generation of several reports. Hence this `root` account `crontab` should also systematically restart the `tomcat` service every night at a quiet time, in order to recuperate memory.

## 8.5 TRAFMON SCHEDULED TASKS

Both the `tmon_probe` and `tmon_collector` programs could be started manually (e.g. within the virtual connection environment provided by the `screen` utility). But in production environment, it is important that those components are re-started as soon as crashed, to minimise the impact of of glitch in the observations. So the `trafmon` account `crontab` file can be used, respectively on the robe and central servers, to detect absence and automatically launch the corresponding trafMon online program automatically. This corresponding script must be run every minute.

On the central server, the `trafmon` account `crontab` file is used to also schedule the several necessary database update and cleanup tasks and, when required, the generation of PDF reports.

A sample `crontab` file, with comments, and corresponding scripts are provided, for the two types of systems, as part of the trafMon distribution.

The generation of protocol counters reports induce the creation of working tables that cannot be made temporary. The MySQL stored procedure `` `trafMon_template`.`Drop_working_tables` ``() destroys them from all candidate trafMon database instance.

Another MySQL stored procedure `` `trafMon_template`.`Drop_working_tables` ``(*DBname*, *tablename*, *daysToKeep*) permits to selectively remove obsolete partitions from those fast growing granular data tables. After some experimentations, the user can carefully schedule such automatic clean-up of selected database tables, to avoid that the database explodes.