An open source network traffic performance monitoring and diagnostics tool.

www.trafmon.org

Detailed Design

Thomas Grootaers, Luc Lechien

Software Release 1.0

2020-10

COPYRIGHT, LICENSE AND TRADEMARKS

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.

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 SOFTWARE STRUCTURE

Although compatible with other versions and different Linux distributions, the trafMon software has been developed for the Linux CentOS 7.x operating system, which is the open-source correspondent of RedHat Enterprise Linux (RHEL) 7.x.

1.1 OVERVIEW OF TRAFMON SOFTWARE COMPONENTS

The trafMon software consists in an on-line part, which continuously analyses the monitored traffic its captures, and an off-line part, which consists in batch measurements loading and metrics computation and aggregation in relational database and in a report generation function.

The *on-line functions* are, by nature, distributed.

The **probe program** materialises the points where the monitored traffic is captured. This program runs at the several monitoring locations. It conducts protocol dissection, filtering and categorising of traffic flows, whose instances are dynamically discovered; some flows are simply reported as individual packet observations, others are feeding statistics counters that are regularly reported, other serve to measure 2-way round-trip delays, other feed full-stack stateful protocol analysis, following the evolution of TCP connections, the command/response dialog of FTP control sessions and producing measurement records of application-level transfer of files.

The **collector program** centralises live observations and measurements from the probes, typically at one location. However collectors can be replicated at different sites; for redundancy reasons or for separately feeding their local monitoring centres. Most of the probes measurements are sufficiently complete and compact to be simply output by the collector. Others require further aggregation processing and metrics production, as the one-way latency and counters of packet losses/partial/dropped. While the granular per-packet partial observations, produced by different probes, are merged before collector output or measurement processing.

Although offline – i.e. not directly connected with the monitored network – the *offline functions* are continuously fed, in near real time, with the results data files produced by the collector.

The **database loading, metrics computation and aggregation** is a script launched periodically, at a period in the order of few minutes to a quarter. It pre-loads all available collector output data into temporary tables. These data serve to selectively update and complement the recent top of the several corresponding metric aggregates, at different time scales. Then the fine grain raw data persistent tables are complemented with the newly loaded information.

A data ageing mechanism, also run regularly, permits to keep the size of persistent raw data and of granular aggregates at acceptable level, by destroying ancient records when

they fall out of the per table defined time window span. This mechanism is implemented in an efficient way by playing with the partitioning property of the concerned MySQL tables into successive time range physical chunks. Cleaning out old data from a table consists in dropping an obsolete partition.

The reporting functions rely on the BIRT public domain tool running on Apache Tomcat and extracting the relevant measurements via ODBC SQL queries executed on the MySQL database.

Several report templates are pre-designed. Those report templates can be run in batch mode to produce reports over fixed time frames, typically as PDF electronic documents.

Those report templates can also be run dynamically via a Web interface, with custom selection of report parameters – typically time boundaries – and through navigation of drill-down sub-reports accessible via hyperlinks. This way, each report figure is produced on-the-fly, implying to wait for execution of database queries and supplemental BIRT reporting computations.

To facilitate the interactive drill-down navigation through the reports and data, a basic Web application presents a dynamic menu bar at the top of the browser window. This is built in JavaScript with AngularJS and JQuery, and is linked to the database via PHP scripts.

While the Web Runtime Engine of BIRT runs as an Apache Tomcat application, the trafMon Menu Bar is served by the Apache HTTPD daemon.

1.2 SOFTWARE CODE STRUCTURE

The trafMon software and configuration samples is a GIT repository. In a first time it won't yet be published on GitHub, before the available lifecycle support tool for open source community are fully understood to permit contributors to participate in an orderly way.

The on-line function are coded in C language. The source code has been written using uniform conventions, keeping in mind that the code must be easy to understand by any newcomer (*comments are felt more important than a perfect code*). Care has been taken to control the memory boundaries and to never trust any portion of the network packets, especially those under inspection. This source code is entirely located in the three following source files sub-directories: **`tmon_core`**, **`tmon_probe`** and **`tmon_coll`**.

```
$ ls -A trafMon
.git            DIST_FILES.sh  NOTICE          tmon_core    trafMon_reports
.gitattributes  INSTALL        README.trafMon  tmon_probe   trafMon_scripts
.gitignore      ISSUES         tgen            trafMon_web
CHANGES         LICENSE        tmon_coll       trafMon_etc
```

The files (schema, database users and permissions, sample configuration files, Python scripts) related to the database, the data loading and the qualificaton of IP addresses are stored in the sub-directory **`trafMon_db`**.

The set of trafMon pre-designed BIRT report templates are stored under the sub-directory **`trafMon_db`**.

The Web application implementing the interactive menu bar for selecting report generation parameters consist in a mix of AngularJS+JQuery view and control JavaScript files and of PHP files in a hierarchy under the sub- directory **`trafMon_web`.**

Several complementary scripts (Python and bash shell), for the batch generation of PDF reports, for optional extraction of SiLK™ NetFlow records, for maintaining a production trafmon system and for the generation of controlled traffic patterns are stored under **`trafMon_scripts`**.

A more specialized C program that permits to implement a cycling over a precise scenario of individual UDP datagrams and TCP pseudo-packets is provided under **`tgen`**.

Several samples of configuration files for setting up a production trafmon system are stored in the sub- directory **`trafMon_etc`**.

A set of text files with upper case names are implied by the open source software distribution and the choose Apache2 License.

The DISFILES.sh script permits to copy the non-binary architecture-independent files from the source tree to a distribution package structure.

NOTE: currently, the source code and files are **not** adapted for the use of **`Autotools`**. Contribution for this would be welcome.

1.2.1 C Programs

The directory **`tmon_core`** contains those modules shared between the `tmon_probe` and `tmon_collector` programs.

```
$ ls tmon_core
Makefile            tmon_circ_buf_tst.c  tmon_diag.h      tmon_metric.c
libtmon_NO_SNMP.a   tmon_circ_pbuf.c     tmon_dict.c      tmon_metric.h
libtmon_SNMP.a      tmon_circ_pbuf.h     tmon_dict.h      tmon_sharedmem.c
tmon_PDU.c          tmon_config.c        tmon_dict_tst.c  tmon_sharedmem.h
tmon_PDU.h          tmon_config.h        tmon_event.c     tmon_snmp.c
tmon_btree.c        tmon_core.h          tmon_event.h     tmon_snmp.h
tmon_btree.h        tmon_delay.c         tmon_hash.c      tmon_statistics.h
tmon_circ_buf.c     tmon_delay.h         tmon_hash.h      tmon_timer.c
tmon_circ_buf.h     tmon_diag.c          tmon_hash_tst.c  tmon_timer.h
```

For the specific case of `tmon_snmp.c`, conditional compilation permits to compile the actual net-SNMP sub-agent related routines, or to replace them by empty stubs:

```
% cd tmon_core
% make clean
% make "COND_SNMP=SNMP"
```

or

```
% cd tmon_core
% make clean
% make "COND_SNMP=NO_ SNMP"
```

Once the right library file has been made, you can build the `tmon_probe` and `tmon_collector` with the same conditional make.

The probe-specific source modules are under **`tmon_probe`**:

```
$ ls tmon_probe/
Makefile                    tmp_child.h          tmp_pktinfo_dump.h
TRAFMON-PRB-MIB.txt         tmp_delay.c          tmp_publish.c
myProbe.diag.sample         tmp_delay.h          tmp_publish.h
singleProbe_sample.xml      tmp_event.c          tmp_reassembly.c
tmon.dtd                    tmp_event.h          tmp_reassembly.h
tmon.xml_pcapFile_sample    tmp_flowclass.c      tmp_snmp.c
tmon_probe.c                tmp_flowclass.h      tmp_snmp.h
tmon_probe.diag             tmp_flowfilter.c     tmp_statistics.c
tmon_probe.h                tmp_flowfilter.h     tmp_statistics.h
tmon_probe_NO_SNMP          tmp_granularflow.c   tmp_tcpconnection.c
tmon_probe_SNMP             tmp_granularflow.h   tmp_tcpconnection.h
tmp_aggregate.c             tmp_interface.c      tmp_transmission.c
tmp_aggregate.h             tmp_interface.h      tmp_transmission.h
tmp_analyse.c               tmp_pkt_dissect.c    tmp_udptransaction.c
tmp_analyse.h               tmp_pkt_dissect.h    tmp_udptransaction.h
tmp_child.c                 tmp_pktinfo_dump.c
```

This directory also holds a sample production version and a typical testing versions of the diagnostics trace tuning (resp. `myProbe.diag.sample` and `tmon_probe.diag`), the SNMP custom MIB definition implemented by the conditional SNMP sub-agent within the probe (`TRAFMON-PRB-MIB.txt`), the fully commented XML DTD file with the syntax for the trafMon runtime configuration tuning (`tmon.dtd`) and two sample trafMon XML configuration files: one for re-playing a packet capture file (`tmon.xml_pcapFile_sample`), and one example with a single probe but with two capture interfaces (`singleProbe_sample.xml`).

The collector-specific source modules are under **`tmon_coll`**:

```
$ ls tmon_coll/
Makefile                  tmc_flowinstance.h  tmc_snmp.h
TRAFMON-COL-MIB.txt       tmc_metric.c        tmc_statistics.c
myCollector.diag.sample   tmc_metric.h        tmc_statistics.h
tmc_delay.c               tmc_output.c        tmc_transmission.c
tmc_delay.h               tmc_output.h        tmc_transmission.h
tmc_event.c               tmc_pdu_decoder.c   tmon_collector.c
tmc_event.h               tmc_pdu_decoder.h   tmon_collector.diag
tmc_flowclass.c           tmc_probes.c        tmon_collector.h
tmc_flowclass.h           tmc_probes.h        tmon_collector_NO_SNMP
tmc_flowinstance.c        tmc_snmp.c          tmon_collector_SNMP
```

This directory also holds a sample production version and a typical testing versions of the diagnostics trace tuning (resp. `myCollector.diag.sample` and `tmon_collector.diag`), the SNMP custom MIB definition implemented by the conditional SNMP sub-agent within the probe (`TRAFMON-COL-MIB.txt`).

1.2.2 Diagnostics Messages Logging

The diagnostics tracing verbosity of every C software module file can be independently tuned in the `.diag` configuration file. For a simple message printing at a given severity level (`fatal, error, warning, trace0, trace1, trace2` or `trace3`), the code contains the following macro invocations:

```
#ifndef NO_SANITY
  if(!pktInfop || !pkt) {
    FATAL "INTERNAL: NULL argument pointer to routine" END;
  }
  if((pktInfop->pktHighestProtocol & isUDP) != isUDP) {
    ERR "pkt 0x%016"PRIx64" WRONG PACKET: Attempting to dissect NTP"
                                  " on non UDP packet", pktInfop->pktID END;
    return(0);
  }
#endif /* NO_SANITY */

…

      if((aTimeT < -4000) || (aTimeT > 4000)) {
        aTimeT = secs; /* uint32_t -> time_t for localtime argument */
        (void)strftime(str, 29, "%T", localtime(&aTimeT));
        WARN "pkt 0x%016"PRIx64" %s Client NTP SKIPPED:"
             " Orig time %s (%u sec) too far from packet capture time(%d)",
             pktInfop->pktID,
             TmpPktInfoIPv4Addrs2Str(pktInfop->pktIpInfo.ipSrcAddr,
                                     pktInfop->pktIpInfo.ipDstAddr,
                                     errMsg, STRMAX),
             str, secs, capTime END;
        return(0);
      }

…

      if( pktInfop->pktNtpInfop->ntpOrgTm.tv_sec == 0 ) {
        TR1 "pkt 0x%016"PRIx64" NTP Request SKIPPED: NULL Orig. Timestamp",
                                                        pktInfop->pktID END;
        TR3 "MEM: free'd %p: SKIPPED", oneWayp END;
        free(oneWayp);
        return;
      }

…

  TR2 "pkt 0x%016"PRIx64" SNMP %s Round-Trip %s - %s DELAY %d.%.06d",
      reqPktp->pktID,
      (reqPktp->pktSnmpInfop->snmpType == SNMP_INFORM)? "INFORM": "Request",
      TmonGMTimeToStr(&(reqPktp->pktPcapHdr.ts)),
      TmonGMTimeToStr(&(rspPktp->pktPcapHdr.ts)), delta.tv_sec,
      delta.tv_usec END;

…

START_TR3
  switch(pktInfop->pktNtpInfop->ntpMode) {
```

```
    case NTP_MODE_CLIENT: /* This has the second timestamp */
      typeStr = "New REQ";
      break;
    case NTP_MODE_SERVER:
      typeStr = "Prev RSP";
      break;
    default:
      return;
  }
  TR3 "pkt 0x%016"PRIx64" Submitted %s for NTP CLIENT ROUND-TRIP: "
      "Org=%d.%.06d, Rcv=%d.%.06d, Xmit=%d.%.06d", pktInfop->pktID, typeStr,
      pktInfop->pktNtpInfop->ntpOrgTm.tv_sec,
      pktInfop->pktNtpInfop->ntpOrgTm.tv_usec,
      pktInfop->pktNtpInfop->ntpRcvTm.tv_sec,
      pktInfop->pktNtpInfop->ntpRcvTm.tv_usec,
      pktInfop->pktNtpInfop->ntpTmtTm.tv_sec,
      pktInfop->pktNtpInfop->ntpTmtTm.tv_usec END;
END_TR3
```

In the `.diag` configuration file, it is possible to globally deactivate the code to format and, maybe even to prepare (see above example between **START_TR3** and **END_TR3**), messages at a given level or higher: `Highest_level` directive (which takes precedence). This alleviates the runtime work.

Then, for each module, the desired level of verbosity and the list of destinations (log filename and/or `stdout` and/or `stderr`) can be specified.

Lines starting with a hash mark are comments

```
Highest_level  trace2

#
# FORMAT
# ======
#   program   module            level     log log ...
#   WHERE log is a full pathname or stdout or stderr
#
…
tmon_probe tmp_udptransaction warning  /var/log/trafMon/myProbe.log
#tmon_probe tmp_udptransaction trace2   /var/log/trafMon/myProbe.log stderr
#tmon_probe tmp_udptransaction trace3   /var/log/trafMon/myProbe.log
```

1.2.3 C Coding Convention

The C code is manually aligned for max. 80 characters per line (easy to print, easy to juxtapose multiple editor windows, e.g. gvim). Right alignment of split lines is made manually.

TABS are ALWAYS EXPANDED, aligned at 2 characters (minimal indentation for better readability).

Routine names start with upper case letter, variables with lowercase, typedefs end with _t, pointer variables end with p.

Routine names are normally prefixed as per their module name of at least program name.

Return type of a routine is written in the line above is definition, so that the routine name is left aligned. This permits to retrieve all definitions at once:

```
% grep '^[A-Z].*(' *.c
```

Each word in an identifier starts with an upper case letter. Except for types, underscore separators are avoided to keep identifiers relatively short.

As in K&R C (before ANSI-C), all local variables are declared at the start of the routine and, were relevant, are pre-initialised (especially pointers). This way, no difficulty for a reader to retrieve its declaration (and type).

Comments and readability are more important than a working code: anybody can later fix the problems, provided he can understand what the code is intended for (even if buggy). Comments must be maintained up-to-date with code changes!

Obfuscated construct must be avoided of fully explained.

Example:

```
/*
 * Extracts type and Request ID from SNMPv1/v2c or MessageID from v3 packet
 */
/*
 * SNMP Implies partial support of ASN.1 Basic Encoding Rules
 *
 * Routine that read one element form a BER sequence:
 * i.e. Its type (one byte)
 *      its length (one or more bytes)
 *      its value (depending on type-length)
 *
 * Returns 1 when successfully decoded next BER item
 *         0 upon parsing limitation (unexpectedly complex format)
 *        -1 upon format error
 */
int
TmpGetBerItem(const u_char **bufp, int *sizp, tmp_ber_t *item)
{
#ifndef NO_SANITY
  if(!bufp || !*bufp || !sizp || *sizp < 3 || !item) {
    WARN "Can't get any BER item from %p of size %u", bufp ? *bufp : 0,
                                                      sizp ? *sizp : 0 END;
    return(-1);
   }
#endif /* NO_SANITY */

  item->type = (*bufp)[0];
  (*bufp)++; (*sizp)--; /* Type removed */

  item->size = (uint16_t)(*bufp)[0]; /* Size [0..127] on 7 bits ? */
  (*bufp)++; (*sizp) --;             /* 1-byte size removed */
  if(item->size == 0x81) {           /* No, size [0..255] in next byte */
```

```
    item->size = (uint16_t) (*bufp)[0];
    (*bufp)++; (*sizp)--;           /* 1-more-byte size removed */
  } else if(item->size == 0x82 ) {   /* No, size [0..65535] in 2 next bytes */
    memcpy(&(item->size), *bufp, 2);
    item->size = ntohs(item->size); /* Put it in host byte order */
    (*bufp) += 2; (*sizp) -= 2;     /* 2-more-byte size removed */
  } else if(item->size & 0x80) {
    WARN "Can't handle BER items over 64K: %02x", item->size END;
    return(0);
  }

  if(item->size > *sizp) {
    TR2 "BER item needs more bytes than what buffer has got: %u > %u",
                                                  item->size, *sizp END;
    item->size = *sizp;
  }

  switch(item->type) {
   case BER_INT_TYPE:
     item->ber_intVal = 0;
     if(item->size > 4) {
       WARN "Can't handle INT over 32 bits: %u", item->size END;
       return(0);
     }
     memcpy(&(item->ber_intVal), *bufp, item->size);
     /* justify to low bytes: [abcdef--] -> [00abcdef]  */
     item->ber_intVal = ntohl(item->ber_intVal);    /* Host byte order */
     item->ber_intVal = item->ber_intVal >> (8 * (4-item->size));
     TR2 "BER decoded %u-byte INT: %u", item->size, item->ber_intVal END;
     break;
   case BER_STR_TYPE:
     item->ber_strVal = (char*)*bufp;
     TR2 "BER decoded STR of %u bytes", item->size END;
     break;
   case BER_SEQ_TYPE:
   case SNMP_GET_REQ:
   case SNMP_GET_NEXT:
   case SNMP_GET_RESP:
   case SNMP_SET_REQ:
   case SNMP_GET_BULK:
   case SNMP_INFORM:
   case SNMP_REPORT:
   case SNMP_V1_TRAP:
   case SNMP_V2_TRAP:
     item->ber_berVal = (void*)*bufp;
     TR2 "BER decoded BER of %u bytes", item->size END;
     break;
   default:
     item->ber_anyVal = (void*)*bufp;
     TR2 "BER decoded ANY(%02x) of %u bytes", item->type, item->size END;
  }
  (*bufp) += item->size; (*sizp) -= item->size;
  return(1);
}
/*
 * Minimal SNMP Dissector
 *
```

```
 *
 * UDP payload is one of:
 * SNMPv1/v2c
 * /--BER-SEQUENCE-------------------------------------------------\
 * |                                  /--PDU------------------\    |
 * | [--VERSION-NUM--] [--COMMUNITY--] | [--REQ-ID--]    ...    |   |
 * |                                  \------------------------/   |
 * \---------------------------------------------------------------/
 * SNMPv3
 * /--BER-SEQUENCE-------------------------------------------------\
 * |                  /--BER-SEQUENCE------------\                 |
 * | [--VERSION-NUM--] | [--REQ-ID--] ...         |    ...        |
 * |                  \-------------------------/                 |
 * \---------------------------------------------------------------/
 *
 * Returns 1 when transaction/request ID parsed from SNMP packet
 *         0 otherwise
 * Exit(1) upon sanity error
 */
int
TmpDissectSNMP(tmp_packet_info_t *pktInfop, const u_char *pkt)
{
   register int  res;       /* remaining payload size */
   const u_char *berp = 0;  /* Current position of BER decoder */
   int           size;      /* remaining payload size */
   tmp_ber_t     berItem;   /* Used to decode BER items */

#ifndef NO_SANITY
  if(!pktInfop || !pkt) {
    FATAL "INTERNAL: NULL argument pointer to routine" END;
  }
  if((pktInfop->pktHighestProtocol & isUDP) != isUDP) {
    ERR "pkt 0x%016"PRIx64" WRONG PACKET: Attempting to dissect SNMP"
                                  " on non UDP packet", pktInfop->pktID END;
    return(0);
  }
  if(pktInfop->pktPayloadLen < SNMP_MINHLEN) {
    ERR "pkt 0x%016"PRIx64" Too Short to dissect NTP Timestamps",
                                                        pktInfop->pktID END;
    return(0);
  }
#endif /* NO_SANITY */

  berp = &(pkt[pktInfop->pktPayloadOfst]);
  size = pktInfop->pktPayloadLen;

  /*
   * Surrounding SEQUENCE
   */
  res = TmpGetBerItem(&berp, &size, &berItem);
  if((res <= 0) || (berItem.type != BER_SEQ_TYPE)) {
    WARN "pkt 0x%016"PRIx64" WRONG PACKET: Not SNMP format",
                                                          pktInfop->pktID END;
    return(0);
  }

  /* Decode the content of the BER sequence */
```

```
  berp = berItem.ber_berVal;
  size = berItem.size;

  /*
   * SNMP Protocol Version: INTEGER
   * ======================
   */
  res = TmpGetBerItem(&berp, &size, &berItem);
  if((res <= 0) || (berItem.type != BER_INT_TYPE)) {
    WARN "pkt 0x%016"PRIx64" WRONG PACKET: Not SNMP format",
                                                        pktInfop->pktID END;
    return(0);
  }

  switch(berItem.ber_intVal) {
    case SNMPV1_VERSION:
    case SNMPV2C_VERSION:
      /*
       * Skip Community Name
       */
      res = TmpGetBerItem(&berp, &size, &berItem);
      if((res <= 0) || (berItem.type != BER_STR_TYPE)) {
        WARN "pkt 0x%016"PRIx64" WRONG PACKET: Not SNMP format",
                                                        pktInfop->pktID END;
        return(0);
      }
      /*
       * SNMP PDU Type
       */
      res = TmpGetBerItem(&berp, &size, &berItem);
      if((res <= 0) || (berItem.type < SNMPPDU_TYPE_MIN)
                    || (berItem.type > SNMPPDU_TYPE_MAX)
                    || (berItem.type == SNMP_V1_TRAP)
                    || (berItem.type == SNMP_V2_TRAP)) {
        return(0);
      }
      /* only for request/response */
      pktInfop->pktSnmpInfop->snmpType = berItem.type;
      /*
       * SNMP Request ID
       */
      berp = berItem.ber_berVal;
      size = berItem.size;
      res  = TmpGetBerItem(&berp, &size, &berItem);
      if((res <= 0) || (berItem.type != BER_INT_TYPE)) {
        return(0);
      }
      pktInfop->pktSnmpInfop->snmpReqID = berItem.ber_intVal;
      return(1);
    case SNMPV3_VERSION:
      /*
       * SEQUENCE
       */
      res = TmpGetBerItem(&berp, &size, &berItem);
      if((res <= 0) || (berItem.type != BER_SEQ_TYPE)) {
        return(0);
      }
```

```
      /*
       * SNMPv3 MessageID
       */
      berp = berItem.ber_berVal;
      size = berItem.size;
      res  = TmpGetBerItem(&berp, &size, &berItem);
      if((res <= 0) || (berItem.type != BER_INT_TYPE)) {
        return(0);
      }
      pktInfop->pktSnmpInfop->snmpReqID = berItem.ber_intVal;
      return(1);
      /* XXX should continue to decode for trying to detect no privacy */
      /* (i.e. not encrypted) and decode the SNMP PDU Type             */
    default:
      /* not right version */
      return(0);
  }

  return(1);
}
```

1.2.4 trafGen: TCP/UDP Packet Generator

The trafgen program re-uses the `tmon_core diag`, `timer` and `config` modules and make use of a similar XML/DTD approach for the configuration file.

```
$ ls tgen
Makefile    tgen_TCP.h  tgen_config.c  trafgen       trafgen.h
tgen.dtd    tgen_UDP.c  tgen_config.h  trafgen.c
tgen_TCP.c  tgen_UDP.h  tgen_ipts.xml  trafgen.diag
```

The DTD gives explanations for use:

```
<!--
      Copyright (c) 2020 AETHIS s.a./n.v., Belgium. All rights reserved.
      www.trafmon.org

      Licensed under the Apache License, Version 2.0 (the "License");
      you may not use this file except in compliance with the License.
      You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

      Unless required by applicable law or agreed to in writing, software
      distributed under the License is distributed on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
      See the License for the specific language governing permissions and
      limitations under the License.
-->
<!-- The Traffic Generator (trafGen) DTD  ### Current version $Id:
aedf10a5fc18d828c81ab6aa2cd6b838c92130f4 $-->
<!ELEMENT TrafGen (Peers, Scenario) >
<!ATTLIST TrafGen name    NMTOKEN #REQUIRED
                  serial  NMTOKEN  #REQUIRED
                  startAt CDATA    #REQUIRED
>
```

```
   <!-- the name identifies the generator instance and its .xml config -->

<!ELEMENT Peers EMPTY >
<!ATTLIST Peers srcIP   NMTOKEN #IMPLIED >
<!ATTLIST Peers srcPort NMTOKEN #IMPLIED >
<!ATTLIST Peers dstIP   NMTOKEN #REQUIRED >
<!ATTLIST Peers dstPort NMTOKEN #REQUIRED >
<!ATTLIST Peers proto   (udp | tcp) "udp" >
   <!-- srcIP and srcport are optional                    -->
   <!--        lets kernel choose if absent              -->
   <!-- dstIP may be a MULTICAST address                  -->
   <!--        implies setsockopt(IP_MULTICAST_LOOP, 0)  -->
   <!--        uses setsockopt(IP_MULTICAST_TTL) instead -->
   <!--         of setsockopt(IP_TTL) when applicable    -->

<!ELEMENT Scenario (Message+) >
<!ATTLIST Scenario period     NMTOKEN #REQUIRED >
<!ATTLIST Scenario firstdelay NMTOKEN "0" >
   <!--  period:      length, in msec, of a cycle covering the     -->
   <!--               generation of msgcount message instances.    -->
   <!--               More precisely, duration between the start   -->
   <!--               times of two successive such cycles.         -->
   <!--        N.B. The first period starts on a boundary aligned  -->
   <!--             on an absolute second (as know by the local    -->
   <!--             computer)                                      -->

<!ELEMENT Message EMPTY >
<!ATTLIST Message flowname   NMTOKEN #REQUIRED >
<!ATTLIST Message payloadlen NMTOKEN #REQUIRED >
<!ATTLIST Message mtu        NMTOKEN #IMPLIED >
<!ATTLIST Message interfrag  NMTOKEN "0" >
<!ATTLIST Message intermsg   NMTOKEN "0" >
<!ATTLIST Message msgcount   NMTOKEN "1" >
<!ATTLIST Message ToSprec    NMTOKEN "0" >
<!ATTLIST Message ToSdtr     NMTOKEN "0" >
<!ATTLIST Message iptscount  NMTOKEN "0" >
<!ATTLIST Message iptsaddr   (false|true) "false" >
<!ATTLIST Message payload    CDATA #IMPLIED >
   <!-- A Message is a unit of payload data that can be fragmented -->
   <!-- When a MTU is explicitly specified, the fragmentation is   -->
   <!-- made explicit by the generator. But the total volume       -->
   <!-- resulting from the sending of one message instance is the  -->
   <!-- same as if it would have been fragmented by IP layer: the  -->
   <!-- UDP or TCP headers of second and remaining fragments are   -->
   <!-- substracted from the message payloadlen                    -->
   <!--  firstdelay:  msec to wait after the boundary aligned on   -->
   <!--               the period before sending the first    -->
   <!--               message instance.                            -->
   <!--  payloadlen:  #bytes excluding first TCP/UDP header and    -->
   <!--               excluding all fragment IP headers (see above)-->
   <!--  mtu:         if present and > sizeof(iphdr+tcp/udp hdr)   -->
   <!--               is used to compute explicit pseudo           -->
   <!--               fragmentation of every message instance      -->
   <!--  interfrag:   msec to wait between successive fragments of -->
   <!--               a same message instance.                     -->
   <!--  intermsg:    msec to wait between last fragment of a msg  -->
   <!--               instance and first frament of next msg within-->
```

```
<!--                the same period.                                  -->
<!--  msgcount:     # of successive message instances to send         -->
<!--                within a same period.                             -->

<!--    CONSTRAINT:                                                   -->
<!--     firstdelay + ((#frags -1) * interfrag) * msgcount            -->
<!--                 + intermsg * (msgcount -1) < period              -->

<!--  ToSprec:      IP ToS precedence value (three high order         -->
<!--                bits of the IP ToS byte)                          -->
<!--        0<= ToSprec <8                                            -->
<!--  ToSdtr:       Value of the three D/T/R bits if the IP ToS       -->
<!--                byte.                                             -->
<!--        0<= ToSdtr <8                                             -->
<!--           N.B. the binary concatenation of (ToSprec|ToSdtr)-->
<!--                form the value of the DifServ DCSP                -->
<!--                The two low order bits of the IP ToS byte         -->
<!--                are always set to zero.                           -->
<!--           N.B. if ToSprec==0 and ToSdtr==0, the generator        -->
<!--                does not specify the IP ToS to the IP stack       -->
<!--  iptscount:  When iptscount > 0, creates an IP TIMESTAMP         -->
<!--              for containing up to iptscount (limited by          -->
<!--              maximum IP header length)                           -->
<!--  iptsaddr:   When iptscount > 0, ask the gateways to place       -->
<!--              their IP address together with their IP             -->
<!--              timestamp.                                          -->
<!--  payload:    Optional content to be placed at start of           -->
<!--              message payload (less than 10000 resulting          -->
<!--              bytes).                                             -->
<!--      format: string of hexadecimal bytes (case insensitive)-->
<!--              with following keywords:                            -->
<!--              - sequence of 'T' (or 't') characters means         -->
<!--                to put current timestamp in second coded          -->
<!--                over as many bytes as there are 'T' (or 't')-->
<!--                characters.                                       -->
<!--                MAXIMUM 4 consecutive 'T'.                        -->
<!--              - sequence of 'N' (or 'n') characters means to-->
<!--                put current message number coded over as          -->
<!--                many bytes as there are 'N' (or 'n') chars        -->
<!--                MAXIMUM 4 consecutive 'N'.                        -->
<!--              - sequence of 'L' (or 'l') characters means to-->
<!--                put current message fragment total payload        -->
<!--                length.                                           -->
```

This sample XML config generates UDP datagram units sent from 141.253.123.200 to UDP port 12345 at 192.168.102.30.

Every 500 ms, it generates:

- first, one short datagram with 128 bytes UDP payload (unfragmented), with room for 4 IP timestamps (unlike Solaris, Linux fills the first at source host), then waits 200 ms;
- then a big datagram, with 2880 bytes UDP payload, which results in
    - one fragment, with room for 3 IP timestamps, then waits 100 ms,
    - one fragment, with room for 3 IP timestamps

```
<!DOCTYPE TrafGen SYSTEM "tgen.dtd">

<TrafGen name="tgen_ipts" serial="1" startAt="2014-03-27 00:00:00">
  <Peers proto="udp" srcIP="141.253.123.200" dstIP="192.168.102.30"
                                                        dstPort="12345"/>
  <Scenario period="500">
    <Message flowname="ipts" payloadlen="128" msgcount="1" intermsg="200"
      iptscount="4"/>
    <Message flowname="ipts" payloadlen="2880" msgcount="1" interfrag="100"
      iptscount="3"/>
  </Scenario>
</TrafGen>
```

1.3 TRAFMON MEASUREMENT MECHANISM

1.3.1 Probing

All measurements conducted by trafMon come from the inspection of IP (IPv4) traffic packets captured by probe interfaces (C source file *tmp_interface.c*).

The packet capture relies on the standardised `libpcap` 1.4 module which, in its Linux implementation, stores the packet in a possibly quite large kernel-resident ring buffer. In trafMon, each probing interface can be configured with a storage capacity large enough to cope with foreseeable peaks in high rate traffic flows. Such user configuration tuning must be derived from a trade between the overall portion of computer RAM dedicated to this usage and the traffic bursts anticipated at all the probing interface of a same probe computer. Only the packet selected by the BSD capture filter assigned to each interface will actually enter its kernel-resident capture buffer.

Such buffer capacity is set by the user in terms of number of number of (1600 bytes) Ethernet frames.

Packet capacity `bufPacketCount`	Maximum Frame Size `snapLen`	Consumed RAM Bytes
70 000	1600	112 000 000
1 000 000	1600	1 600 000 000

Table 1: Per Probe Capture Interface Kernel Buffer Memory Consumption

Care must however been taken in tuning such buffering for each interface. RAM is also required for the probe processing (internal data structures, but also shared memory buffer between father and child processes of the trafMon probe program) and file I/O (although limited on probe dedicated computing platforms), as well as the potential output queue of un-acknowledged trafMon PDU's at destination of (temporarily unreachable) collectors.

By hiding this Linux specific kernel interface behind a general portable API, the `libpcap` does unfortunately not reflect the actual amount of RAM that has been reserved. This

could well be only half of the requested size. An indirect (uneasy) feedback can be given when launching the `tmon_probe` with `strace`:

```
# strace -o strace.log ./tmon_probe -l probe
…
^C
```

In the strace.log, you can see the way the `socket(PF_PACKET, SOCK_RAW)` is initialized:

```
socket(PF_PACKET, SOCK_RAW, 768)        = 3
ioctl(3, SIOCGIFINDEX, {ifr_name="lo", ifr_index=1}) = 0
ioctl(3, SIOCGIFHWADDR, {ifr_name="Auto_p1p1", ifr_hwaddr=00:10:18:f7:b2:88}) =
0
ioctl(3, SIOCGIFINDEX, {ifr_name="Auto_p1p1", ifr_index=4}) = 0
bind(3, {sa_family=AF_PACKET, proto=0x03, if4, pkttype=PACKET_HOST, addr(0)={0,
}, 20) = 0
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
setsockopt(3, SOL_PACKET, PACKET_ADD_MEMBERSHIP,
"\4\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0", 16) = 0
setsockopt(3, SOL_PACKET, PACKET_AUXDATA, [1], 4) = 0
getsockopt(3, SOL_SOCKET, 0x30 /* SO_??? */, [52], [4]) = 0
getsockopt(3, SOL_PACKET, PACKET_HDRLEN, [28], [4]) = 0
setsockopt(3, SOL_PACKET, PACKET_VERSION, [1], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RESERVE, [4], 4) = 0
ioctl(3, SIOCETHTOOL, 0x7ffc17353e10)   = 0
getsockopt(3, SOL_SOCKET, SO_TYPE, [3], [4]) = 0
getsockopt(3, SOL_PACKET, PACKET_RESERVE, [4], [4]) = 0
setsockopt(3, SOL_PACKET, PACKET_RX_RING, {block_size=4096, block_nr=532610,
frame_size=288, frame_nr=7456540}, 16) = -1 ENOMEM (Cannot allocate memory)
setsockopt(3, SOL_PACKET, PACKET_RX_RING, {block_size=4096, block_nr=505979,
frame_size=288, frame_nr=7083706}, 16) = 0
```

Look at the Linux header file `/usr/include/linux/if_packet.h`: the argument to **PACKET_RX_RING** socket option is a `struct tpacket_req`, as below:

```
/*
   Frame structure:

   - Start. Frame must be aligned to TPACKET_ALIGNMENT=16
   - struct tpacket_hdr
   - pad to TPACKET_ALIGNMENT=16
   - struct sockaddr_ll
   - Gap, chosen so that packet data (Start+tp_net) alignes to
TPACKET_ALIGNMENT=16
   - Start+tp_mac: [ Optional MAC header ]
   - Start+tp_net: Packet data, aligned to TPACKET_ALIGNMENT=16.
   - Pad to align to TPACKET_ALIGNMENT=16
 */

struct tpacket_req
{
        unsigned int    tp_block_size;  /* Minimal size of contiguous block */
        unsigned int    tp_block_nr;    /* Number of blocks */
        unsigned int    tp_frame_size;  /* Size of frame */
        unsigned int    tp_frame_nr;    /* Total number of frames */
};
```

We see that a **snaplen**="210" is translated by `libpcap` into `tp_frame_size`=288. And we asked for a **bufPacketCount**="10 000 000". Due to the page size (`tp_block_size`=4096), a block can hold a maximum of 14 frames. We were asking room for more than 210*10 000 000 bytes (via `libpcap pcap_set_buffer_size()`). And the library translates that by asking room for 7 456 540 frames (max 14 frames per block, and an amount of 532610 blocks). But available memory doesn't suffice (**ENOMEM**), hence libpcap reduces its request to 7 083 706 frames (5% less), and this passes. But the probe (with single interface) reserves more than 2000 MiB upon initialisation (`free -m` command before and after the probe launch).

Being hidden inside the libpcap API, the per-interface packet buffer is not directly accessible to the trafMon probe program, running in user space. And, for the sake of reaching the *stringent performance requirements* imposed by the high and sustained data rate of the EO ground network, the actual content of the vast majority of capture packets will **not undergo any further copy** inside the probe program user space.

Packets are therefore analysed one after the other.

Each captured packet is systematically dissected at all possible layers of its protocol stack (IPv4, ICMP, UDP, TCP, DNS, NTP, SNMP, FTP (HTTP hasn't been implemented): C source file *tmp_pkt_dissect.c*. Note that Flow Class filter mapping (single pass of combined sieve) is necessary to determine which dissection is requested above UDP or TCP transport layer (if any).

During protocol dissection, global probe-wide counters are updated.

It is not required to assign an IP address to the probing interface. The only mandatory operation is to configure such interfaces as UP. This way, packet capture will occur in stealth mode: the probe will not be visible on the LAN segments it is connected to, and cannot be a target for attack at IP layer. Furthermore, the packet inspection and analysis does not rely on trust placed on information field (e.g. announced data lengths), to avoid that the probe be impacted when capturing a malevolent forged packet.

Although the probe requires packet capture privilege to run, it is not mandatory to run it as root super-user:

- produce a /opt/tmon/bin/**tmon_probe** executable binary;
- create a specific Linux *group*, e.g. **pcap** and Linux *runtime account for the probe*, e.g. **probe** having pcap as primary group
- Assign pcap as executing group to the **tmon_probe:**

```
# chgrp pcap /opt/tmon/bin/tmon_probe
# chmod 0750 /opt/tmon/bin/tmon_probe
```

- Assert the CAP_NET_RAW, CAP_NET_ADMIN capabilities for the **tmon_probe** binary:

```
# setcap cap_net_raw,cap_net_admin=eip /opt/tmon/bin/tmon_probe
# getcap /opt/tmon/bin/tmon_probe
```

```
/opt/tmon/bin/tmon_probe = ap_net_admin,cap_net_raw+eip
```

- The use of `sched_setaffinity`(2) could also requires CAP_SYS_NICE capability. But this isn't necessary by the process modifying its own affinity.

1.3.2 Filtering for Flow Classes Matching

The results of a packet dissection (up to transport level) permit to apply all combined predicates of all defined filters from all applicable configured flow classes. This filtering method has been carefully implemented in the trafMon probe, in the most efficient way: one single sieve traversal for rejecting all non-applicable Flow Classes (C source file *tmp_flowfilter.c*). Indeed, all flow calls applicable to a given probe interface provides a filter combining predicates on its protocol fields (Px). In its most general form, such filter combines up to three levels 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)

Flow class filters are not necessarily mutually exclusive: a packet may well match several flow classes and, therefore, participate to different measurements.

A same protocol field can therefore be ruled by multiple predicates coming from different flow class filter expressions.

The design driver for efficient flow class filtering relies therefore on the following optimisation strategies:

- All predicates relative to a same protocol field are tested altogether, one after the other (with the field value kept in a CPU register).
- As soon as a predicate test permits to conclude on the success or rejection of an entire filter expression of a flow class, all its remaining predicates are disabled for the packet under inspection.
- Systematic resort to switch/case C code constructs permits to avoid the overhead of long series of if/then/else if ... exhaustive testing sequences.

Each time a predicate is tested, its result recursively updates (from top to bottom) the 1, 2 or 3-layers expression tree of connectives for the corresponding class.

Once the set of applicable flow classes has been determined for a first or single fragment packet, the probe (father process) can decide whether the packet

- will be ignored or
- will require only its dissected field information for further analysis and measurement,
- will require to keep also its actual payload data for specific analysis (e.g. checksum of fragmented datagram.

1.3.3 Retaining FTP Data Connection Packets and IPv4 Second and Subsequent fragments

Two specific cases also occur here, for not yet fully qualified packets:

- A TCP packet between a pair of host IP addresses could be member of the Data connection of an FTP session. As it is always the case in practice, FTP data transfers are assumed to occur between the same peers as the corresponding FTP control connection (this is not imposed by the protocol itself but by the implementations tradition). Therefore when first encountering the establishment of such FTP control connection, the peer addresses are dynamically registered (in *TmpPktKeepFtpDataConnRoot* binary search tree).

  Any remaining TCP packet (or only its connection start/end SYN / FIN / RST when only "*start-stop*" heuristic measurement is requested for the file transfer) that match this address pair has its dissected information remembered for further stateful analysis.

  Note that analysing only the **ftpdata**="start-stop" of TCP data connection permits to skip the analysis (and the per-packet copy of dissected information) of nearly all packets in a network dominated by FTP transfers.

  While encountering the end packet of the control session (in practice, always closed only after the end of a data transfer), the address pair is de-registered.

- A second or subsequent IP fragment does not contain the upper layer protocol headers. Hence its dissection and flow class membership is incomplete. However, such fragments need further analysis only when measurements are required:
  - for "fullReassembly" checksum verification or
  - when requested measurement is for "allFragments" or for reassembled "datagram"

  But fragments could be captured and inspected out of order. So the implemented strategy is to retain all second or subsequent fragments, except those whose first fragment has already been analysed, concluding that remaining fragments are not needed and may be dropped.

  So after analysis of a first fragment of an incomplete datagram, if none of its matched flow classes imposes any of the above two rules then its source and destination addresses and IP identifier are registered (*TmpReassRegSkippedDgram()*), permitting to reject remaining corresponding fragments (*TmpReassIsSkippedDgram*()). Those registry entries are regularly cleaned-out by timeout.

  Note that today, IP fragments are quite seldom. In a real target network context, we have observed that only the UDP SNMP packets (response PDU) are sometimes too long to fit in a 1500 bytes IP packet. Note the TCP is never fragmented, as the WAN link can now afford the same MTU size as for Ethernet LAN segments.

Finally, for those first or single fragment packets which require individual reporting, the packet signature hash is computed, permitting to release the packet payload at this stage.

Note that, by dissecting the FTP Control packets, in the father probe process, the **sensitive information (i.e. the user FTP password) is voluntarily not remembered**. This way such items do not leave the captured packet residing only in the OS kernel of the probe computer.

1.3.4 Protocol Stateful Analysis and Probe Measurements

All possible processing based on single packet content is now complete.

In order to take party of the modern processor architecture (multi-core/multi-CPU parallelism), and for conducting further stateful protocol analysis and/or metrics measurements updates on a separate processor core, the rest of the processing of the retained packets is executed in a child process (C source file *tmp_child.c*).

The father process removes itself from all CPU core, but one. This CPU ID is then removed from the scheduling affinity list of the child process (*sched_getaffinity()* and *sched_setaffinity()* Linux system calls). The resulting effect is that the father assigns itself to one CPU core only, while the child can be scheduled only on all other available CPU cores.

The child process is launched only after parsing of the configured Flow Class XML definitions. The resulting internal data structures, created on the father process virtual memory, are therefore inherited as such by the child (at same virtual memory addresses as those in father's space). So the global table *tmpFlowClassTab[]* and its anchored substructures can be referred to by the same index from father and from child processes.

A double circular buffer (common core C source file *tmon_circbuf.c*) for variable size data chunks is instantiated by the father process inside a shared memory area (common core C source file *tmon_sharedmem.c*). The father is the sole writer in the circular buffer, while the child is the sole reader. Hence a circular buffer is the most efficient way to implement queuing from father to child, without need of mutual exclusion via semaphores. But the father warns its child about new availability of queued data by sending it a SIGUSR2 signal, waking it up when blocked in a `select`() system call. Inversely, when the buffer is full, the father blocks on a `usleep`() (`nanosleep`() system call) and can also be woken up by a SIGUSR2 signal from the child.

A queued packet always requires its dissected information (*tmp_packet_info_t*) and upper layer "transport" sub-structure (*tmp_tcp_info_t* or *tmp_dns_info_t* or *tmp_ntp_info_t* or *tmp_snmp_info_t* or *tmp_icmp_info_t*) and "application" sub-structure (*tmp_ftp_info_t* or *tmp_http_info_t*). These occupy up to three successive slots in the circular buffer.

When the IP payload is also needed, for further processing of the packet, a last circular buffer slot is filled with this captured upper part raw data chunk.

After having fetched all queued data slots information on a next captured packet, the child's processing is driven by the *TmpAnalyse()* routine (C source file *tmp_analyse.c*).

Where relevant, it starts by re-assembling the datagram (with or without payload data content), parking fragments in reassembly queue as long as the latest fragment isn't gathered (or reassembly timeout of **TMP_MAX_REASS_TIME=7 seconds** in *tmp_reassembly.h*).

All pending reassembly queues for one probing interface are stored in a dictionary *tmpReassTrees[interface index]* (common C source files *tmon_dict.c* and *tmon_btree.c*). Inside a given queue, fragment structures are linked in a list ordered by the sequence order of their payload data in the complete datagram.

Reassembly queues are scanned every TMP_MAX_REASS_TIME seconds for detecting timed-out queues and releasing their member fragments (via *TmpReassFailed()*).

A reassembled datagram consists in an ordered linked list of *tmp_packet_info_t* structures whose handling pointer (*pktInfop*) indicates the first fragment.

In a first analysis pass, the list of flow classes matched by the **datagram** (by the first fragment thereof) is scanned.

- If requested, the checksum is verified on reassembled datagram.
- Current flow class is attached to every fragment of the datagram (as if they matched themselves the filter).
- Depending on requirement of upper layer protocol analysis imposed by the current class:
    - TCP connection stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulTCP()*);
    - FTP control stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulFTPCtl()*);
    - HTTP stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulHTTP ()* hasn't been implemented yet);
    - The further <`FlowClass`><`Condition`> statements are verified on the datagram, to validate or reject the current class membership;
- Finally, the required processing synthetic flags for the datagram are extended with those applicable, for the corresponding probe interface, from the current class.

Maybe a TCP datagram not yet leading to TCP connection state analysis is member of an FTP data connection. It is then mapped to its connection by performing the missing TCP stateful analysis.

When some applicable class requests reporting every 20s and another every 30s, the period is adapted to 10s. Or the same is done for period in minutes.

Then, according to the applicable <GranularFlow> specification(s), the **datagram**

- either leads to the creation of newly discovered probe flow instance(s), which is given a probe own unique flow ID,
- or is mapped to already discovered flow instance(s)

Note that it is not precluded that a same packet participates to the measurements made for more than one flow instance.

Knowing the flow instance(s) the datagram belongs to, the relevant flow instance statistics counters can then be updated at flow instance level.

Then, **each subsequent fragment** undergoes the similar processing, where required by the concerned flow classes:

- The <Condition> criteria for every applicable flow class are verified or the class membership is denied for the fragment;
- Fragment-based flow instance membership is determined;
- If requested per packet, corresponding flow instance statistics counters are updated.

The second analysis pass is dedicated to specific measurements:

- <OneWayDelay>: leads to the retrieval of timestamps (capture, IP Timestamp option, NTP timestamps) and size individually reported, per datagram or per fragment, to the collector(s).
- <InterPacket> delay: when measured and pre-aggregated by the probe, the delays between successive datagrams (and fragments inside a datagram) are updated (not implemented yet).
- <RoundTripDelay>: depending on applicable flow classes specifications, and on the detected type of datagram, following round-trip delay measurements are computed:
    - TCP SYN vs. SYN-ACK (between probe and responder) and/or

      Only once at start of a new connection; this is a true transmission delay as the SYN/ACK is replied immediately by the listener.
    - TCP RTTM peer packets mapping: double probe round-trip delays, resp. with responder and with initiator sides of the TCP connection.

      Although optional, RTTM is apparently systematically used by modern operating systems.

      This provides an upper bound of the combined delay to transmit, to queue, to ingest and to acknowledge (possible after a delayed ack wait time) the data

packets. For connection transmitting data only in one direction, only the delay between the probe and data receiver can be computed.

At least with Linux systems, it has been observed that some instance of measurements produce an artificially long delay: the sender forwards three packets in a burst; all three with the same RTTM value. But these are individually acknowledged; the first ACK comes back quickly, the second one after a longer queuing time and the third after a significantly longer time.

- TCP Data/Ack delay is more efficiently replaced by the RTTM analysis. Finally, it hasn't been implemented.

  The RTTM standard mechanism has actually been proposed for producing more accurate measurements, especially in the case of retransmissions. So this way of measurement can only be worse than that based on RTTM.

- NTP-based delays from probe: with responder (NTP server) and possibly with initiator (client: this is the current NTP client polling period vis-à-vis a server).

  With the responder, this measurement typically reflects the pure 2-way transmission delay (server responds immediately).

  Delay with the initiator (NTP client) depends on the fact that the client embeds the server time of the previous NTP response (from this same server) as origin time in its next request. For servers considered stable (or too bad) by a client, the client polling period (delay between successive request/response transactions) is quite long (1024s =~17 minutes). A short value (few dozens of a second), observed during a significantly long period of time, indicates problem in correct time synchronisation by the client (e.g. unstable time server clock or jitter in the network travel).

- DNS round-trip delay between probe and responder (DNS server).

  Depending on what is requested and whether the response is cached or not, the DNS transaction can be quick (pure 2-way transmission time) or long (cascaded resolution with potentially distant servers). So only those values near to the minimum (lower delay histogram slice) are representative of the 2-way round-trip transmission time.

- SNMP round-trip delay between probe and responder (SNMP agent).

  What is characteristic of the SNMP round-trip time is that it measures the delay of remote nodes vis-à-vis the control centre location (not necessarily representative of client-to-server network paths).

- ICMP Echo (ping) round-trip delay between probe and responder.

1.3.5 Probe PDU Protocol with Collector

There are several different types of measurement data that are reported by the probe program to the set of collectors. Some types consist in individual measurement records, other record types consist in time-regular samples of values of variables accumulating the individual measurements (updated at each occurrence of a relevant monitored packet). Each such type corresponds to its own type of probe PDU (in common source file *tmon_PDU.h*).

The general mechanism for transmission of observations by the probe is the following:

- Records are formatted by an appropriate publishing routine (in C source file *tmp_publish.c*).
- For every target collector (and/or the optional local file logging), the formatted record is appended to the PDU under construction at destination to the target collector. Each PDU type has its own table of such pending PDU per collector (*tmpPubFlowPduCollTab[], tmpPubTSPduCollTab[], tmpPub2TSPduCollTab[], tmpPubFlCtrsPduCollTab[], tmpPubHistDescrPduCollTab[], tmpPubHistDataPduCollTab[], tmpPubTCPConnPduCollTab[], tmpPubFTPXferPduCollTab[]*), dynamically allocated at start-up. There is also a standalone pending PDU per type for local saving into a buffer file (currently not exploited) with same names where the '*CollTab[]*' is replaced by '*LocalSavep'*.
- When the PDU is full, or its contained data are older than the configured "*maxPDUBuildTime*" seconds, the PDU is written to the local buffer file or passed to the sending module (C source file *tmp_transmission.c*).
- Sometimes the PDU is flushed-out earlier (flow instance descriptions must be passed to collectors before data potentially referring to them; same for histogram slice definitions before the aggregated metric data).

The PDU sending (*TmonXmitPduSend()*) obeys sophisticated control mechanism.

When no PDU have been sent since a while, the PDU is immediately sent:

- a record is kept in the acknowledgement pending queue (*tmpXmitConfigs[collector_index]. pduSentTreep*)
- the PDU is sent over the UDP socket (*TmpXmitSendPduNow()*)
- a retry timer is armed for the PDU (*TmpXmitSendPduNow()*)

Otherwise, a next free time slot is obtained for the PDU via *TmpXmitCollTimeSlot(TMP_XMITSLOT_GET_TIME_SLOT)*. Indeed, for each target collector, a minimum of "*minTimeGap*" milliseconds delay must be respected between consecutive sending or re-sending of probe PDU's. Hence the need to assigned scheduling time slots to output pending PDU's. And a timer is armed to invoke *TmpXmitSendPduNow()* at corresponding time slot.

Each time a PDU is sent, its retry count is decreased, and its retry timer timeout delay is re-computed with based on "**TOMult**" and/or "**TOIncr**".

When the first pending PDU exhausts its "**retries**" count (has been sent **retries**+1 times and after last computed timeout value), the connectivity with the collector is declared *inLongRetryMode*:

- This particular PDU will be continuously retried at the frequency given by the initial timeout period for this collector;
- Next pending PDUs to timeout their last retry will be handled according to their type:
    - Important PDU with flow instance descriptions (*TMON_PDU_TYPE_FLDESCR*), with histogram/aggregate descriptions (*TMON_PDU_TYPE_HISTDSC*) or with FTP file transfer records (*TMON_PDU_TYPE_FTPXFER*) will be continuously retried at the longest timeout period.
    - For the other PDU types, a break border time window is computed as [start, start+ "**breakBorderTime**"] in seconds, where start is the time of oldest info in the PDU that triggered the long retry mode. This determines a period "*just before the lack of connectivity*" (hence the name "*at the border of a break*"). PDU's whose oldest record fit inside this break border time window are also continuously retried at their longest timeout period: indeed they may contain observations illustrating the degradation of the traffic conditions.
    - Those other PDU types with data younger than the break border are finally dropped: those which continue to accumulate information (flow counters *TMON_PDU_TYPE_FLCNTRS*), which contain detailed individual observations (per packet/datagram *TMON_PDU_TYPE_PKTOBS* or per individual delay measurement *TMON_PDU_TYPE_DELOBS* or individual TCP connection details *TMON_PDU_TYPE_TCPCONN*) or which contain less crucial measurements (current time interval of probe aggregated metrics *TMON_PDU_TYPE_HISTDTA*).

NOTES:

1) The "**TOMult**" parameter must be used with care, as successive timeout would increase quite quickly to unreasonably long duration.
2) When the connectivity from a probe to a collector is re-established, it can take a while for the remembered observations data arrive at the collector: longest delay = timeout * **TOMult**^**retries** + **TOincr** * (**TOMult**^**retries**-1 + **TOMult**^**retries**+ ... +**TOMult**^2+**TOMult**^1+1).

When no PDU has been sent by the probe since "**heartBeatDelay**" seconds (*TmonXmitHeartBeat()*), a fake empty heart beat is sent (*TmonXmitHeartBeatSend()*), without registration as no acknowledged will be expected (C source file *tmp_transmission.c*).

The collector program receives the probe PDUs (*TmcXmitValidateAndAckPDU()* in C source file *tmc_transmission.c*), validates its shape, type and CRC, updates the corresponding probe known status (C source file tmc_probes.c) and, when it isn't a fake

heart beat empty PDU, sends back an acknowledge with full redundancy (2 times the PDU ID and first byte of CRC).

Then, if the PDU hasn't been received yet (tracked in probeInfop->rcvdPduDict), it is pushed at end of the *tmcInputQp* input queue (instance of the common pointer based circular buffer implemented by core *tmon_circ_pbuf.c*).

1.3.6 Collector Further Processing and Output

In addition to its role in centralising the probe observations, the collector should have three functions:

- Merging information from several probes:
  - assigning a unique identifier to probe dynamically discovered flow instances;

    Experience has revealed that this is only necessary for partial one-way observations produced for a same flow by multiple probes. All the rest of flow measurements must be tagged with their probe interface reference. So they could keep their probe-assigned unique flowID, complemented by the (fixed) probeID. This way, receiving flow observations before the corresponding flow description wouldn't be a problem. And the collector could be restarted, independently of the probes, with less consequences.
  - matching partial per-packet (per datagram) observations records (with size and list of timestamps) in order to produce one complete record for a same data unit observed at different locations;
- Pre-computing certain metrics in order to detect abnormalities in the traffic performance in near real-time (this hasn't been implemented yet);
- Further aggregation of instant measurements to reduce the amount of information pushed to the database (not implemented):
  - this feature won't be needed in case the volume of measurements produced by the probes is reasonable;
  - there is no universal criteria, besides those already implemented upstream by the probes, that can determine which metrics must be computed (once for all) by the collector at the exclusion of all others that could be derived for the then destroyed raw observations;
  - if needed, this function could be best implemented outside the online collector C program, via scripts applied to the post processing of the collector output raw data files; this would offer the necessary flexibility in addressing the specific needs of different use case environments;

Therefore, the current collector primarily acts as a formatter of the observations provided by the probe into corresponding types of output files.

And for the specific case of individual packet/datagram one-way observations, its stores the partial observations in a dictionary with sophisticated ability to retrieve stored items "near in time" to a given search key.

The several probe PDU types are decoded by dedicated routines in the C source file *tmc_pdu_decoder.c*.

- Flow instance descriptions (*TMON_PDU_TYPE_FLDESCR*):

  Retrieves from the registry (*probeInfop->prFlowIdMap*) based on the corresponding probe Flow ID.

  If found and same description, simply maps to collector-assigned globally unique ID.

  If not found or new definition, remove potential ancient record; generate a new collector-assigned globally unique ID: the UNIX timestamp at the millisecond; keep track of the new flow description (*TmcFlowInstanceNew()*) and writes the flow instance description in corresponding output file (*TmcFlowInstanceOutput()*) in C source file *tmc_flowinstance.c*).

- Protocol counters records per flow instance (*TMON_PDU_TYPE_FLCNTRS*):

  When probe flow isn't yet known, the record is IGNORED.

  Decodes the variable length fields of each record and invokes *TmcStatsFlowInstanceUpdate*(): Outputs those sub-records that have non zero counters (C source file t*mc_statistics.c*):

  - IPv4 counters: *TmcStatsFlowIPUpdate()*
  - IPv4 packet sizes distribution: *TmcStatsFlowIPDistrbUpd()*
  - ICMP counters: *TmcStatsFlowICMPUpdate()*
  - UDP Counters: *TmcStatsFlowUDPUpdate()*
  - TCP counters: *TmcStatsFlowTCPUpdate()*
  - FTP counters: *TmcStatsFlowFTPUpdate()*
  - [HTTP counters: *TmcStatsFlowHTTPUpdate()* not implemented]

- Per packet/datagram individual observations (potentially partial) (*TMON_PDU_TYPE_PKTOBS*):

  The TmcPduDecodePktObs() routine implements a specific processing that is detailed below (see section 4.4.6 below).

  When the collector reconstitute a complete packet/datagram observation record, with size and all expected hop timestamps, it outputs it via *TmcPduDecodePktObs()*

- Per delay individual records (*TMON_PDU_TYPE_DELOBS*):

  Those individual delay measurements (round-trip, currently) are optionally produced, not aggregated inside the probe, and delivered to the collector. But these aren't yet further processed because they would be used for collector–based event detection and, possibly, collector aggregation, which are functions targeted but not implemented.

Therefore, those PDU are currently simply dissected by a debugging trace routine, without output yet. Nevertheless, round-trip delays are provided as histogram distributions pre-aggregated inside the probe (see below).

- Probe-aggregated Metrics Histogram Slices/Time Aggregate Definitions (*TMON_PDU_TYPE_HISTDSC*):

  A general approach has been followed to store the results of pre-aggregated metrics (common core module **tmon_metric.c** – see below). In short, a given type of metric (configured in a Flow Class) has its range of values split in 1, 2 or multiple slices. This permits to optionally preserve the notion of statistical distribution (histogram), whatever be the time span in further aggregation. For the general case of more than 3 slices (N>3), the first and last slices are voluntarily unbound, respectively, at bottom and at top; the first slice ends before the specified histogram lower bound (typically covering unexpected low values) while the last slice starts at the specified histogram upper bound (typically covering those unexpected too high values) The range between lower and upper bounds is partitioned in N equal size value ranges, delimiting the histogram slices. When N==1, no distribution characteristics are remembered: the metrics measurements form a simple aggregate ranging from lower (possibly unbound) to upper (possibly unbound) configured values. When N==2, some heuristic is used to determine the slices based on specified lower and upper bounds. When N==3, the aggregate consists in a single slice of most probable values, framed by two unbound slices (keeping track of unexpected too low and too high respective values).

  When an histogram or single-slice aggregate is aggregated in the probe, the applicable parameters (slice count, lower bound – possibly INT32_MIN -- and upper bound – possibly INT32_MAX -- for a particular metric type and probe detected flow instance are published at destination of the collector(s) before any actual metric data.

  Only the round-trip delay metrics are (currently) implementing this histogram approach in the probe, and reported by the *TMON_PDU_TYPE_HISTDSC and TMON_PDU_TYPE_HISTDTA PDU units*. Histograms are also provided by the probe for the distribution of IPv4 packet or datagram sizes. But this last is specific and part of the protocols statistics gathering and reporting (embedded in the overall *TMON_PDU_TYPE_FLCNTRS* PDU types).

  The routine *TmcProbeMetricInstanceNew()* (C source file *tmc_metric.c*) computes the set of respective slice bounds (via common TmonDelayHistoSplit() routine of core C source file tmon_delay.c), stores (or updates) the characteristics of every metric instance slice and outputs them to the corresponding results log file.

- Probe-aggregated Metrics Data (*TMON_PDU_TYPE_HISTDTA*):

  In-line with their published histogram/aggregate characteristics, the probes accumulates the statistics synthetic results of every measurement in their corresponding histogram slice: incrementing the population, updating the sum and the sum of square, possibly updating the min or max. At the required frequency, the

probe publishes these four values of its not-empty histogram slices over the time period, and reset them for the next period.

The collector (*TmcPduDecodeHistogramsData()*) first maps the probe flow ID to is collector-assigned unique value. When probe flow isn't yet known, the record is IGNORED.

Then the collector (*TmcMetricHistogramDecode()*) retrieves the corresponding stored histogram record (*tmcMetricDictp*), verifies its description and replaces the stored record slice with values for the new period.

Then it re-computes the slice average and outputs those data in the corresponding results log file.

- TCP Connections detailed Monitoring Data (*TMON_PDU_TYPE_TCPCONN*):

  Whether being a regular update or the final data after closure, each TCP connection record is handled the same way.

  When probe flow isn't yet known, the record is IGNORED.

  Through *TmcPduDecodeTcpConnection()*, the several monitoring variables (when present, otherwise being zero) have their variable-length encoding parsed into a working *tmc_tcpconn_stats_t* structure. Then TmcStatsTCPConnectionUpdate() prepares the corresponding record string to be output to the corresponding results log file.

- FTP File Transfer detailed Monitoring Data (*TMON_PDU_TYPE_FTPXFER*):

  Whether being a regular update or the final data after closure, each TCP connection record is handled the same way.

  When probe flow isn't yet known, the record is IGNORED.

  Through *TmcPduDecodeFtpXfer()*, the several monitoring variables (when present, otherwise being zero) have their variable-length encoding parsed into a working `tmc_ftpxfer_stats_t` structure. Then *TmcStatsFTPFileXferUpdate*() prepares the corresponding record string to be output to the corresponding results log file.

1.3.7 One-Way Flows Observations Consolidation and Measurement

The main processing performed by the collector component is the handling of one-way flow observations.

A tricky job is the decoding of the *TMON_PDU_TYPE_PKTOBS* compact PDU units, for extracting per-packet (per reassembled datagram) partial observations (probe FlowID, signature, timestamp(s) and size). See section 4.4.6 below for details.

The probe FlowID is replaced by the collector-assigned unique value.

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 (*TmcPktObsRetrieve*() in *tmc_delay.c*). Because the few bytes of content signature risks to collide with that of another packet 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 (*tmon_dict.c*). It is made of BTree (balanced 2-3 tree, where each tree node has 2 or 3 children, *tmon_btree.c*) 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 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 (or first and last timestamps of a fragmented datagram),
- 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 (*TmcPktObsComplete*() in *tmc_delay.c*). 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 (*TmcDelPartialObsCleanup*() in *tmc_delay.c*) for finding ancient incomplete records (*TmcDelIncompleteObs*() in *tmc_delay.c*).

- 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 (*TmcOutputRecord*()) in , each in their log file corresponding to the exception type (lost: TMC_RECORD_TYPE_1WLOST, dropped: TMC_RECORD_TYPE_1WDROPD, or missed: TMC_RECORD_TYPE_1WMISSG), or these exception counters are aggregated by the collector (*Tmc1WDelayAggrOutput*() in *tmc_delay.c*).

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.

2. TRAFMON C-CODE ON-LINE SOFTWARE COMPONENTS

2.1 TRAFMON MEASUREMENT MECHANISM

2.1.1 Probing

All measurements conducted by trafMon come from the inspection of IP (IPv4) traffic packets captured by probe interfaces (C source file *tmp_interface.c*).

The packet capture relies on the standardised libpcap 1.4 module which, in its Linux implementation, stores the packet in a possibly quite large kernel-resident circular buffer. In trafMon, each probing interface can be configured with a storage capacity large enough to cope with foreseeable peaks in high rate traffic flows. Such user configuration tuning must be derived from a trade between the overall portion of computer RAM dedicated to this usage and the traffic bursts anticipated at all the probing interface of a same probe computer. Only the packet selected by the BSD capture filter assigned to each interface will actually enter its kernel-resident capture buffer.

Such buffer capacity is set by the user in terms of number of number of (1600 bytes) Ethernet frames.

Packet capacity `bufPacketCount`	Maximum Frame Size `snapLen`	Consumed RAM Bytes
70 000	1600	112 000 000
1 000 000	1600	1 600 000 000

Table 2: Per Probe Capture Interface Kernel Buffer Memory Consumption

Care must however been taken in tuning such buffering for each interface. RAM is also required for the probe processing (internal data structures, but also shared memory buffer between father and child processes of the trafMon probe program) and file I/O (although limited on probe dedicated computing platforms), as well as the potential output queue of un-acknowledged trafMon PDU's at destination of (temporarily unreachable) collectors.

By hiding this Linux specific kernel interface behind a general portable API, the libpcap does unfortunately not reflect the actual amount of RAM that has been reserved. This could well be only half of the requested size (see section XXX for a hint of indirect feedback).

Being hidden inside the libpcap API, the per-interface packet buffer is not directly accessible to the trafMon probe program, running in user space. And, for the sake of reaching the *stringent performance requirements* imposed by the high and sustained data

rate of the EO ground network, the actual content of the vast majority of capture packets will **not undergo any further copy** inside the probe program user space.

Packets are therefore analysed one after the other.

Each captured packet is systematically dissected at all possible layers of its protocol stack (IPv4, ICMP, UDP, TCP, DNS, NTP, SNMP, FTP [and HTTP]): C source file *tmp_pkt_dissect.c*.

It is not required to assign an IP address to the probing interface. The only mandatory operation is to configure such interfaces as UP. This way, packet capture will occure in stealth mode: the probe will not be visible on the LAN segments it is connected to, and cannot be a target for attack at IP layer. Furthermore, the packet inspection and analysis does not rely on trust placed on information field (e.g. announced data lengths), to avoid that the probe be impacted when capturing a malevolent forged packet.

Although the probe requires packet capture privilege to run, it is not mandatory to run it as root super-user:

- produce a /opt/tmon/bin/**tmon_probe** executable binary;
- create a specific Linux *group*, e.g. **pcap** and Linux *runtime account for the probe*, e.g. **probe** having pcap as primary group
- Assign pcap as executing group to the **tmon_probe:**

```
# chgrp pcap /opt/tmon/bin/tmon_probe
# chmod 0750 /opt/tmon/bin/tmon_probe
```

- Assert the CAP_NET_RAW, CAP_NET_ADMIN capabilities for the **tmon_probe** binary:

```
# setcap cap_net_raw,cap_net_admin =eip /opt/tmon/bin/tmon_probe
# getcap /opt/tmon/bin/tmon_probe
/opt/tmon/bin/tmon_probe = ap_net_admin,cap_net_raw+eip
```

- The use of sched_setaffinity(2) also requires CAP_SYS_NICE capability**:**

2.1.2 Filtering for Flow Classes Matching

During protocol dissection, global probe-wide counters are updated.

The results of a packet dissection permit to apply all combined predicates of all defined filters from all applicable configured flow classes. This filtering method has been specifically implemented in trafMon probe, in the most efficient way., (C source file *tmp_flowfilter.c*) Indeed, all flow calls applicable to a given probe interface provides a filter combining predicates on its protocol fields (Px). In its most general form, such filter combines up to three levels 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)

Flow class filters are not necessarily mutually exclusive: a packet may well match several flow classes and, therefore, participate to different measurements.

A same protocol field can therefore be ruled by multiple predicates coming from different flow class filter expressions.

The design driver for efficient flow class filtering relies therefore on the following optimisation strategies:

- All predicates relative to a same protocol field are tested altogether, one after the other.
- As soon as a predicate test permits to conclude on the success or rejection of an entire filter expression of a flow class, all its remaining predicates are disabled for the packet under inspection.
- Systematic resort to switch/case C code constructs permits to avoid the overhead of long series of if/then/else if ... exhaustive testing sequences.

Each time a predicate is tested, its result recursively updates (from top to bottom) the 1, 2 or 3-layers expression tree of connectives for the corresponding class.

Once the set of applicable flow classes has been determined for a first or single fragment packet, the probe (father process) can decide whether the packet

- will be ignored or
- will require only its dissected field information for further analysis and measurement,
- will require to keep also its actual payload data for specific analysis (e.g. checksum of fragmented datagram.

2.1.3 Retaining FTP Data Connection Packets and IPv4 Second and Subsequent fragments

Two specific cases also occur here, for not yet fully qualified packets:

- A TCP packet between a pair of host IP addresses could be member of the Data connection of an FTP session. As it is always the case in practice, FTP data transfers always occur between the same peers as the corresponding FTP control connection. Therefore when first encountering the establishment of such FTP control connection, the peer addresses are dynamically registered (in *TmpPktKeepFtpDataConnRoot* binary search tree).

  Any remaining TCP packet (or only its connection start/end SYN / FIN / RST when only "*start-stop*" heuristic measurement is requested for the file transfer) that match this address pair has its dissected information remembered for further stateful analysis.

While encountering the end packet of the control session (in practice, always closed only after the end of a data transfer), the address pair is de-registered.

- A second or subsequent IP fragment does not contain the upper layer protocol headers. Hence its dissection and flow class membership is incomplete. However, such fragments need further analysis only when measurements are required:
    - for "*fullReassembly*" checksum verification or
    - when requested measurement is for "*allFragments*" or for reassembled "*datagram*"

  But fragments could be captured and inspected out of order. So the implemented strategy is to retain all second or subsequent fragments, except those whose first fragment has already been analysed, concluding that remaining fragments are not needed and may be dropped.

  So after analysis of a first fragment of an incomplete datagram, if none of its matched flow classes imposes any of the above two rules then its source and destination addresses and IP identifier are registered (*TmpReassRegSkippedDgram()*), permitting to reject remaining corresponding fragments (*TmpReassIsSkippedDgram*()). Those registry entries are regularly cleaned-out by timeout.

  Note that today, IP fragments are quite seldom. In the target EO network context, only the UDP SNMP packets (response PDU) are sometimes too long to fit in a 1500 bytes IP packet. Note the TCP is never fragmented, as the WAN link can afford the same MTU size as for Ethernet LAN segments.

Finally, for those first or single fragment packets which require individual reporting, the packet signature hash is computed, permitting to release the packet payload at this stage.

Note that, by dissecting the FTP Control packets, in the father probe process, the **sensitive information (i.e. the user FTP password) is voluntarily not remembered**. This way such items do not leave the captured packet residing only in the OS kernel of the probe computer.

2.1.4 Protocol Stateful Analysis and Probe Measurements

All possible processing based on single packet content is now complete.

In order to take party of the modern processor architecture (multi-core/multi-CPU parallelism), and for conducting further stateful protocol analysis and/or metrics measurements updates on a separate processor core, the rest of the processing of the retained packets is executed in a child process (C source file *tmp_child.c*).

The father process removes itself from all CPU core, but one. This CPU ID is then removed from the scheduling affinity list of the child process (*sched_getaffinity()* and *sched_setaffinity()* Linux system calls). The resulting effect is that the father assigns itself

to one CPU core only, while the child can be schedule only of all other available CPU cores.

The child process is launched only after parsing of the configured Flow Class XML definitions. The resulting internal data structures, created on the father process head, are therefore inherited as such by the child (at same virtual memory addresses as those in father's space). So the global table *tmpFlowClassTab[]* and its anchored substructures can be referred to by the same index from father and from child processes.

A double circular buffer (common core C source file *tmon_circbuf.c*) for variable size data chunks is instantiated by the father process inside a shared memory area (common core C source file *tmon_sharedmem.c*). The father is the sole writer in the circular buffer, while the child is the sole reader. Hence a circular buffer is the most efficient way to implement queuing from father to child, without need of mutual exclusion via semaphores. But the father warns its child about new availability of queued data by sending it a SIGUSR2 signal, waking it up while in blocking select() system call. Inversely, when the buffer is full, the father blocks on a usleep() (nanosleep() system call) and can also be woken up by a SIGUSR2 signal from the child.

A queued packet always requires that its dissected information (*tmp_packet_info_t*) and upper layer "transport" sub-structure (*tmp_tcp_info_t* or *tmp_dns_info_t* or *tmp_ntp_info_t* or *tmp_snmp_info_t* or *tmp_icmp_info_t*) and "application" sub-structure (*tmp_ftp_info_t* or *tmp_http_info_t*). These occupy up to three successive slots in the circular buffer.

When the IP payload is also needed for the packet further processing, a last circular buffer slot is filled with this captured upper part raw data chunk.

After having fetched all queued data slots information on a next captured packet, the child's processing is driven by the *TmpAnalyse()* routine (C source file *tmp_analyse.c*).

Where relevant, it starts by re-assembling the datagram (with or without payload data content), parking fragments in reassembly queue as long as the latest fragment isn't gathered (or reassembly timeout of **TMP_MAX_REASS_TIME=7 seconds** in *tmp_reassembly.h*).

All pending reassembly queues for one probing interface are stored in a dictionary *tmpReassTrees[interface index]* (common C source files *tmon_dict.c* and *tmon_btree.c*). Inside a given queue, fragment structures are linked in a list ordered by the sequence order of their payload data in the complete datagram.

Reassembly queues are scanned every TMP_MAX_REASS_TIME seconds for detecting timed-out queues and releasing their member fragments (via *TmpReassFailed()*).

A reassembled datagram consists in an ordered linked list of *tmp_packet_info_t* structures whose handling pointer (*pktInfop*) indicates the first fragment.

In a first analysis pass, the list of flow classes matched by the **datagram** (by the first fragment thereof) is scanned.

- If requested, the checksum is verified on reassembled datagram.
- Current flow class is attached to every fragment of the datagram (as if they matched themselves the filter).
- Depending on requirement of upper layer protocol analysis imposed by the current class:
    - TCP connection stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulTCP()*);
    - FTP control stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulFTPCtl()*);
    - HTTP stateful analysis is updated (once) on the basis of the new datagram (*TmpStatefulHTTP ()* not implemented);
    - The further <FlowClass><Condition> statements are verified on the datagram, to validate or reject the current class membership;
- Finally, the required processing synthetic flags for the datagram are extended with those applicable, for the corresponding probe interface, from the current class.

Maybe a TCP datagram not yet leading to TCP connection state analysis is member of an FTP data connection. It is then mapped to its connection by performing the missing TCP stateful analysis.

When some applicable class requests reporting every 20s and another every 30s, the period is adapted to 10s. Or the same is done for period in minutes.

Then, according to the applicable <GranularFlow> specification(s), the **datagram**

- either leads to the creation of newly discovered probe flow instance(s), which is given a probe own unique flow ID,
- or is mapped to already discovered flow instance(s)

Note that it is not precluded that a same packet participates to the measurements made for more than one flow instance.

Knowing the flow instance(s) the datagram belongs to, the relevant flow instance statistics counters can then be updated at flow instance level.

Then, **each subsequent fragment** undergoes the similar processing, where required by the concerned flow classes:

- The <Condition> criteria for every applicable flow class are verified or the class membership is denied for the fragment;
- Fragment-based flow instance membership is determined;
- If requested per packet, corresponding flow instance statistics counters are updated.

The second analysis pass is dedicated to specific measurements:

- <OneWayDelay>: leads to the retrieval of timestamps (capture, IP Timestamp option, NTP timestamps) and size individually reported, per datagram or per fragment, to the collector(s).
- <InterPacket> delay: when measured and pre-aggregated by the probe, the delays between successive datagrams (and fragments inside a datagram) are updated (not implemented).
- <RoundTripDelay>: depending on applicable flow classes specifications, and on the detected type of datagram, following round-trip delay measurements are computed:
    - TCP SYN vs. SYN-ACK (between probe and responder) and/or

      Only once at start of a new connection; this is a true transmission delay as the SYN/ACK is replied immediately by the listener.
    - TCP RTTM peer packets mapping: double probe round-trip delays, resp. with responder and with initiator sides of the TCP connection.

      Although optional, RTTM is apparently systematically used by modern operating systems.

      This is provides upper bound of the combined delay to transmit, queue, ingest and acknowledge (possible after a delayed ack wait time) the data packets. For connection transmitting data only in one direction, only the delay between the probe and data receiver can be computed. At least with Linux systems, it has been observed that some instance of measurements produce an artificially long delay: the send the forwards three packets in a burst; all three are individually acknowledged; the first ACK comes back quickly, the second one after a longer queuing time and the third after a significantly longer time.
    - TCP Data/Ack delay is more efficiently replaced by the RTTM analysis. It has not been implemented.

      The RTTM standard mechamism has just been proposed for producing more accurate measurements, especially in the case of retransmissions. So this way of measurement can only be worse than that base on RTTM.
    - NTP-based delays from probe: with responder (NTP server) and possibly with initiator (client: this is the current NTP client polling period vis-à-vis a server).

      With the responder, this measurement typically reflects the pure 2-way transmission delay (server responds immediately).

      Delay with the initiator (NTP client) depends on the fact that the client embeds the server time of the previous NTP response (from this same server) as origin time in its next request. For servers considered stable (or bad) by a client, the client polling period (delay between successive

request/response transactions) is quite long (1024s =~17 minutes). A short value (few dozens of a second) for a significantly long period of time indicates problem in correct time synchronisation (e.g. unstable time server clock).

- DNS round-trip delay between probe and responder (DNS server).

  Depending on what is requested and whether the response is cached or not, the DNS transaction can be quick (pure 2-way transmission time) or long (cascaded resolution with potentially distant servers). So only those values near to the minimum (lower delay histogram slice) are representative of the 2-way round-trip transmission time.

- SNMP round-trip delay between probe and responder (SNMP agent).

  What is characteristic of the SNMP round-trip time is that it measures the delay of remote nodes vis-à-vis the control centre location (not necessarily representative of client-to-server network paths).

- ICMP Echo (ping) round-trip delay between probe and responder.

2.1.5 Probe PDU Protocol with Collector

There are several different types of measurement data that are reported by the probe program to the set of collectors. Some types consist in individual measurement records, other record types consist in time-regular samples of values of variables accumulating the individual measurements (updated at each occurrence of a relevant monitored packet). Each such type corresponds to its own type of probe PDU (in common source file *tmon_PDU.h*).

The general mechanism for transmission of observations by the probe is the following:

- Records are formatted by an appropriate publishing routine (in C source file *tmp_publish.c*).
- For every target collector (and/or the optional local file logging), the formatted record is appended to the PDU under construction at destination to the target collector. Each PDU type has its own table of such pending PDU per collector (*tmpPubFlowPduCollTab[], tmpPubTSPduCollTab[], tmpPub2TSPduCollTab[], tmpPubFlCtrsPduCollTab[], tmpPubHistDescrPduCollTab[], tmpPubHistDataPduCollTab[], tmpPubTCPConnPduCollTab[], tmpPubFTPXferPduCollTab[]*), dynamically allocated at start-up. There is also a standalone pending PDU per type for local saving into a buffer file (currently not exploited) with same names where the '*CollTab[]*' is replaced by '*LocalSavep'*.
- When the PDU is full, or its contained data are older than the configured "*maxPDUBuildTime*" seconds, the PDU is written to the local buffer file or passed to the sending module (C source file *tmp_transmission.c*).

- Sometimes the PDU is flushed-out earlier (flow instance descriptions must be passed to collectors before data potentially referring to them; same for histogram slice definitions before the aggregated metric data).

The PDU sending (*TmonXmitPduSend()*) obeys to sophisticated control mechanism.

When no PDU have been sent since a while, the PDU is immediately sent:

- a record is kept in the acknowledgement pending queue (*tmpXmitConfigs[collector_index]. pduSentTreep*)
- the PDU is sent over the UDP socket (*TmpXmitSendPduNow()*)
- a retry timer is armed for the PDU (*TmpXmitSendPduNow()*)

Otherwise, a next free time slot is obtained for the PDU via *TmpXmitCollTimeSlot(TMP_XMITSLOT_GET_TIME_SLOT)*. Indeed, for each target collector, a minimum of "*minTimeGap*" milliseconds delay must be respected between consecutive sending or re-sending of probe PDU's. Hence the need to assigned scheduling time slots to output pending PDU's. And a timer is armed to invoke *TmpXmitSendPduNow()* at corresponding time slot.

Each time a PDU is sent, its retry count is decreased, and its retry timer timeout delay is re-computed with based on "*TOMult*" and/or "*TOIncr*".

When the first pending PDU exhaust its "retry" count (has been sent retry+1 times and after last computed timeout value), the connectivity with the collector is declared *inLongRetryMode*:

- This particular PDU will be continuously retried at the frequency given by the initial timeout period for this collector;
- Next pending PDUs to timeout their last retry will be handled according to their type:
    - Important PDU with flow instance descriptions (*TMON_PDU_TYPE_FLDESCR*), with histogram/aggregate descriptions (*TMON_PDU_TYPE_HISTDSC*) or with FTP file transfer records (*TMON_PDU_TYPE_FTPXFER*) will be continuously retried at the longest timeout period.
    - For the other PDU types, a break border time window is computed as [start, start+ "*breakBorderTime*"] in seconds, where start is the time of oldest info in the PDU that triggered the long retry mode. This determines a period "*just before the lack of connectivity*" (hence the name "*at the border of a break*"). PDU's whose oldest record fit inside this break border time window are also continuously retried at their longest timeout period: indeed they may contain observations illustrating the degradation of the traffic conditions.
    - Those other PDU types with data younger than the break border are finally dropped: those which continue to accumulate information (flow counters *TMON_PDU_TYPE_FLCNTRS*), which contain detailed individual observations (per packet/datagram *TMON_PDU_TYPE_PKTOBS* or per individual delay measurement *TMON_PDU_TYPE_DELOBS* or individual

TCP connection details *TMON_PDU_TYPE_TCPCONN*) or which contain less crucial measurements (current time interval of probe aggregated metrics *TMON_PDU_TYPE_HISTDTA*).

NOTES:

3) The “TOMult” parameter must be used with care, as successive timeout would increase quite quickly to unreasonably long duration.

4) When the connectivity from a probe to a collector is re-established, it can take a while for the remembered observations data arrive at the collector: longest delay = timeout * TOMult^retries + TOincr * (TOMult^retries-1 + TOMult^retries+ ... +TOMult^2+TOMult^1+1).

When no PDU has been sent by the probe since “*heartBeatDelay*” seconds (*TmonXmitHeartBeat()*), a fake empty heart beat is sent (*TmonXmitHeartBeatSend()*), without registration as no acknowledged will be expected (C source file *tmp_transmission.c*).

The collector program receives the probe PDUs (*TmcXmitValidateAndAckPDU()* in C source file *tmc_transmission.c*), validates its shape, type and CRC, updates the corresponding probe known status (C source file tmc_probes.c) and, when it isn’t a fake heart beat empty PDU, sends back an acknowledge with full redundancy (2 times the PDU ID and first byte of CRC).

Then, if the PDU hasn’t been received yet (tracked in probeInfop->rcvdPduDict), it is pushed at end of the *tmcInputQp* input queue (instance of the common pointer based circular buffer implemented by core *tmon_circ_pbuf.c*).

2.1.6 Collector Further Processing and Output

In addition to its role in centralising the probe observations, the collector should have three functions:

- Merging information from several probes:
  - assigning a unique identifier to probe dynamically discovered flow instances;
  - matching partial per-packet (per datagram) observations records (with size and list of timestamps) in order to produce one complete record for a same data unit observed at different locations;
- Pre-computing certain metrics in order to detect abnormalities in the traffic performance in near real-time (such event condition detection could be implemented later);
- Further aggregation of instant measurements to reduce the amount of information pushed to the database -- not implemented:
  - this feature won’t be needed in case the volume of measurements produced by the probes is reasonable;

- there is no universal criteria, besides those already implemented upstream by the probes, that can determine which metrics must be computed (once for all) by the collector at the exclusion of all others that could be derived for the then destroyed raw observations;
- if needed, this function could be best implemented outside the online collector C program, via scripts applied to the post processing of the collector output raw data files; this would offer the necessary flexibility in addressing the specific needs of different use case environments;

Therefore, besides for one-way flow measurements, the collector primarily acts as a formatter of the observations provided by the probe into corresponding types of output files.

And for the specific case of individual packet/datagram observations, its stores the partial observations in a dictionary with sophisticated ability to retrieve stored items “near in time” to a given search key.

The several probe PDU types are decoded by dedicated routines in the C source file *tmc_pdu_decoder.c*.

- Flow instance descriptions (*TMON_PDU_TYPE_FLDESCR*):

  Retrieves from the registry (*probeInfop->prFlowIdMap*) based on the corresponding probe Flow ID.

  If found and same description, simply maps to collector-assigned globally unique ID.

  If not found or new definition, remove potential ancient record; generate a new collector-assigned globally unique ID: the UNIX timestamp at the millisecond; keep track of the new flow description (*TmcFlowInstanceNew()*) and writes the flow instance description in corresponding output file (*TmcFlowInstanceOutput()*) in C source file *tmc_flowinstance.c*).

- Protocol counters records per flow instance (*TMON_PDU_TYPE_FLCNTRS*):

  When probe flow isn’t yet known, the record is IGNORED.

  Decodes the variable length fields of each record and invokes *TmcStatsFlowInstanceUpdate*(): Outputs those sub-records that have non zero counters (C source file t*mc_statistics.c*):

  - IPv4 counters: *TmcStatsFlowIPUpdate()*
  - IPv4 packet sizes distribution: *TmcStatsFlowIPDistrbUpd()*
  - ICMP counters: *TmcStatsFlowICMPUpdate()*
  - UDP Counters: *TmcStatsFlowUDPUpdate()*
  - TCP counters: *TmcStatsFlowTCPUpdate()*
  - FTP counters: *TmcStatsFlowFTPUpdate()*

- [HTTP counters: *TmcStatsFlowHTTPUpdate()* not implemented]

- Per packet/datagram individual observations (potentially partial) (*TMON_PDU_TYPE_PKTOBS*):

  The TmcPduDecodePktObs() routine implements a specific processing that is detailed below.

  When the collector reconstitute a complete packet/datagram observation record, with size and all expected hop timestamps, it outputs it via *TmcPduDecodePktObs()*

- Per delay individual records (*TMON_PDU_TYPE_DELOBS*):

  Those individual round-trip delay measurements are optionally produced, not aggregated inside the probe, and delivered to the collector. But these aren't (yet) further processed because they would be used for collector–based event detection and, possibly, collector aggregation, which are not implemented.

  Therefore, those PDU are currently simply dissected by a debugging trace routine, without output yet. Nevertheless, round-trip delays are provided as histogram distributions pre-aggregated inside the probe (see below).

- Probe-aggregated Metrics Histogram Slices/Time Aggregate Definitions (*TMON_PDU_TYPE_HISTDSC*):

  A general approach has been followed to store the results of pre-aggregated metrics (common core module **tmon_metric.c** – see below). In short, a given type of metric (configured in a Flow Class) has its range of values split in 1, 2 or multiple slices. This permits to optionally preserve the notion of statistical distribution (histogram), whatever be the time span in further aggregation. For the general case of more than 3 slices (N>3), the first and last slices are voluntarily unbound, respectively, at bottom and at top; the first slice ends before the specified histogram lower bound (typically covering unexpected low values) while the last slice starts at the specified histogram upper bound (typically covering those unexpected too high values) The range between lower and upper bounds is partitioned in N equal size value ranges, delimiting the histogram slices. When N==1, no distribution characteristics are remembered: the metrics measurements form a simple aggregate ranging from lower (possibly unbound) to upper (possibly unbound) configured values. When N==2, some heuristic is used to determine the slices based on specified lower and upper bounds. When N==3, the aggregate consists in a single slice of most probable values, framed by two unbound slices (keeping track of unexpected too low and too high respective values).

  When an histogram or single-slice aggregate is aggregated in the probe, the applicable parameters (slice count, lower bound – possibly INT32_MIN -- and upper bound – possibly INT32_MAX -- for a particular metric type and probe detected flow instance are published at destination of the collector(s) before any actual metric data.

  Only delay metrics are computed and aggregated in the probe.

The routine *TmcProbeMetricInstanceNew()* (C source file *tmc_metric.c*) computes the set of respective slice bounds (via common TmonDelayHistoSplit() routine of core C source file tmon_delay.c), stores (or updates) the characteristics of every metric instance slice and outputs them to the corresponding results log file.

- Probe-aggregated Metrics Data (*TMON_PDU_TYPE_HISTDTA*):

  In-line with their published histogram/aggregate characteristics, the probes accumulates the statistics synthetic results of every measurement in their corresponding histogram slice: incrementing the population, updating the sum and the sum of square, possibly updating the min or max. At the required frequency, the probe publishes these four values of its not-empty histogram slices over the time period, and reset them for the next period.

  The collector (*TmcPduDecodeHistogramsData()*) first maps the probe flow ID to is collector-assigned unique value. When probe flow isn't yet known, the record is IGNORED.

  Then the collector (*TmcMetricHistogramDecode()*) retrieves the corresponding stored histogram record (*tmcMetricDictp*), verifies its description and replaces the stored record slice with values for the new period.

  Then it re-computes the slice average and outputs those data in the corresponding results log file.

- TCP Connections detailed Monitoring Data (*TMON_PDU_TYPE_TCPCONN*):

  Whether being a regular update or the final data after closure, each TCP connection record is handled the same way.

  When probe flow isn't yet known, the record is IGNORED.

  Through *TmcPduDecodeTcpConnection()*, the several monitoring variables (when present, otherwise being zero) have their variable-length encoding parsed into a working *tmc_tcpconn_stats_t* structure. Then TmcStatsTCPConnectionUpdate() prepares the corresponding record string to be output to the corresponding results log file.

- FTP File Transfer detailed Monitoring Data (*TMON_PDU_TYPE_FTPXFER*):

  Whether being a regular update or the final data after closure, each TCP connection record is handled the same way.

  When probe flow isn't yet known, the record is IGNORED.

  Through *TmcPduDecodeFtpXfer()*, the several monitoring variables (when present, otherwise being zero) have their variable-length encoding parsed into a working *tmc_ftpxfer_stats_t* structure. Then TmcStatsFTPFileXferUpdate() prepares the corresponding record string to be output to the corresponding results log file.

2.2 PROBE: MAIN DATA STRUCTURES

2.2.1 Capture Interfaces

MODULE:

tmp_interface.c / tmp_interface.h

```
/* Interface activity state at previous capture stats sampling */
#define TMP_IF_NOMINAL  0 /* receiving sustainable pkt rate - initial default*/
#define TMP_IF_SILENT   1 /* no packet received */
#define TMP_IF_DROPPING 2 /* was dropping filtered packets  */

typedef struct tmp_cap_if {
  pcap_t              *ifHandle;
  uint8_t              ifIdx;
  uint16_t             ifId;          /* XML interface ID: for PDU */
  char                *ifName;        /* release through xmlFree() */
  char                *ifDescr;       /* release through xmlFree() */
  char                 ifFilter[STRMAX+1];
  struct bpf_program   ifFltrPgm;     /* compiled packet ifFilter expression */
  int                  ifFileNo;      /* I/F file descriptor for select() */
  int                  ifPcapSnapLen; /* Snap length "suggested" to libpcap */
                                      /* see below for observed linux values */
  int                  ifPcapBufPkts; /* of libpcap/kernel internal buffering */
  int                  ifPcapBufSize; /* of libpcap/kernel internal buffering */
                                      /* see below for observed linux values */
  uint32_t             ifPcapPktRcvd; /* Last sampled pcap_stat.ps_recv */
  uint32_t             ifPcapPktDrops;/* Last sampled pcap_stat.ps_drop */
  uint32_t             ifPcapPktIfDrps;/* Last sampled pcap_stat.ps_ifdrop */
  float                ifPcapPktRate; /* Last computed Captured Pkt/sec */
  float                ifPcapDropRate;/* Last computed Dropped  Pkt/sec */
  float                ifPcapIfDrpRate;/* Last computed Dropped  Pkt/sec */
  uint64_t             ifPcapStatTime;/* Time of last pcap_stats() refresh */
  char                 ifPcapStatSnmpDT[12]; /* When compiled with SNMP: */
                                    /* SNMP DateAndTime formatted of StatTime*/
  uint8_t              ifPrevState;   /* IF_NOMINAL=0, IF_SILENT, IF_DROPPING */
  /* Keep track of previously received packet on capture interface */
  /* ==> Removing subsequent duplicate by SPAN port */
  bpf_u_int32          ifPrevCapLen;
  bpf_u_int32          ifPrevLen;
  const u_char        *ifPrevPkt; /* due to Linux circular buffer, the prev. */
                                  /* slot won't be re-used yet at next packet*/
} tmp_cap_if_t;
/*
 *  Table of active capture interfaces on probe
 */
#define TMP_MAX_CAP_IF         10  /* Max # of capture interfaces per probe  */
```

```
extern tmp_cap_if_t   tmpCapIfTab[TMP_MAX_CAP_IF];
extern uint8_t        tmpCapIfCnt; /* use type int instead of uint8_t
                                    * to make it a watched Integer32
                                    * in the NetSNMP embedded (sub-)agent
                                    */
```

```
/*
 * Bit Mask of active capture interfaces (for select())
 */
extern fd_set         tmpCapFdSet;
extern int            tmpCapMaxFD;
/*
 * Using a capture file instead of network interface(s)
 */
extern uint8_t        tmpCapfromFile;   /* boolean: <CapFile> is applicable */
extern uint8_t        tmpCapFileDoDelay;/* boolean: delay processing of
                                           successive packets from capture
                                           file in order to respect their time
                                           schedule */
extern tmon_timeval_t tmpCapFileTmOfst; /* Delay to add to capture time of
                                           every packet from capture file:
                                           equal to delta time between first
                                           packet and start time of its
                                           processing by the probe */
/*
 * On LINUX with mmap'ed ring buffer, following mappings have been observed,
 * and captured packets are aligned on 16 bytes boundaries (hence the steps)
 *
 * snaplen=140 leads to caplen=122 (-14 ethernet => IP len=108)
 * snaplen=142 leads to caplen=138 (-14 ethernet => IP len=124)
 * snaplen=154 leads to caplen=138 (-14 ethernet => IP len=124)
 * snaplen=156 leads to caplen=138 (-14 ethernet => IP len=124)
 * snaplen=157 leads to caplen=154 (-14 ethernet => IP len=130)
 * snaplen=158 leads to caplen=154 (-14 ethernet => IP len=130)
 */


/*
 * Disable following NIC Card flags of Rx offloaded processing, when enabled
 *
 * - NO RX  "Receive Checksumming":  no checksum drop before capture
 * - NO LRO "Large Receive Offload": no reassembly before capture
 * - NO GRO "Generic Receive Offload": no reassembly before capture
 *        ("generic" GRO is a new way for LRO that deals better
 *           with forwarding and bridging)
 */
```

```
extern int tmpIfStatTimer;     /* Timer handle for the regular update of   */
                               /* per I/F counters of pkt captured/dropped */
#define TMP_IF_STAT_UPD_TIME   60000 /* 1 minute */

  /* rearm the timer */
  tmpIfStatTimer = TmonTimerTimeout(TmIfStatsUpdate, 0, TMP_IF_STAT_UPD_TIME);
```

2.2.2 Dissected Packet Information

MODULES:

tmp_pkt_dissect.h + **tmp_udptransaction.h** + **tmp_tcpconnection.h** + **tmon_probe.h**

In father: **tmp_pkt_dissect.c**

Ring buffer from father to child:

tmon_sharedmem.c / tmon_sharedmem.h

tmon_circ_buf.c / tmon_circ_buf.h

```
struct tmp_packet_info {
  uint64_t            pktID;             /* Unique identifier for this pkt */
  uint32_t            pktHighestProtocol;/* One of the above type patterns */
  uint8_t             pktSaveCount;      /* Whether and how many time this
                                            structure is saved in pending
                                            data structure */
  uint8_t             pktSanity;         /* What error/warn in pkt, if any */
  uint8_t             pktInterface;      /* ID of capture interface */
  struct pcap_pkthdr  pktPcapHdr;        /* Timestamp+capt. len+org. pkt len */
  uint16_t            pktIpTotLen;       /* Len of complete IP hdr + payload*/
  uint16_t            pktIpHdrOfst;      /* Offset in captured packet towards
                                             the start of IP Header */
  uint16_t            pktIpHdrLen;       /* Len of IPv4 hdr + IP options */
  uint16_t            pktIpOptOfst;      /* Offset in captured packet towards
                                            the start of IP options */
  uint16_t            pktIpOptLen;       /* Len of IPv4 IP options */
  uint16_t            pktTcpUdpHOfst;    /* Offset in captured packet towards
                                            the start of UDP/TCP header */
  uint16_t            pktTcpUdpHLen;     /*Length of TCP/TCP header + options*/
  uint16_t            pktPayloadOfst;    /* Offset in captured packet towards
                                            the start of payload (above
                                            pktHighestProtocol */
  uint16_t            pktPayloadLen;     /* Captured packet length
                                            after IP or TCP or UDP header */
  int16_t             pktChecksum;       /* <0: FAILED, 0: not verif., >0: OK*/
  tmp_ip_info_t       pktIpInfo;         /* Useful info fields from IP header*/
  uint16_t            pktSrcPort;        /* TCP/UDP Source Port */
  uint16_t            pktDstPort;        /* TCP/UDP Source Port */
  tmp_upper_protos_t  pktProtos;         /* Optional upper protocols fields */
#define pktTcpInfop  pktProtos.pr_tcp.pr_pktTcpInfop
#define pktFtpInfop  pktProtos.pr_tcp.u_tcp.pr_pktFtpInfop
#define pktHttpInfop pktProtos.pr_tcp.u_tcp.pr_pktHttpInfop
#define pktNtpInfop  pktProtos.pr_pktNtpInfop
#define pktSnmpInfop pktProtos.pr_pktSnmpInfop
#define pktDnsInfop  pktProtos.pr_pktDnsInfop
#define pktIcmpInfop pktProtos.pr_pktIcmpInfop
  void               *pktIpPayloadp;     /*Optionally presrvd IP Payload data*/
  tmp_pkt_flinstance_t *pktFlowInstp;    /* List of Flow Inst. pkt belongs to*/
  int16_t             pktFlowClasses[TMF_MAX_CLASSPERPKT+1];
                                         /* Indexes of FlowClass pkt belongs to*/
  char                pktProtocolName[STRMAX+1];/* Printable protocol name */
  uint8_t             pktSignature[TM_SIGNMAX];/* content hash signature */

};
typedef struct tmp_packet_info    tmp_packet_info_t;

typedef struct tmp_ip_info {
  /* IMPORTANT: These 3 first fields must stay contiguous at start */
  /*            of this structure, because used in memcmp(3) as an */
  /*            array of 10 bytes in the tree of pending fragments */
```

```
  uint32_t  ipSrcAddr;    /* in host byte order */
  uint32_t  ipDstAddr;    /* in host byte order */
  uint16_t  ipIdentifier; /* in host byte order */
  uint8_t   ipTOS;
  uint8_t   ipTTL;
  uint16_t  ipFragOffset; /* byte offset in host byte order,
                                                  ==0 means first frag */
  uint16_t  ipFragEnd;    /* 1+offset of last payload datagram byte in pkt */
  tmp_packet_info_t *ipFragNext; /* doubly linked chain of IP fragments, */
  tmp_packet_info_t *ipFragPrev; /* if any, during and after reassembly */
  uint16_t  ipDgramSz;    /* how many payload bytes in datagram/segment */
                          /*           INCLUDING transport header       */
  uint8_t   ipFragNum;    /* ordinal fragment number (first == 1) */
  uint8_t   ipFragCnt;    /* how many fragments in datagram/segment */
  uint8_t   ipLastFrag;   /* is this the last (or unique) fragment ? */
  uint8_t   ipDontFrag;   /* the IP header Don't Fragment flag */
  uint8_t   ipProtocol;   /* the IP header Protocol field */
  uint8_t   ipTScnt;      /* count of optional IP Timestamps present */
  uint32_t  ipTS[TM_IP_TS_MAX]; /* optional sequence of IP Timestamps */
} tmp_ip_info_t;

/*
 * Following is a fragment of a tmp_packet_info_t that should ideally
 * be a union of structures, but in fact consists of pointer to separately
 * allocated structures.
 * The reason for this is to reduce the volume of per packet dissected
 * information to the minimum.
 * This information is indeed transferred between the capture front-end
 * processing unit and the back-end in charge of stateful analysis and
 * production of statistics. The less info is to be transferred per packet,
 * the better the probe processing performances
 */
/*
 * Max size is for an FTP Control packet: TCP + FTP infos
 * Hence the definition of the tmp_largest_protos_t, used for
 * the sizeof() in allocating the largest structure.
 */
typedef union tmp_upper_protos {
  struct {
    tmp_tcp_info_t     *pr_pktTcpInfop;  /* Addit'l TCP info fields from hdr */
    union {
      tmp_ftp_info_t   *pr_pktFtpInfop;  /* Addit'l FTP Ctl decoded cmd/rsp */
      tmp_http_info_t  *pr_pktHttpInfop; /* Addit'l HTTP decoded cmd/rsp */
    } u_tcp;
  } pr_tcp;
  tmp_ntp_info_t       *pr_pktNtpInfop;  /* Addit'l NTP decoded timestamps */
  tmp_snmp_info_t      *pr_pktSnmpInfop; /* Addit'l SNMP decoded: Req ID */
  tmp_dns_info_t       *pr_pktDnsInfop;  /* Addit'l DNS decoded: Transact.ID */
  tmp_icmp_info_t      *pr_pktIcmpInfop; /* Addit'l ICMP info fields from hdr*/
} tmp_upper_protos_t;
/*
 * This is to get the size of a statically allocated tmp_upper_protos_t
 *
 *   Mimics the tmp_upper_protos_t,
 *   but with real struct instead of pointers
 */
typedef union tmp_largest_protos {
  struct {
```

```
    tmp_tcp_info_t     pr_pktTcpInfo;
    union {
        tmp_ftp_info_t      pr_pktFtpInfo;
        tmp_http_info_t     pr_pktHttpInfo;
    } u_tcp;
  } pr_tcp;
  tmp_ntp_info_t       pr_pktNtpInfo;
  tmp_snmp_info_t      pr_pktSnmpInfo;
  tmp_dns_info_t       pr_pktDnsInfo;
  tmp_icmp_info_t      pr_pktIcmpInfo;
} tmp_largest_protos_t;

struct tmp_icmp_info {
  uint16_t            icmpType;
  uint16_t            icmpEchoID;
  uint16_t            icmpEchoSeq;
  uint16_t            icmpNextHopMTU;
  tmp_packet_info_t   embedPktInfo; /* Dissected embedded UDP/TCP packet,
                                       if any */
};
typedef struct tmp_icmp_info tmp_icmp_info_t;

typedef struct tmp_ntp_info {
  struct timeval  ntpOrgTm;
  struct timeval  ntpRcvTm;
  struct timeval  ntpTmtTm;
  uint8_t   ntpMode; /* fromClient / fromServer / broadcast / others */
} tmp_ntp_info_t;

typedef struct tmp_snmp_info {
  uint32_t  snmpReqID;
  uint8_t   snmpType;
} tmp_snmp_info_t;

typedef struct tmp_dns_info {
  uint16_t  dnsTransID;
  uint8_t   dnsType; /* fromClient / fromServer */
} tmp_dns_info_t;

typedef struct tmp_tcp_info {
  uint32_t  tcpSeq;
  uint32_t  tcpAck;
  uint16_t  tcpWindow;

  uint8_t   tcpDirection:2;/*one of TMP_TCPDIR_UNKNOWN(==0) / _ATOB / _BTOA */
  uint8_t   tcpNewRTTM:1; /* boolean: Is tcpRTTMdelay filled-in ? */

                     /* Not used flags are 1-bit long, important flags *
                      * are in directly addressable bytes              */
  uint8_t   tcpNSflag :1; /* boolean: ECN-nonce concealment protect. RFC3540 */
  uint8_t   tcpCWRflag:1; /* boolean: Congestion Window Reduced RFC3168 */
  uint8_t   tcpECEflag:1; /* boolean: ECN Echo */
  uint8_t   tcpURGflag:1; /* boolean: */
  uint8_t   tcpPSHflag:1; /* boolean: */
  uint8_t   tcpACKflag;   /* boolean: */
  uint8_t   tcpRSTflag;   /* boolean: */
  uint8_t   tcpSYNflag;   /* boolean: */
  uint8_t   tcpFINflag;   /* boolean: */
```

```
  uint8_t   tcpFlags;     /* original Flags byte from TCP header */
  uint16_t  tcpMSS;       /* Maximum Segment Size option */
  uint8_t   tcpType;      /* Start/First Data/Retr. Data/empty Ack/End/RST */
  uint8_t   tcpWinScale;  /* Shift Count of the TCP Window Scaling */
                          /* bit 0x80 == true for SYN packet with this opt */
  uint8_t   tcpSACKperm;  /* boolean: Selective ACK Permitted option */
  uint8_t   tcpHasRTTM;   /* boolean: Round-Trip Time option present */
  uint8_t   tcpAltChksum; /* which checksum algorithm, only #0 is supported */
  uint8_t   tcpSACKcnt;   /* number of SACK blocks present */
  uint32_t  tcpSACKstrt[4];/*start of Select. Ack'ed blocks: inclusive bound */
  uint16_t  tcpSACKendOfs[4];/*RELATIVE end of Sel. Ack'ed blks: excl. bound */
  uint32_t  tcpTSval;     /* Timestamp Value */
  uint32_t  tcpTSecr;     /* Timestamp Echo Reply */
  struct timeval tcpRTTMstart; /* Valid only when tcpNewRTTM==1 */
                          /* Start time of TCP round-trip with probe
                           * that this back packet closes with its TSecr
                           */
} tmp_tcp_info_t;

/*
 * Dissected useful Info of FTP Control packet
 *
 * Also used to temporarilly buffer changes to apply
 * to FTP related flow counters between:
 * - the FTP stateful analysis ofthis packet
 * - the assignment of the pkt to flow instances
 */
typedef struct tmp_ftp_info {
  uint8_t   ftpType:3;    /* Command / Response / Other */
  uint8_t   truncated:1; /* whether the ftpText is truncated */
  uint8_t   addr:1;       /* whether the ftpDataAddr is present */
  uint8_t   port:1;       /* whether the ftpDataPort is present */
  union {
    uint8_t uc_ftpCmd;
    char    uc_ftpReplyCode[3];
  } u_code;
#define ftpCmd       u_code.uc_ftpCmd
#define ftpReplyCode u_code.uc_ftpReplyCode
  union {
    struct {
      uint32_t dp_ftpDataAddr;
      uint16_t dp_ftpDataPort;
    } dp;
    char     ua_ftpText[TMON_FTP_STRMAX+1];
  } u_arg;
#define ftpDataAddr u_arg.dp.dp_ftpDataAddr
#define ftpDataPort u_arg.dp.dp_ftpDataPort
#define ftpText     u_arg.ua_ftpText
  /* packet state info for flow stats counters */
  uint32_t newCtlConn:1;
  uint32_t newPassive:1;
  uint32_t newActive:1;
  uint32_t dirList:1;
  uint32_t xferRestart:1;
  uint32_t xferGet:1;
  uint32_t xferPut:1;
  uint32_t xferGetFailed:1;
  uint32_t xferPutFailed:1;
```

```
  uint32_t cmdFailed:1;
  uint32_t cipherFailed:1;
  uint32_t loginFailed:1;
  uint32_t dataAbort:1;
  uint32_t cleanClose:1;
  uint32_t dirtyClose:1;
  uint32_t crypted:1;
  uint32_t noLogin:1;
  uint32_t noCommand:1;
  uint32_t noFileXfer:1;
  uint32_t fileXfer:1;
} tmp_ftp_info_t;
```

2.2.3 IP Fragments to Skip

MODULE:

In father: **`tmp_reassembly.h / tmp_reassembly.c`**

```
typedef struct tmp_dgram_id {
  uint8_t  srcDstId[10]; /* key: src IP | dst IP | IP id */
  time_t   creatTime;
} tmp_dgram_id_t;

/*
 * Balanced Tree of recent IPv4 datagram identifiers whose
 * second and subsequent fragment do not participate to
 * the probe flow measurements:
 *
 * These can be skipped after basic IPv4 decoding
 */
static tmon_dict_t **tmpReassSkipTree;

/*
 * Initisation of Fragment skip contexts:
 *  Allocates one fragment skip dictionary per capture I/F
 *
 *  Exit(1) upon memory allocation failure
 */
void
TmpReassSkipInit()
{
  int i;

  tmpReassSkipTree = (tmon_dict_t**)malloc(tmpCapIfCnt * sizeof(tmon_dict_t*));
  if(!tmpReassSkipTree) {
    FATAL "OUT OF MEMORY in allocating Reassembly context dictonaries" END;
  }

  for(i = 0; i < tmpCapIfCnt; i++) {
    tmpReassSkipTree[i] = TmonDictCreate(TmpReassSkipCmp);

    if(!tmpReassSkipTree){
      FATAL "OUT OF MEMORY in allocating Reassembly context dictonaries" END;
    }
  }
```

```
  (void)TmonTimerTimeout(TmpReassSkipClean, 0, TMP_MAX_REASS_TIME*1000);
}
```

2.2.4 Flow Class Parsed Specifications

MODULE:

`tmp_flowclass.h + tmon_probe.h / tmp_flowclass.c`

Coding of Reporting Period:

```
/*
 * Probe observations sending period for FlowClass,
 *  Also time interval for probe pre-aggregation of measurement
 *
 *    either TMF_PERIOD_NONE: send every measurement unit produced, when ready
 *    or     time for histogram slice, sent at that rate
 *
 *    Values are structured as low byte for deci-seconds (10s units)
 *                             high byte for deci-minutes (10m units)
 *    the 6 low-order bits of a byte cover a period of six units (1 min/hour)
 *            bit #0 == 10 sec/min offset
 *                #1 == 20 sec/min offset
 *                #2 == 30 sec/min offset
 *                #3 == 40 sec/min offset
 *                #4 == 50 sec/min offset
 *                #5 == 60 sec/min offset
 *    This way, different interval specs (from different applicable flow
 *    classes) can be combined to determine the finest requested period:
 *      + The different specs are OR'ed together to define resulting period
 *      + Resulting high order byte is ignored when low order byte is non-null
 *      + Actual period is the smallest distance between asserted bits
 *
 *  Example:
 *  A)
 *     + one flow class requests every 30 m: high byte==00 1 0 0 1 0 0 low b==0
 *     + one flow class requests every 60 m: high byte==00 1 0 0 0 0 0 low b==0
 *   Result:
 *     Both patterns are OR'ed: result = high byte==00 1 0 0 1 0 0 low b==0
 *     ==> actual period is 30 sec
 *  B)
 *     + one flow class requests every 30 s: high byte==0 low bits 1 0 0 1 0 0
 *     + one flow class requests every 20 s: high byte==0 low bits 1 0 1 0 1 0
 *     + one flow class requests every 20 m: high byte==00 1 0 1 0 1 0 low b==0
 *   Result:
 *     + high byte ignored, low byte combined through bitwise OR:
 *        high byte == 0 low bits == 1 0 1 1 1 0
 *     + because bit#1 and bit#2 are both asserted, actual period is 10 seconds
 */
#define TMF_PERIOD_NONE 0x0000 /* Send individual measurements when produced */
                               /* Don't send counters */
#define TMF_PERIOD_10s  0x003f /* 10 seconds: all 10s-slots are true */
#define TMF_PERIOD_20s  0x002a /* 20 seconds: every 2 10s-slots are true */
#define TMF_PERIOD_30s  0x0024 /* 30 seconds: every 3 10s-slots are true */
#define TMF_PERIOD_60s  0x0020 /*  1 minute : only sixth 10s-slot is true */
#define TMF_PERIOD_10m  0x3f00 /* 10 minutes: all 10m-slots are true */
```

```
#define TMF_PERIOD_20m  0x2a00 /* 20 minutes: every 2 10m-slots are true */
#define TMF_PERIOD_30m  0x2400 /* 30 minutes: every 3 10m-slots are true */
#define TMF_PERIOD_60m  0x2000 /*  1 hour   :  only sixth 10m-slot is true */

#define TMF_PERIOD_MASK 0x3f3f /* mask for any period value */
#define TMF_PERIOD_SECs 0x003f /* mask to check if period has seconds slots*/
#define TMF_PERIOD_MINs 0x3f00 /* mask to check if period has minutes slots*/
/* Resulting time interval */
#define TMF_INTRVL_10s  0x0001
#define TMF_INTRVL_20s  0x0002
#define TMF_INTRVL_30s  0x0004
#define TMF_INTRVL_60s  0x0008
#define TMF_INTRVL_10m  0x0100
#define TMF_INTRVL_20m  0x0200
#define TMF_INTRVL_30m  0x0400
#define TMF_INTRVL_60m  0x0800
#define TMF_INTRVL20_30s 0x0006/* mask to check if interval has 20s & 30s */
#define TMF_INTRVL20_30m 0x0600/* mask to check if interval has 20m & 30m */
```

Flow Class Overall Information:

```
struct tmp_flowclass {
  uint16_t                flowClassId; /* TrafMon-wide <FlowClass> ID number */
  uint16_t                flowClassIdx;/* Index of this FlowClass for probe  */
  char                   *flowClassName; /*<FlowClass> Name string           */
  char                   *flowClassDescr;/*<FlowClass> description          */
                                         /* These strings require xmlFree()  */
  tmp_flow_if_t          *flowClassIf[TMP_MAX_CAP_IF];
                                        /* <FlowFilter> expression data     */
                                        /* structures foreach I/F of  probe */
  uint8_t                 flowClassCond; /* Connective type for opt. list of */
                                        /* addtional conditions on stateful */
                                        /* context information              */
  uint8_t                 flowClassStPrdCnt; /* # used slots in below table  */
  tmff_predicate_t        flowClassStatePreds[TMF_MAX_CLSSTATPRED];
                                        /* The opt. state predicates that   */
                                        /* complement the FlowFilter expr.  */
                                        /* on individual packets            */
  uint8_t                 flowDoChecksum;   /* Boolean, whether or not the   */
                                            /* UDP/TCP checksum has to be    */
                                            /* verified where possible:      */
                                            /* fully capt'd pkts &full reass.*/
  uint8_t                 flowDoReassmbly;  /* Boolean, whether or not IP    */
                                            /*     reassembly is necessary   */
  uint16_t                flowMeasIntrval;  /* Period to send data samples   */
                                            /*  Also interval for aggregation*/
  uint8_t                 flowDoMeasDelay;  /*Whether and which delay to meas*/
  uint8_t                 flowRTTSide;      /* RTT betw. prb and which end(s)*/
  uint8_t                 flowDoAggrDelay;  /* Boolean, are the measurements */
                                            /*to be aggregated before sending*/
  tmon_histospec_t        flowHisto;        /* Slices for aggregated delays */
  uint8_t                 flowSendCounters; /* does probe send pkt or dgram  */
                                            /* counters for flow to collector*/
  uint8_t                 flowReportTCPConn;/* Are TCP connections for flow: */
                                            /* - ignored                     */
                                            /* - individually reported       */
                                            /* - or grouped and reported     */
  uint8_t                 flowReportFTPTfrs;/* Are FTP sessions for flow:    */
```

```
                                        /* - ignored                      */
                                        /*       TMF_TCPFTPREPORT_NONE    */
                                        /* - individually reported        */
                                        /*   (with their file transfers)  */
                                        /*       TMF_TCPFTPREPORT_EACH    */
                                        /* - or grouped and reported      */
                                        /*       TMF_TCPFTPREPORT_GRP     */
                                        /* For data connection:           */
                                        /* - Whether only start/stop pkts*/
                                        /*       TMF_TCPFTPREPORT_BOUND   */
                                        /* - or whether every packet      */
                                        /*      ~TMF_TCPFTPREPORT_BOUND   */
  uint8_t                flowAnalyseProto; /* Which stateful analysis to    */
                                        /*   apply to matching packets    */
  uint8_t                flowObsUnit;     /* Which timestamping strategy   */

  uint8_t                flowHopCnt;      /* number of hops for FlowClass  */
  char                  **flowHopNames;   /* ordered list of FlowClass hops*/
                                  /* free: table via free()              */
                                  /*       every string via xmlFree()    */
  tmp_sign_specs_t       flowSignSpecs;   /* pkt/dgram signing hash specs  */
};
typedef struct tmp_flowclass      tmp_flowclass_t;

/*
 * This table contains the loaded definitions of all <FlowClass> definitions
 * applicable to this probe.
 */
#define TMF_MAX_FLOWCLASS 50 /* max <FlowClass> per probe Interface*/

extern tmp_flowclass_t tmpFlowClassTab[TMF_MAX_FLOWCLASS];
```

Flow Class Information at Capture Interface level:

```
struct tmp_flow_if {
  struct tmp_flowclass *flowClassp;/* back pointer to father <FlowClass>    */
  uint16_t      flowClassIfIdx;    /* Probe interface for this Class Filter */
  uint16_t      flowGrainIdx;      /* Specifies flow granularity for obs.   */
                                   /* as index in global TmpFlowGrainTab    */
  tmff_node_t   flowClassIfFilter; /* anchor of <FlowFilter> exprression    */
                                   /* data structure for probe/if           */
  uint16_t      flowClassIfPredicates[TMF_MAX_CLASSPRED];
                                   /*
                                    * Contains the indexes, in the table of
                                    * ordered predicates of all flow classes
                                    * applicable to the capture interface,
                                    * namely tmpIfFilterPredicatesTab, of all
                                    * those predicates members of the
                                    * <FlowFilter> for this <FlowClass>.
                                    * Permits to selectively de-activate them
                                    * all when the current packet under match
                                    * has solved the FlowFilter of this
                                    * FlowClass (known to either match or
                                    * not match)
                                    */
  uint16_t      flowClassIfPredCnt;/* First free index in above table       */
  uint8_t       flowIfCaptTHopNum; /* This hop Seq.#  in the TS list        */
  uint8_t       flowIfIpTSCnt;     /* # of expected IP Timestamp            */
```

```
  uint8_t       flowIfNtpTSCnt;    /* # of expected NTP applic. TS          */
  tmp_iptshop_t *flowIfIpTSHops;   /* Table of IP TimeStamp specs           */
  tmp_ntphop_t *flowIfNtpTSHops;   /* table of NTP Time Hop specs           */

      /* Detailed Further processing Flags, to be easily combined at run    */
      /* time with those of other applicable flow classes for a given packet */
  uint16_t                doPublishHopTS;
  uint8_t                 doInterPktDelay;
  uint8_t                 doRoundTripWith;
  uint16_t                doRoundTripDelay;
  uint16_t                doAggrDelay;
  uint16_t                doStatCounters;
  uint16_t                doTCPConns;
  uint16_t                doFileXfers;
};
typedef struct tmp_flow_if        tmp_flow_if_t;

typedef struct tmp_iptshop {
  uint8_t     iptsNum;    /* ordinal # of this TS in the pkt IP Timestamp opt*/
  uint8_t     iptsHopNum; /* ordinal num. of this TS in the OneWayDelay list */
} tmp_iptshop_t;

typedef struct tmp_ntphop {
  uint8_t     ntpHopType;
  uint8_t     ntpHopNum;  /* ordinal num. of this TS in the OneWayDelay list */
} tmp_ntphop_t;
```

Flow Classes structures are stored in a table.

Each member keeps its own index in the table: `flowClassIdx`

Each Flow Class specification has a table indexed as per the `tmpCapIfTab[]` interface table (`ifIdx`): `tmpFlowClassTab[]`. For those interface where the Flow Class do apply, the corresponding table is a pointer to a `tmp_flow_if_t` structure; other entries are NULL. The `tmp_flow_if_t` structure has a back pointer to the overall `tmp_flowclass_t` structure in the `tmpFlowClassTab[]`.

The dissected packet structure has a table of pointers towards its directly or indirectly matched Flow Classes (**`tmp_pkt_dissect.h`**)

Flow Classes are parsed in the father process, and its structures allocated. Through a simple **`fork()`** system call, the entire memory image of the father process it copied for starting the child. Therefore, although not physically shared, the child obtains the complete structure of Flow Classes specifications.

2.2.5 Single-Pass Combined Flow Classes Filtering

For efficient matching a captured packet with all candidate Flow Class filters, all predicates from all possible filter expressions are checked in sequence, grouped by packet information field they involve (see section 1.3.2 above).

MODULE:

Father: **`tmp_flowfilter.c / tmp_flowfilter.h`**

```
/*
 * A <FlowFilter> if a boolean expression having up to three layers:
 * - Layer 1: set of predicates (Px) or sub-expressions (Sx) connected by
 *            AND or NAND, as in
 *                 [NOT] ( P1 AND S2 AND S3 AND P4 )
 * - Layer 2: layer of the sub-expressions (Sx) part of layer 1
 *            set of predicates (px) or sub-expressions (sx) connected by
 *            OR or NOR, as in
 *             S2::= [NOT] ( p5 OR p6 OR s7 )
 * - Layer 3:  layer of the sub-expressions (sx) part of layer 2
 *            set of predicates (prx) connected by AND or NAND, as in
 *             s7::= [NOT] ( pr8 AND pr9 )
 *         Layer 3 may only consist in predicates, not sub-expressions
 */
/*
 * NON OPTIMISED WOULD-BE IMPLEMENTATION:
 *  All predicates and anchors of sub-expressions are materialised by
 *  a tmff_node_t structure.
 *      A union u distinguishes between the fields for predicate and those
 *      for sub-expression
 *      The tmffOp operator identifier permit to know whether the node is
 *      a sub-expression connective (TMFF_AND, TMFFNAND, TMFF_OR and TMFF_NOR)
 *  Inside a layer, the heading node is always a connective, and is the start
 *  (via its tmffChildp) of a linked list of nodes, based on the
 *  respective tmffNextp pointers.
 *  Every node in the list points back to the heading connective via tmffHeadp.
 *
 *  A sub-expression is materialised by a connective node, whose tmffOper
 *  is,
 *      as <FlowFilter> starting anchor: TMFF_AND or TMFF_NAND
 *      at layer 1:                      TMFF_OR  or TMFF_NOR
 *      at layer 2:                      TMFF_AND or TMFF_NAND
 *  A connective node has a tmffChildp with the list of it components
 *  A connective node also has a tmffEltCnt constant count of the number
 *  of its components.
 *
 *  A predicate has a comparison operator, an ID of the packet field it applies
 *  to and 1 or 2 operand values to compare the packet field with
 *
 * IMPORTANT TRICK:
 *  This <FlowFilter> data structure encompasses the definition of the
 *  expression, but is also dynamically collects the partial results during
 *  the flow class matching of every packet, one-by-one.
 *
 *  Hence, the connective nodes also have a tmffRes field whose value is
 *   TMFF_UNKNOWN (0): at packet match initialisation
```

```
 *   TMFF_FALSE  (-1): as soon as one member invalidates the AND/OR connective
 *   > 0:              number of member predicates already processed for the
 *                     current packet, which do not invalidate the connective
 *            When tmffRes == tmffEltCnt, then the connective succeeds
 *
 *typedef struct tmff_node tmff_node_t;
 *
 *typedef struct tmff_node {
 *  tmff_node_t          *tmffNextp;   * to brother of the AND/OR expression *
 *  tmff_node_t          *tmffHeadp;   * to the expr. heading AND or OR node *
 *  union {
 *    struct connective {
 *      tmff_node_t *u_c_tmff_childp;  * the the sub-expression nodes list  *
 *      int          u_c_tmff_eltCnt;  * how many members in this list?     *
 *      int          u_c_tmff_res;     * partial result for cur. pkt match  *
 *#define tmffChildp u.c.u_c_tmff_childp
 *#define tmffEltCnt u.c.u_c_tmff_eltCnt
 *#define tmffRes    u.c.u_c_tmff_res
 *    } c;
 *    struct predicate {
 *      int          u_p_tmff_pkField; * on which protocol field of packet  *
 *      int32_t      u_p_tmff_val1;    * single or first comparison operand *
 *      int32_t      u_p_tmff_val2;    * optional second comparison operand *
 *#define tmffPkField u.p.u_p_tmff_pkField
 *#define tmffVal1    u.p.u_p_tmff_val1
 *#define tmffVal2    u.p.u_p_tmff_val2
 *    } p;
 *  } u;
 *} tmff_node_t;
 *
 * And all the predicates also need to be copied in a table where they are
 * tetsed in sequence upon matching a given packet for determining its flow
 * class membership.
 */
/*
 * OPTIMISED ACTUAL IMPLEMENTATION:
 * RATIONALE:
 *    Because the predicates themselves sit in a table, and refer back to their
 *    connective anchor point;
 *    And because the connective anchor points know how many members
 *    (predicates and sub-expression connectives) they have
 *   ==> Then the linked list structures do not have to keep explicit nodes
 *       for the predicates themselves. Only connectives form linked lists
 *       below their father connectives
 *
 * RESULTING IMPLEMENTATION:
 *  It is split in two parts
 *     One is a linked structure of the connectives (AND/NAND or OR/NOR) for
 *     every subexpression
 *     The other is a table of the predicates:
 *           packet field, operator, operand(s), pointer to the sub-expression
 *           anchor (used to flow back the partial result)
 *
 * Linked Structure for one <FlowClass><FlowFilter>
 *
 *  All anchors of sub-expressions are materialised by a tmff_node_t structure,
 *  representing its connective (TMFF_AND, TMFF_NAND, TMFF_OR and TMFF_NOR).
 *
```

```
 *  A sub-expression is materialised by a connective node, whose tmffConn
 *  operator is,
 *      as <FlowFilter> starting anchor: TMFF_AND or TMFF_NAND
 *      at layer 1:                      TMFF_OR  or TMFF_NOR
 *      at layer 2:                      TMFF_AND or TMFF_NAND
 *
 *  A connective node has a tmffEltCnt constant count of the number of
 *  its components (a mix of predicates and sub-expressions at the lower
 *  layer -- only predicates at layer 3).
 *
 *  A connective node has a tmffChildp with the list of its potential
 *  sub-expression components.
 *
 *  A connective node also has a tmffNextp permitting to link all brothers
 *  sub-expressions present in a same upper layer expression.
 *
 *  The anchor of the complete <flowFilter> is a TMFF_AND/TMFF_NAND without
 *  brother (tmffNextp is NULL), but whose tmffChildp starts the list at
 *  layer 1. For this, tmffNextp is replaced by tmffClassp pointing to its
 *  englobing <FlowClass> definition tmp_flowclass_t structure.
 *
 *  Each sub-expression anchor (as well as each predicate in the table) have
 *  a tmffHeadp which points back to the connective node at start of the list.
 *  This pointer permits to flow back the partial result to the upper layer.
 * IMPORTANT TRICK:
 *  This <FlowFilter> data structure encompasses the definition skeleton of
 *  the expression, but it also dynamically collects the partial results
 *  during the flow class matching of every packet, one-by-one.
 *
 *  Hence, the connective nodes also have a tmffRes field whose value is
 *   TMFF_UNKNOWN (0): at packet match initialisation
 *   TMFF_FALSE  (-1): as soon as one member invalidates the AND/OR connective
 *   > 0:              number of member predicates already processed for the
 *                     current packet, which do not invalidate the connective
 *            When tmffRes == tmffEltCnt, then the connective succeeds
 */
```

Each Flow Class filter expression is materialised by a tree of following connective nodes:

```
/*
 * Possible Logical Connectives
 */
#define TMFFCON_AND  1   /* AND(element, element, element ... ) */
#define TMFFCON_NAND 2   /* NOT AND(element, element, element ... ) */
#define TMFFCON_OR   3   /* OR(element, element, element ... ) */
#define TMFFCON_NOR  4   /* NOT OR(element, element, element ... ) */

struct tmff_node {
  union { /*
           * The TOP level AND/NAND (level == 0) has no next brother, but
           * points to its <FlowClass definition
           */
    tmff_node_t *u_n_tmff_nextp;  /* To brother sub-expression of the expr.*/
    tmp_flowclass_t
                *u_c_tmff_classp; /* To this <FlowClass> Definition   */
  } u;
#define tmffNextp  u.u_n_tmff_nextp
#define tmffClassp u.u_c_tmff_classp
```

```
  tmff_node_t        *tmffHeadp;   /* To the expr. heading connective node   */
  tmff_node_t        *tmffChildp;  /* The list of potential sub-expressions  */
  uint8_t             tmffConn;    /* Which logical connective: [N]AND/OR    */
  uint8_t             tmffLevel;   /* 0:top [N]AND, 1: [N]OR, 2: bot. [N]AND */
  uint8_t             tmffEltCnt;  /* How many members in this expression?   */
  uint8_t             tmffRes;     /* partial result for current pkt match   */
};
typedef struct tmff_node           tmff_node_t;
```

The root of this `tmff_node_t` tree is anchored in the `tmp_flowclass_t` structure.

The above nodes do not point to the predicates of the expression. This is not needed. Indeed, the structure is used to solve the filter expression for a given captured packet. Hence the packet is first checked against a predicate expression; and its result is mapped upwards in the tree. As soon as a connective node is itself fully resolved, its result in turn is propagated upwards.

All predicates (from all Flow Classes filter expressions) applicable to a given interface are stored in a table, ordered by involved packet information field. Aside this, a corresponding Boolean table marks which predicates are still enabled and which shouldn't be tested anymore, as the global list of comparisons progresses for a given packet.

Predicates point back to the instance of connective node (inside an instance of Flow Class interface structure). This permits to pass its result upwards:

```
/*
 * A table groups all comparison operations in the global set of
 * <FlowClass><FlowFilter> expressions applicable to packets from a
 * give capture interface
 *
 * A predicate element contains a definition part:
 *     Packet field ID,
 *     Comparison operator
 *     Single or first operand to compare with
 *     Optional second operand (e.g. for subnet membership: nw radix and mask
 *
 * This table is sorted by packet field ID, operator, value1, value2 in that
 * order.
 *
 * A predicate element also contains
 *     An active flag: true until the corresponding <FlowClass> is known to
 *                     match or to not match.
 *     A back pointer to heading connector, in the <FlowClass>/<FlowFilter>
 *     linked structure, representing the expression this predicate is a
 *     member of.
 *
 * When matching the flow classes for a given packet, all predicates applying
 * to a same packet field are evaluated in sequence:
 *
 *    A same predicate expression appearing in more than one
 *    <FlowClass><FlowFilter> expressions then appear as consecutive entries
 *    in the table. The test is executed once and result is flown back to
 *    all predicate instances within the <FlowClass><FlowFilter> data
 *    structures.
 *
 *    A same field is collected once, and all comparisons on it are applied
```

```
 *    as a grouped sequence; respective results being immediately flown back
 *    to corresponding predicate instances within the <FlowClass><FlowFilter>
 *    data structures.
 */
typedef struct tmff_predicate {
     tmff_node_t   *tmffHeadp;   /* Head of the sub-expression this is */
                                 /* a predicate of; point back in the  */
                                 /* corresponding FlowClass/FlowFilter */
     uint16_t       tmffPkField; /* On which protocol field of packet  */
     uint8_t        tmffOper;    /* Comparison operator on pkt field   */
     uint32_t       tmffVal1;    /* Single or first comparison operand */
     uint32_t       tmffVal2;    /* Optional second comparison operand */
} tmff_predicate_t;

/*
 * The global data structure capturing the ordered definitions of the set of
 * all predicates present in all <FlowClass>/<FlowFilter> definitions
 * applicable to every capture interfaces of this probe instance.
 *
 * Each entry in the table point back to the enveloping connective
 * tmf_node_t representingthe sub-expression this predicate is a member of.
 *
 * Each entry in the table is also pointed to by an entry in the
 * flowClassPredicates[] table part of the different tmp_if_flowclass_t
 * <FlowClass> definition record inside the global tmpIfFlowClassTab[][]
 */
static tmff_predicate_t tmpIfFlowPredicatesTab[TMP_MAX_CAP_IF]
                                        [TMF_MAX_FLOWCLASS*TMF_MAX_CLASSPRED];
/*
 * Next free entry in the above table for every probe capture interface
 */
static int                tmpIfFlowPredFreeTab[TMP_MAX_CAP_IF];
/*
 * Runtime table with a disabled flag per predicate for one interface
 *
 * All initialised at false (0) before starting a packet match
 * Records are selectively asserted when a FlowClass filter has been
 *   resolved positively or negatively
 */
static uint8_t tmpIfFlowPredDisableTab[TMF_MAX_FLOWCLASS*TMF_MAX_CLASSPRED];
```

As soon as the first predicate participating to an AND connective is TRUE, or FALSE within an OR connective, the global result of the connective is known:

- it is propagated upwards in the connectives tree;
- all remaining predicates for this connective are disabled

A similar, but simplified, mechanism is used to check the post <Condition>, this time on a class by class basis, after the further packet stateful analysis in the child process.

2.2.6 Granular Flows and Discovered Flow Instances

MODULE:

tmp_granularflow.c / tmp_granularflow.h

Granular Flow definitions are stored in a table, sorted by name to permit search by dichotomy.

```
 /*
  * Structure for storing the set of criteria defining the granularity
  * of discovered traffic flow for which to accumulate observations
  */
typedef struct tmp_flowgrain {
  char              *flowGrainName;    /* XML name ID of the <GranularFlow> */
  uint32_t           flowGrainIdx;     /* Index in the global table */
  tmp_flowgrspec_t   flowGrainSpec;
  uint32_t           flowGrainAddrMask;/* when withNetMsk == 1 */
  uint32_t           flowGrainTcpSpec; /* when perTcpSpec == 1 */
  tmon_dict_t       *flowInstancesTree;/* Balanced 2 tree with all
                                          corresponding flow instance records*/
} tmp_flowgrain_t;

/*
  * Set of flow granularity criteria specifiers
  */
typedef struct tmp_flowgrspec {
  unsigned int  perIf     :1;/* Preserve Probe interface ID in granular flow */
  unsigned int  perSource :1;/* Preserve Source IP Address in granular flow */
  unsigned int  perDest   :1;/* Preserve Dest.  IP Address in granular flow */
  unsigned int  perAddr   :1;/* Covers all traffic of given IP from/to peers */
  unsigned int  perAddPair:1;/* One granular flow per Pair of IP Addresses */
  unsigned int  withNetMsk:1;/* Compare only masked subnet part of the addr's*/
  unsigned int  perSrcPort:1;/* Preserve UDP/TCP Source Port in granular flow*/
  unsigned int  perDstPort:1;/* Preserve UDP/TCP Dest. Port in granular flow */
  unsigned int  perPort   :1;/* Preserve UDP/TCP Port in granular flow */
  unsigned int  perPrtPair:1;/* Preserve both UDP/TCP Ports in granular flow */
  unsigned int  onlyPrivPt:1;/* Only Privileged Port(s) kept in distinct flow*/
  unsigned int  perSiz200 :1;/* Preserve IP packet sizes in 200 bytes buckets*/
  unsigned int  perSiz400 :1;/* Preserve IP packet sizes in 400 bytes buckets*/
  unsigned int  perIpProto:1;/* Discriminate between UDP | TCP | Other */
  unsigned int  perIpPREC :1;/* Preserve IP Type-of-Svc preced. in gran. flow*/
  unsigned int  perIpDSCP :1;/* Preserve IP Type-of-Svc DSCP in granular flow*/
  unsigned int  perIpTOS  :1;/* Preserve IP Type-of-Svc byte in granular flow*/
  unsigned int  perIpTTL  :1;/* Preserve IP Time-To-Live in granular flow*/
  unsigned int  perIpDF   :1;/* Preserve IP Don't Fragment Flag in gran. flow*/
  unsigned int  perIpMF   :1;/* Preserve IP More  Fragment Flag in gran. flow*/
  unsigned int  perIpFrOrd:1;/* Preserve IP Frag. ordinal # in granular flow */
  unsigned int  perIpFrOfs:1;/* Preserve IP Fragment Offset in granular flow */
  unsigned int  perIcmpCls:1;/* Preserve class of ICMP Type/Code in gr. flow */
  unsigned int  perIcmpTyp:1;/* Preserve ICMP Type byte in granular flow */
  unsigned int  perIcmpTC :1;/* Preserve ICMP Type/Code in granular flow */
  unsigned int  perTcpSpec:1;/* Preserve the Packets as per TCP specifier */
} tmp_flowgrspec_t;

/*
  * Stores all <GranularFlow> definitions.
  *  Table is dynamically allocated at config. file loading time and sorted
  *  on flowGrainName.
  *
  *  ID/IDREF mapping between <GranularFlow name=xxx> and <FlowGrain ref=xxx>
```

```
 *  is resolved by dichotomic search.
 */
extern tmp_flowgrain_t *TmpFlowGrainTab;
extern int              TmpFlowGrainCount;
```

Granular Flows specifications are parsed in the father process, and its structures allocated. Through a simple **`fork()`** system call, the entire memory image of the father process it copied for starting the child. Therefore, although not physically shared, the child obtains the complete structure of Flow Grain specifications.

Flow Instances are only created and maintained in the child process, after stateful analysis of the packet and possible Flow Class membership fine tuning, based on the <`Condition`> expressions.

Derived discovered Flow Instances are identified by a key consisting in a series of bytes resulting from the concatenation of its preserved data fields, such that one can quickly retrieve the potential existing flow instance a new packet is member of.

```
/*
 * Instance of a flow at selected granularity
 */
#define TMFG_MAXKEY 50

struct tmfg_flowinstance {
  uint64_t flowUniqId;
  uint8_t  flowRefCnt; /* How many times this record is referred tor     */
                       /* by another struct or by dictionary?           */
                       /* NOTE: dissected packet/datagram are ephemeral */
                       /*       so they do NOT participate to flowRefCnt*/
  uint8_t  flowIfIdx;
  uint8_t  flowKeyLen;
  uint8_t  flowKey[TMFG_MAXKEY];
  tmp_flowgrspec_t  flowGrainSpec; /* what below fields are used and how */
                                   /* to interpret them?                 */
  uint32_t flowAddr1;
  uint32_t flowAddr2;
  uint32_t flowNwMask;
  uint16_t flowFrag;
  uint16_t flowIcmp;
  uint16_t flowPort1;
  uint16_t flowPort2;
  uint8_t  flowSzRange;
  uint8_t  flowIpProto;
  uint8_t  flowTTL;
  uint8_t  flowTOS;
  uint8_t  flowDF;
  uint8_t  flowMF;
  uint8_t  flowTCPspec;
  uint8_t  flowTCPflagsOrType;

  tmp_flow_pkt_stats_t   flowPktStats;/* from per packet/dgram counters */
  tmp_fi_del_histogram_t flowDelayHistoTab[TMON_MTR_DELAY_COUNT];
                         /*
                          * The five possible types of probe-aggregated
```

```
                             * Delay histogram:
                             * - one-way delay between interfaces of probe
                             * - round-trip time towards responder
                             * - round-trip time towards initiator
                             * - inter-datagram delay at probing point
                             *    ( unfragmented of reassembled data units)
                             * - inter-fragment delay at probing point:
                             *     between real incomplete fragments of a same
                             *     datagram -- complementary of the above one
                             */
  tmp_flclass_if_ref_t  *flowClIfRefp;/* list of reference to tmp_flow_if_t */
                                      /* of flow classes that induced this */
                                      /* flow instance */
                            /* forced 64bit time stamps, even on 32bit platform */
  int64_t                flowDetectTm;/* Time at which flow has been detected */
                                      /* == capture TS of first matched packet*/
  int64_t                flowLastseenTm;/* Last time a packet has matched */
};
typedef struct tmfg_flowinstance  tmfg_flowinstance_t;
/*
 * Anchor for a list of back references from flow instance to
 * flow class specification for the probe/interface
 *
 * This way, the measurements specifications required for the flow instance
 * can be retrieved from the flow instance itself
 */
typedef struct tmp_flclass_if_ref {
  struct tmp_flclass_if_ref *flClsIfRefNextp;
  tmp_flow_if_t             *flClsIfp;
} tmp_flclass_if_ref_t;
```

The dissected packet structure also anchors the linked list of Flow Instances it is a member of (**`tmp_pkt_dissect.h+tmon_probe.h`**):

```
/*
 * Anchor for linked list of FlowInstances attached to a given packet
 *  also reused for attaching FlowInstances to TCP connections
 */
struct tmp_pkt_flinstance {
  tmp_pkt_flinstance_t  *pktFlowNextp; /* linked list */
  tmfg_flowinstance_t   *pktFlowp; /*
                                    * A reference to the common flow instance
                                    * record, which collects observations
                                    */
};
typedef struct tmp_pkt_flinstance tmp_pkt_flinstance_t;
```

2.2.7 IP Reassembly Queues

MODULE:

`tmp_reassembly.c / tmp_reassembly.h`

There is one reassembly dictionary per capture interface.

Members are datagram units under reassembly (“reassembly queues”): more precisely, members are `tmp_packet_info_t*` as anchor of doubly linked list of per fragment packet information structures (***see section 2.2.2 above***).

One `tmp_packet_info_t` refer to successor fragment in the list via its `pktIpInfo.ipFragNext`, and to its predecessor fragment in the list via its `pktIpInfo.ipFragPrev`.

Each reassembly queue is maintained ordered by increasing IPv4 fragment offset (`pktIpInfo.ipFragOffset`).

Once reassembled, a first-to-last travel permits to count the fragments and their cumulated datagram size and to assign their ordinal fragment number (first is 1); a last-to-first back travel assigns the total count and datagram size to each.

```
/*
 * Table of BTree based TmonDict of IPv4 Fragment Collections under reassembly.
 * There is one dictionary (reassembly queue) per capture interface.
 * Entries are key'ed by {IPv4 Src Address, IPv4 Dst Address, IPv4 Identifier}.
 */
static tmon_dict_t **tmpReassTrees;

typedef struct tmp_dgram_id {
  uint8_t  srcDstId[10]; /* key: src IP | dst IP | IP id */
  time_t   creatTime;
} tmp_dgram_id_t;
```

Every `TMP_MAX_REASS_TIME=7` seconds, the pending queues are travelled through, checking whether the incomplete reassembly has time-out.

2.2.8 TCP Connection Record

MODULE:

`tmp_tcpconnection.c / tmp_tcpconnection.h`

The TCP Connection record has a series of information fields and a table of two sub-structures, one per direction. The key is such that `(tcpAddrA < tcpAddrB) or (tcpPortA < tcpPortB).`

```
/*
 * TCP Connection Identifying Key
 */
typedef struct tmp_tcp_conn_key {
  uint32_t  tcpAddrA;
  uint32_t  tcpAddrB;
  uint16_t  tcpPortA;
  uint16_t  tcpPortB;
  uint8_t   Interface;
} tmp_tcp_conn_key_t;

/*
 * TCP Connection State Information
 */
```

```
#define TMP_TCPDIR_UNKNOWN 0 /* MUST be 0, used as structure initial value */
#define TMP_TCPDIR_ATOB    1 /* MUST be 1, so that TMP_TCPDIR_ATOB-1 == 0 */
#define TMP_TCPDIR_BTOA    2 /* MUST be 2, so that TMP_TCPDIR_BTOA-1 == 1 */

struct tmp_tcp_connection {
  tmp_tcp_conn_key_t    tcpConnKey;
  int                   tcpRefCnt;   /*how many pointers refer to this struct*/
  int                   tcpConnState; /* see below */
  struct timeval        tcpConnFirstTime; /* maybe not start of conn. */
  struct timeval        tcpConnLastTime;
  tmp_tcp_conn_oneway_t tcpConnDir[2];  /* resp. [TMP_TCPDIR_ATOB-1]
                                           and [TMP_TCPDIR_BTOA-1] */
  uint8_t               tcpConnInitiator; /* 0: unknown
                                           or TMP_TCPDIR_ATOB == 1
                                           or TMP_TCPDIR_BTOA == 2 */
  uint8_t               tcpConnTerminator; /* 0: unknown
                                           or TMP_TCPDIR_ATOB == 1
                                           or TMP_TCPDIR_BTOA == 2 */
  uint8_t               tcpConnIsCounted;  /* has this connectn already */
                                           /* led to increment corresp. */
                                           /* Flow Instance Counter?    */
  uint16_t              doReport;          /* TMF_TCPCN_INTRVL period   */
                                           /* TMF_TCPCN_REGULR flag     */
                                           /* TMF_TCPCN_ATEND flag      */
                                           /* TMF_TCPCN_AGGR flag       */
  uint8_t               tcpConnUpperType;  /* what is the service this  */
                                           /* connection is linked to   */
                                           /*     see below             */
  void                 *tcpConnUpperSvc;   /* ptr to data struct with   */
                                           /* state info of upper layer */
                                           /* service context: e.g.     */
                                           /*  FTP Control session      */
                                           /*  FTP Data file transfer   */
                                           /*  HTTP session             */
  tmp_pkt_flinstance_t *tcpFlowsp;         /* List of anchors referring */
                                           /* to corresp. flow instances*/
  time_t                lastReport;        /* time of last produced     */
                                           /* regular report, if any    */
};

typedef struct tmp_tcp_conn_oneway {
  uint64_t              tcpConnDirSegmBytes;/* Tot bytes of TCP Hdr + Payload*/
  uint64_t              tcpConnDirPaylBytes;/* Tot bytes of TCP Payload only */
  uint64_t              tcpConnDir1stPlByts; /* Cumul. 1st tansm'd payld bytes*/
  uint32_t              tcpConnDirRetrPayld;/* Tot Retransmitted Payld bytes */
  uint32_t              tcpConnDirSegmCount;/* # of TCP Segments             */
  uint32_t              tcpConnDirFirstCnt; /* How many 1st transm'd DATA seg*/
  uint32_t              tcpConnDirRetrSegmt;/* # Retransmitted TCP Segments  */
  uint32_t              tcpConnDirEmAckCnt; /* How many empty Ack packets    */
  struct timeval        tcpConnDirLastTime; /* Last packet seen for direction*/
  struct timeval        tcpConnDirTSvalTime;/* Earliest Time of tcpTSval */
  struct timeval        tcpRevDirTSecrTime; /* Earliest Time of tcpTSval echo*/
  struct timeval        tcpConnDirRTTime  ; /* Round-trip t towards direction*/
  uint32_t              tcpConnDirStartSeq; /* Seq. # of SYN pkt for dir.    */
  uint32_t              tcpConnDirLastSeq;  /* ABS.Seq of Lst pkt byte in dir*/
  uint32_t              tcpConnDirLastAck;  /* ABS. Furthest ACK in rev dir  */
  uint32_t              tcpConnDirWouldAck; /* First contiguous payload byte */
                                           /* not yet seen by contiguous seg*/
```

```
                                        /* for this dir. or by highest   */
                                        /* ack in reverse direction      */
                                        /* Corresponds to the expected   */
                                        /* next ack. IN ABSOLUTE SEQ#    */
  uint32_t              tcpConnDirLastWin;  /* Scal Win of Last pk in rev dir*/
  uint32_t              tcpConnDirMaxScWin; /* Max. observed real (scaled)   */
                                        /*   sending window for this dir.*/
  uint32_t              tcpSACKstart[4];    /* ABS from latest rev. SACK opt */
  uint32_t              tcpSACKend[4];      /* ABS from latest rev. SACK opt */
  uint32_t              tcpTSval;           /* Last Tcp opt. TSval in rev dir*/
  uint16_t              tcpConnDirFirstWin; /* Init. Win from SYN in rev. dir*/
  uint16_t              tcpMSS;             /* Max. Segment Size for dir. */
  uint8_t               tcpSACKcnt;         /* # of curr. active SACK entries*/
  uint8_t               tcpConnDirWinScale; /* shift count of Win in this dir*/
                                        /* !! high order bit (0x80) is
                                         * ==1: scale proposed by rev. SYN
                                         *      but not yet agreed
                                         * ==0: scale has been agreed by
                                         *      WIN Scale opt in this dir.
                                         *      SYN
                                         */
} tmp_tcp_conn_oneway_t;
```

As for packet information record, a TCP connection is attached a linked list of anchors of directly or indirectly applicable Flow Instance records (see section 2.2.6 above).

2.2.9 FTP Control Session Record

MODULE:

`tmp_tcpconnection.c / tmp_tcpconnection.h`

The FTP Control session is made of a TCP Connection records (see section 2.2.8 above) whose

- tcpConnUpperType == `TMP_TCPUPPER_FTPCTL  1`
- tcpConnUpperSvc == `ftp_control_session_t*`

```
typedef struct ftp_control_session {
  tmp_tcp_connection_t  *ftpCtlConnp;   /* correspdng TCP Connection record */
  ftp_data_conn_t       *ftpDataConnp;  /* Data Conns List in reverse order */
  uint8_t                ftpCtlStatus;          /*       see below         */
  uint8_t                ftpCtlActFlags;        /*       see below         */
  uint8_t                ftpCtlTrMode;          /*       see below         */
  uint8_t                ftpCtlType;            /*       see below         */
  uint8_t                ftpCtlDataMode;        /*       see below         */
  uint16_t               ftpCtlDataPort;        /* Data Connection rsp port*/
  uint32_t               ftpCtlDataAddr;        /*Data Connection responder*/
  uint64_t               ftpCtlFileSz;        /* from SIZE or RETR response */
  uint64_t               ftpCtlFileOffset;      /* from REST cmd on stream */
  uint8_t                ftpCtlLastCmd;         /* Code of last seen cmd   */
  char                   ftpCtlUser[NAMEMAX+1];  /* Log'ed FTP username     */
  char                   ftpCtlWorkDir[TMON_FTP_STRMAX+1];
                                        /* Current workg directory */
```

```
  char                    ftpCtlFileName[TMON_FTP_STRMAX+1];
                                                  /*Latest referred filename*/
  char                    ftpCtlTmpStr[TMON_FTP_STRMAX+1];
                                                  /* prepared new string ... */
                                                  /* ... waiting command rsp */
} ftp_control_session_t;
```

This points back to its underlying TCP connection and anchors a list of FTP Data Connection records (see `ftp_data_conn_t` section 2.2.10 below)

2.2.10 FTP Data Connection

MODULE:

`tmp_tcpconnection.c / tmp_tcpconnection.h`

There are two types of FTP Data connections:

- One is less interesting, it contains the response of a directory listing command (`TMP_TCPUPPER_FTPLIST  2`).
- The other corresponds to a file transfer (`TMP_TCPUPPER_FTPFILE  3`).

The underlying TCP connection has

tcpConnUpperSvc == `ftp_data_conn_t*`

```
struct ftp_data_conn {
  ftp_control_session_t *ftpCtlSessionp; /* reverse pointer to Ctl session */
  ftp_data_conn_t *ftpDataConnNextp; /* prev conducted data conn for session */
  tmp_tcp_connection_t *tcpDataConnp;/* Correspdng TCP Connection record     */
  uint8_t          tcpConnUpperType; /* TMP_TCPUPPER_FTPLIST or ..._FTPFILE  */
  uint8_t          ftpDataTrMode;    /* TMP_FTPTRMODE_xxx:     see above     */
  uint8_t          ftpDataType;      /* TMP_FTPTYPE_xxx:       see above     */
  uint8_t          ftpDataConnMode;  /* TMP_FTPDMODE_xxx:      see above     */
  char             ftpCtlUser[NAMEMAX+1]; /* Log'ed FTP username            */
  char             ftpCtlWorkDir[TMON_FTP_STRMAX+1];
                                          /* Current workg directory       */
  /* Only for File Transfers:                                               */
  /* ========================                                               */
  uint8_t          ftpDataXferDir;          /* a GET or a PUT of the file  */
  char             ftpDataFileName[TMON_FTP_STRMAX+1];
                                           /* Transferred Filename         */
  uint64_t         ftpDataFileSz;          /*  from SIZE or RETR response */
  uint64_t         ftpDataFileOffset;      /* 0 or from REST cmd on stream*/
  time_t           lastReport;             /* Time of last produced        */
                                           /*   regular report, if any     */
  uint16_t         doFileXfers;            /* TMF_TCPCN_INTRVL period      */
                                           /* TMF_FILEXF_REGULR flag       */
                                           /* TMF_FILEXF_ATEND  flag       */
                                           /* TMF_FILEXF_AGGR   flag       */
};
```

The above structure points back to its underlying TCP connection, but also to the FTP Control session record. And a next pointer permits to link all data connections belonging to a same control session.

2.2.11 Packet Counters

MODULE:

`tmp_statistics.c / tmp_statistics.h / tmon_statistics.h`

Besides the global instance at the probe level (for later populating an embedded SNMP agent), the per-Flow statistics consist in:

```
typedef struct tmp_flow_pkt_stats {
  uint16_t                periodPattern;/* TMF_PERIOD_XXX */
  uint16_t                reportPeriod; /* Lowest periodPattern in seconds */
  uint8_t                 isPerPkt;/* Boolean: per Fragment or per Dgram */
  tmon_sizes_t            ipSizes; /* Sizes at IP layer: */
                                   /*   either 1st fragments only */
                                   /*   or cummulated for UDP dgram/TCP segm */
  int64_t                 lastReset;/*forced 64bit time, even on 32bit pltfrm*/
  tmon_ipv4_counters_t    ipv4;
  tmon_icmpv4_counters_t  icmpv4;  /* Either flow matched ICMP packets */
                                   /* Or reflects ICMP errors on UDP/TCP  */
  tmon_udpv4_counters_t   udpv4;
  tmon_tcpv4_counters_t   tcpv4;
  tmon_ftp_counters_t     ftp;
  tmon_http_counters_t    http;
} tmp_flow_pkt_stats_t;

typedef struct tmon_sizes {
  uint64_t           sumBytes; /* cummul. volume of IPv4 pkts (hdr+payload) */
  tmon_size_bucket_t szDistrib[TMON_SIZE_BUCKETS]; /* size distribution: */
                                /* bucket  0:    0 <= size <  200 */
                                /* bucket  1:  200 <= size <  400 */
                                /* bucket  2:  400 <= size <  600 */
                                /* bucket  3:  600 <= size <  800 */
                                /* bucket  4:  800 <= size < 1000 */
                                /* bucket  5: 1000 <= size < 1200 */
                                /* bucket  6: 1200 <= size < 1400 */
                                /* bucket  7: 1400 <= size        */
  /* OR, opt. for datagram/segment bucket  7: 1400 <= size < 1600 */
                       /* optional bucket  8: 1600 <= size < 2000 */
                       /* optional bucket  9: 2000 <= size < 3000 */
                       /* optional bucket 10: 3000 <= size < 4000 */
                       /* optional bucket 11: 4000 <= size < 5000 */
                       /* optional bucket 12: 5000 <= size < 6000 */
                       /* optional bucket 13: 6000 <= size < 7000 */
                       /* optional bucket 14: 7000 <= size < 8000 */
                       /* optional bucket 15: 8000 <= size        */
  uint8_t            szBucketsCnt; /* # of szDistrib buckets actually used */

} tmon_sizes_t;
typedef struct tmon_size_bucket {
    uint16_t  sz_lower;  /* bucket includes this lower boundary */
    uint16_t  sz_upper;  /* value just above the bucket boundary
                            ==0 when not upward bounded */
    uint16_t  sz_min;    /* actual minimal size within this bucket population*/
    uint16_t  sz_max;    /* actual maximal size within this bucket population*/
    uint16_t  sz_avg;    /* average size within this bucket population */
    uint64_t  sz_pop;    /* actual population covered by this size bucket */
```

```
    uint64_t  sz_sum;    /* sum of all size of members of this bucket */
    uint64_t  sz_sumsq;  /* sum of the square of all sizes within this bucket*/
} tmon_size_bucket_t;

typedef struct tmon_ipv4_counters {
  uint64_t reassemblyTimeout;
  uint64_t fragmentOverlap;
  uint64_t icmp;
  uint64_t udp;
  uint64_t tcp;
  uint64_t others;
} tmon_ipv4_counters_t;

typedef struct tmon_icmpv4_counters {
  uint64_t icmpCksumFailed; /* not counted as malformed */
  uint64_t icmpCksumSkipped;/* checksum not verified (!UDP/TCP or trunc.) */
  uint64_t echoRequest;
  uint64_t echoReply;
  uint64_t fragNeeded;
  uint64_t srcQuench;
  uint64_t ttlExpired;
  uint64_t reassemblyTimeout;
  uint64_t unReach;          /* except fragNeeded */
  uint64_t redirect;
  uint64_t otherErrMsg;
  uint64_t otherInfo;
} tmon_icmpv4_counters_t;

typedef struct tmon_udpv4_counters {
  uint64_t udpCksumFailed; /* not counted as malformed */
  uint64_t udpCksumSkipped;/* no checksum or not verified (frag. or trunc.) */
  uint64_t udpEmpty;
  uint64_t snmp;
  uint64_t dns;
  uint64_t ntp;
  uint64_t other;
} tmon_udpv4_counters_t;

typedef struct tmon_tcpv4_counters {
  uint64_t tcpCksumFailed; /* not counted as malformed */
  uint64_t tcpCksumSkipped;/* checksum not verified: frag. or trunccated */
  uint64_t tcpRestransmit;
  uint32_t tcpLatePkt;
  uint32_t tcpStartConns;
  uint32_t tcpCleanClose;
  uint32_t tcpDirtyClose;
  uint32_t ftpCtlConns;
  uint32_t ftpFileXfers;
  uint32_t httpFileXfers;
  uint32_t otherTcpConns;
} tmon_tcpv4_counters_t;

typedef struct tmon_ftp_counters {
  uint32_t ftpStartSessions;/* # started CTL sessions */
  uint32_t ftpCleanClose;   /* # of QUIT commands seen */
  uint32_t ftpDirtyClose;   /* # of FTP ctl session closed without QUIT */
  uint32_t ftpEncrypted;    /* # of FTP ctl session encrypted */
  uint32_t ftpNoLogin;      /* # of FTP ctl session without succ. login */
```

```
  uint32_t ftpNoCommand;    /* # of FTP ctl session without logged-in command*/
  uint32_t ftpNoFileXfer;   /* # of FTP ctl session without logged-in xfer */
  uint32_t ftpWithFileXfer; /* # of FTP ctl session with logged-in file xfer */
  uint32_t ftpActives;      /* # of PORT/EPRT/LPRT based data connections,
                                                 including Directory Listing */
  uint32_t ftpPassives;     /* # of PASV/EPSV/LPSV based data connections,
                                                 including Directory Listing */
  uint32_t ftpDirList;      /* # directory listing */
  uint32_t ftpGets;         /* # succeeded RETR of files */
  uint32_t ftpPuts;         /* # succeeded STOR/STOU/APPE of files */
  uint32_t ftpFailedGets;   /* # failed to complete RETR of files */
  uint32_t ftpFailedPuts;   /* # failed to complete STOR/STOU/APPE of files */
  uint32_t ftpRestarts;     /* # of REST commands seen */
  uint32_t ftpDataAborts;   /* # of ABOR commands seen */
  uint32_t ftpLoginFailed;  /* # failed USER/PASS exchanges */
  uint32_t ftpCipherFailed; /* # of AUTH command failures */
  uint32_t ftpCmdFailed;    /* # of Negative Replies to commands */
} tmon_ftp_counters_t;

typedef struct tmon_http_counters {
  /* TBD */
} tmon_http_counters_t;
```

2.2.12 Histograms and Delay Metrics

MODULES:

tmp_delay.c / tmp_delay.h

tmon_metric.h / tmon_metric.c

Delay histogram split: **tmon_delay.c / tmon_delay.h**

2-way delays can be aggregated in the probe. Configurable histogram specifications determine whether the aggregate is a single slice or a multi-slices histogram where, typically, the top and bottom slices are unbound and all intermediate slices are partitions of equal size of a specified range between and upper and a lower bounds.

See sections 2.4.2 below and 4.4.8 below for the generic data structures and principles.

For round-trip delays, special data structures are used to store the one-way packet record, waiting to be matched with its reverse peer. This is needed for ICMP Echo, NTP, DNS, SNMP and TCP SYN/ACK, but not for TCP RTTM whose round-trip is deduced during TCP connection stateful analysis.

```
/*
 * Round-trip Time related one-way packet information,
 * for whatever 2-way protocol requiring to save one of the peer
 * packets.
 */
#define MAX_RTT_KEY   100

#define RTT_FIRST       1 /* First in loop */
```

```
#define RTT_ICMP_ECHO    1
#define RTT_NTP_SVR      2
#define RTT_NTP_CLI      3
#define RTT_SNMP         4
#define RTT_DNS          5
#define RTT_TCP_SYN      6
#define RTT_TCP_ACK      7

#define RTT_LAST         7 /* Last in  loop */

/*
 * NOTES:
 *
 * RTTM does not require storing packets: RTTM matching is part of
 *       TCP connection stateful analysis
 * TCP DATA/ACK does only require to store Data packet (in each direction),
 *              and not the ACK
 */
#define RTT_DIR_REQ      0
#define RTT_DIR_RSP      1

extern uint8_t TpMaxRTDelay;
extern int     TpRTDelayTimer;
/*
 * Hash tables that stores not (yet) matched one-way pkt obs, waiting
 * to see the peer pkt in reverse direction.
 *
 * First index:  RTT_xxx (protocol)
 * Second index: RTT_DIR_xxx
 */
extern tmon_hash_t *TpRTDelayHashTabs[RTT_LAST+1][2];

typedef struct tmp_1way {
  tmp_packet_info_t   *oneway_pktinfop;
  uint8_t              oneway_key[MAX_RTT_KEY+1];
  uint8_t              oneway_keylen;
  uint8_t              oneway_protocol:7;
  uint8_t              oneway_isresponder:1;
} tmp_1way_t;
/*
 * Module interface routines
 */
extern void    TpRoundTripTypeInit(uint8_t bits, int);
extern void    TpRTDelayProcess(uint8_t type, uint8_t withResponder,
                                struct timeval *start, struct timeval *end,
                                tmp_packet_info_t *pktInfop,
                                uint16_t doAggrDelay);
extern void    TmpDelayCheckPublish(void *);
```

2.2.13 Probe PDU Pending ACK

MODULE:

tmp_transmission.c / tmp_transmission.h

Per target collector configuration and state variables:

```
/*
 * Probe UDP socket(s) for communication with Collector(s)
 */
extern int    *tmpXmitSocks; /* Table: one socket per <PDUSending> */
extern int     tmpXmitSockCnt;
/*
 * Stores the runtime config params of a <PDUSending>/<SendTo>
 *
 *    Shared between tmp_publish and tmp_transmission
 */
typedef struct tmp_xmit_conf {
  /*
   * Config. parameters per Collector
   * --------------------------------
   */
  char               *collName;
  struct sockaddr_in  collAddr;
  int                 collIdx;    /* index of this entry in tmpXmitConfigs[] */
  int                 sockIdx;    /* use socket tmpXmitSocks[sockIdx] */
  int                 maxPDUSize;
  int                 maxPDUTime;
  int                 minGapMsec;
  struct timeval      minTimeGap;
  int                 heartBtDelay;
  int                 ackTimeout;
  int                 ackTimeMult;
  int                 ackTimeIncr;
  int                 sendTries; /* first send counts for 1 */
  int                 breakBorder;
  /*
   * Protocol State Variables per Collector
   * --------------------------------------
   */
  void               *pduSentTreep; /* sent PDU's to collector waiting for ACK*/
  int                 pduAckQueueCnt; /* # of sent PDU's waiting for ACK */
  uint32_t            nextPduUniqueId;
  struct timeval      lastPduTime; /* for Rate control time slots management */
  struct timeval      nextPduSlot; /* for Rate control time slots management */
  time_t              lastAckRecvd;
  /*
   * Long retry mode for per packet obs (at Border of a connectivity Break):
   *
   * = starts as soon as a PDU exhausts its allowed # of reties, while in
   *   normal mode, named PDU* below
   * = ends as soon as an PDU ACK is received for any sent PDU (meaning
   *   the central TrafMon server connectivity is restored
   * = implies continuous retry of the PDU*, that caused the switch to
   *   the long retry mode, as well as of all PDU whose pduTime is
   *   before the pduTime of PDU* + tpsConfig.breakBorderTime
   * = implies normal retry/timeout sending process for all other PDU's:
   *   those under retry and the newly published ones.
   */
  int                 inLongRetryMode;    /* boolean */
  time_t              breakBorderEndTime; /* absolute time at which the pkt obs
                                           * PDU's are no more part of the
                                           * break border boundary
```

```
                                                */

} tmp_xmit_conf_t;

extern tmp_xmit_conf_t *tmpXmitConfigs; /* Table of per-Collector runtime cfg*/
extern int              tmpXmitCollCnt; /* size above table== # of collectors*/

extern int              tmpXmitMaxPduBuiltPeriodMs;
                                        /* Minimum value, over all configured*/
                                        /*      <SendTo maxPduBuiltTime=xxx> */
                                        /* == Period (ms) for regular checks */
```

Queue of PDU pending their scheduled transmission time slot:

> This is implemented by the `tmon_timer` queue (section 2.4.6 below): a dedicated timer is launched for each PDU whose transmission is delayed to a scheduled time slot.

Queue of sent PDU pending ACK/Retry:

```
/*
 * Sent PDU Descriptor for ACK/Retry Management
 *
 * NOTE:
 */
typedef struct tmp_ack_info {
  tmon_pdu_hdr_t *pdup;      /* entire PDU starting with the tmon_pdu_hdr_t */
  uint32_t        pduID;     /* Sequential PDU # towards this collector */
  int             timerID;   /* What timer to clear when ACK is received */
  time_t          pduTime;   /* Time of oldest observations inside PDU */
  int             collIdx;   /* probe-internal index of destination collector */
  uint8_t         tryCnt;    /* Number of times PDU was sent inside current */
                             /*                    retry cycle (can be reset) */
  uint8_t         sentCnt;   /* Number of times PDU was sent */
  uint8_t         sockTryCnt;/* Number of times non-blockng sock write failed*/
  uint8_t         isLngRetry;/* If part of break border under long retry */
  int             sendTimeout;/* Current msec timeout value (can increase) */
  int             pduRecCnt; /* how many observations records in PDU */
  int             pduLen;    /* how many bytes in PDU data */
  uint8_t         pduType;   /* what type of info in the PDU */
  uint8_t         pduCRC0;   /* First byte of PDU signature, as complement */
                             /* of pduID for late ack'd ancient PDU's */
  uint8_t         isFirstContRetry; /* This PDU is the first that exhausted
                                     * the maximum retry/timeout: the first
                                     * of a connectivity break with collector.
                                     * This PDU is PERMANENTLY RETRIED at the
                                     * initial MINIAL TIMEOUT period.
                                     * Other PDUs will be either
                                     * - retried at their maximal timeout
                                     *   period
                                     * - or dropped after their max.
                                     *   retry/timeout
                                     */
} tmp_ack_info_t;
```

The actual queue is implemented by the `tmon_timer` queue (section 2.4.6 below): a dedicated timer is launched for next retry timeout for a sent PDU pending its acknowledge.

A binary search tree (`tsearch(3)`), `pduSentTreep` anchored in the per collector `tmp_xmit_conf_t` is used to retrieve the pending PDU upon ACK reception, for cancelling its timer and releasing its pending record.

2.3 COLLECTOR: MAIN DATA STRUCTURES

2.3.1 Peer Probe Records

MODULE:

`tmc_probe.c / tmc_probe.h`

Probe configuration record with state variables:

```
/*
 * Probe Statuses
 */
#define TMC_PROBE_ALIVE  0 /*== false */
#define TMC_PROBE_SILENT 1 /*== true; probe is silent since < dropObsTimeout */
#define TMC_PROBE_DROPNG 2 /*== true; probe is silent since > dropObsTimeout */
/*
 * Record of per probe information held at central processing module
 */
typedef struct tmc_probe {
  int                prID;
  char              *prName;
  char              *prDescr;
  struct sockaddr_in prAddr;       /* To validate sender of probeID PDU's */
  int64_t            prLastPDUTime;/* Youngest ref. time amongst recvd PDUs */
  int64_t            prLastRcvTime;/* When last PDU was received from probe */
  int32_t            maxIdleTime;  /* Max idle time before Peer lost */
  int32_t            maxObsDelay;  /* Max time for a probe obs to reach coll.*/
  int64_t            dropObsTimeout;/* Max. lifetime of observations in probe*/
  int64_t            lastStartSilentTime;
                                   /* When probe was last detected silent */
  int64_t            lastEndSilentTime;
                                   /* When probe was last detected (back) up */
  uint64_t           pduCount;     /* How many different PDU's received */
  uint64_t           pduTotCount;  /* How many PDU's recvd, incl. dups */
  tmon_dict_t       *rcvdPduDict;  /* Dict. of correctly received young PDU's*/
  tmon_dict_t       *prFlowIdMap;  /* Dict. mappings of flow IDs probe<->coll*/
  tmc_probe_if_t    *prIfTab;      /* Table of probe capture Interfaces      */
  uint8_t            prIfCnt;      /* Number of probe capture Interfaces     */
  uint8_t            prIdx;        /* Index of this entry in the tmcProbesTab*/
  uint8_t            prSilent;     /* Is probe conn'y declared lost :
                                    *     - since recently (TMC_PROBE_SILENT)?
                                    *     - since > dropObsTimeout
                                    *                    (TMC_PROBE_DROPPING)?
                                    */
} tmc_probe_t;
```

```
/* Probe description Array & probe total number*/
extern tmc_probe_t *tmcProbesTab;
extern int          tmcProbesCnt;
```

The probe interfaces are also recorded, to permit mapping between XML interface ID and its name and description:

```
typedef struct tmc_probe_if {
  uint16_t    ifID;
  char       *ifName;
  char       *ifDescr;
} tmc_probe_if_t;
```

2.3.2 Input PDU Ring Buffer

MODULES:

`tmc_transmission.c / tmc_pdu_decoder.c / tmc_transmission.h / tmc_probe.h`

For detecting duplicates, a trace of the correctly received (and acknowledged) PDU's is kept for a while in a `tmon_dict` (see section 2.4.3 below) anchored (`rcvdPduDict`) in the corresponding `tmc_probe_t` record.

The first copy of every correctly received PDU is appended to a ring buffer `tmon_circ_pbuf` (see 2.4.4 below, by pointer – no copy-in/copy-out).

```
/*
 * TrafMon Server Input Queue of newly received probe observations PDU's
 */
extern tmon_circ_pbuf_t *tmcInputQp;
```

2.3.3 Flow Instance Records

MODULE:

`tmc_flowinstance.c / tmc_flowinstance.h`

Each probe reported flow instance description is remembered by the collector:

- The `tmc_probe_t` record (see section 2.3.1 above) anchors a `tmon_dict` (see section 2.4.3 below) of its reported flow instances, with the probe-assigned flow ID: `prFlowIdMap`
- A global `tmon_dict` `tmcFlowInstanceDict` remembers all known flow instance records
- A unique Flow ID (UNIX timestamp at the millisecond) is assigned by the collector, which maps all incoming observations references to a probe Flow ID to its unique collector-assigned value.

- When two probes provided a flow description with same descriptive content, they are mapped to the same collector based record.

  NOTE:

  Most of the PDU observations relate to a flow that is unique per probe interface. But due to this re-mapping of flowID by the collector, observation PDU cannot be taken into account while the collector hasn't yet received the corresponding flow description record. When the collector restarts, this can cause a significant glitch in the data, or all probes have also to be restarted (which isn't nice either). In fact, the complete probe observations could more ideally be output on the basis of the (statically configured) probeID together with the original probe-assigned flowID. Anyway, the database maintains a mapping between successive flowIDs assigned to a same flow description (due to successive probe restarts) and the unique ID of the flow description.

  Also, the data tables, where long flow names are used as key in place of simple numeric value, would maybe more efficiently simply store this unique flowID, and simple join with the flow description table would be done where required. This would significantly reduce the volume of database rows and could reduce the time taken to produce reports (*to be further investigated*)

```
/*
 * Dictionary record that maps a given probe Flow Instance ID
 * to the Collector common record for the Flow Instance
 *
 * Some flow instances are distinct per probe
 * Some are common over several probes (possibly NAT'ed at some probes)
 */
typedef struct tmc_probe_flow {
  uint64_t    probeFlowId;
  tmc_flow_t *collFlowp;
} tmc_probe_flow_t;

typedef struct tmc_flow {
  tmon_flow_descr_t fDescr;
} tmc_flow_t;

tmon_dict_t *tmcFlowInstanceDict=0;
```

2.3.4 Flow Class Hops Records

MODULE:

`tmc_flowclass.c / tmc_flowclass.h`

Either the one-way delay observations are translated into actual latency values, and aggregated. Or these are reported individually for each observed data unit (fragment/datagram) as a size and a series of hop timestamps; typically the various timestamps are not all provided by the same probe (interface).

At (re)start, the collector parses those flows measuring <OneWayDelay> and providing the ordered list of hops. This information is recorded in the collector and output as a resulting data log file.

```
typedef struct tmc_hop_spec {
  char         *hopName;      /* xmlChar* attribute */
  tmc_probe_t *hopProbes[1]; /* Array of possible observing source probes
                                  for the given hop
                                Last is followed by NULL pointer
                                Allocated dynamically */
} tmc_hop_spec_t;

typedef struct tmc_flow_hops {
  struct tmc_flow_hops *flowHopsNextp; /* organised in a linked list */
  char                 *flowClassName;
  char                 *flowClassDesc;
  int32_t               flowClassID;
  uint8_t               flowHopCnt;
  tmc_hop_spec_t       *flowHops[1];/* Ordered array of hop name and probes
                                       Allocated dynamically */
} tmc_flow_hops_t;

/* Linked list of Flow Hops specifications */
extern tmc_flow_hops_t *TmcFlowHopsList;
```

2.3.5 Consolidated Packet Observations Records

MODULE:

`tmc_delay.h + tmc_delay.c`

The per-data unit (fragment or datagram) observations consist in a size and a partial list of hop timestamps.

Normally, for an observations record to be complete, corresponding partial observations records from two or more probe interfaces must be received by the collector and consolidated. In the meantime, those observations must be stored in a dictionary, keyed by the (unique) Flow ID, the packet signature by string and the youngest known timestamp.

Two observations partitions about a same data unit will generally not exactly match their respective key: the youngest timestamp of each is expected to differ by the about the one-way latency.

For this, the “near” searching of the `tmon_dict` dictionary (see section 2.4.3 below) is exploited to retrieve the stored record matching a newly received partial record.

When a search is performed through the BTree, if the key is not present, either its right or left sibling is returned. And all elements in the dictionary are also doubly linked in increasing order. So when the sibling has the same key, except for the timestamp, we can retrieve the best match record that the newly received partial observations would complement.

```
/*
 * Stores individual packet (data unit) obs (signature, timestamps, size)
 * during their gradual merge until corresponding packet observations are
 * received from all concerned probes ... or a LOST (PARTIAL, NO_SOURCE)
 * timeout occurs
 */
extern tmon_dict_t *partialPktObsDictp;

/*
 * Stores an individual packet (data unit) observations from several
 * probes under consolidation by the central colector
 */
typedef struct tmc_cons_pkt_obs {
  tmc_flow_hops_t *obsFlClsHopsp;/* Flow Class record specifng expected hops */
  uint64_t         obsFlowID;  /* Collector-assigned unique flow instance ID */
  tmon_timeval_t   lastUpdate; /* Collector time of last received probe obs */
  tmon_timeval_t   oldestTime; /* oldest among already known obs timestamps */
  uint16_t         obsSize;    /* Size (in bytes) of obs pkt/complete dgram */
  uint8_t          obsFragNum; /* Fragment number of individual pkt obs */
                               /* first==1, 0 means unfrag or reassmbld dgram*/
  uint8_t          obsTSExpCnt; /* # of expected [pairs of] timestamps */
  uint8_t          obsTSMissCnt;/* # of not yet rcvd [pairs of] timestamps */
  uint8_t          obsSignature[TM_SIGNMAX];/*Signature hash of obs data unit*/
  char             obsSignStr[(2*TM_SIGNMAX)+3];/* Signature as hex string
                                                   between [] */
  tmc_hop_time_t   obsTimestamps[1];/* Ordered list of hops with corresponding
                                       timestamp(s) */
                                  /* dynamically allocated array */
} tmc_cons_pkt_obs_t;

typedef struct tmc_hop_time {
  tmon_timeval_t       hopFirstTS;  /* First or single hop timestamp */
  tmon_timeval_t       hopLastTS;   /* optional last hop timestamp */
  uint8_t              hopFirstLast;/* boolean: double TS fisrt and last */
  uint8_t              hopAssigned; /* boolean: timestamp(s) for hop already
                                                known ? */
} tmc_hop_time_t;
```

For a same hop at packet capture probing point, the observation about a reassembled datagram consists in two different timestamps: youngest (first seen fragment) and oldest (last seen fragments); other hop observations consists only of the first timestamp.

The observation record refers to the XML FlowClass ID, and each timestamp also provides it sequence number (starting at 1) in the FLowClass hops list.

Thanks to the registration of hop lists (see section 2.3.4 above), this reference is translated into the corresponding hopName writing the record in the collector output log.

Either the complete or partial list of hop timestamps is appended to the proper output log (.1wobs, .1wlost, .1wmiss, .1wdrop suffixes), for further custom processing by the user, or the either the latency between to hops, or the counters of the three types of abnormalities

(lost, incomplete/missed, dropped), or both, are aggregated per time interval and are output as statistics. In this case the latency values are produced as histogram slices.

Because of (variable) delay for the several one-way probe observations to arrive at the collector, and due to the timeout necessary for deciding that per packet/datagram observations aren't complete, the output records are available after a non-negligible delay.

This window of time is split into successive output periods. Gradual aggregation of newly produced (complete or incomplete) records are updating the result data at the several slots of the window. When time advances, the oldest slot is appended to the output log file (.latcy, .1wct), and a new slot is created for the 'current' results.

```
/*
 * Collector-aggregated measurements related to one-way delay
 */
/* one period of data */
typedef struct tmc1w_data {
  int64_t       tc1w_lostCount;
  int64_t       tc1w_IncomplCount;
  int64_t       tc1w_DroppedCount;
  tmon_metric_t tc1w_delayHisto;
} tmc1w_data_t;

/* window of successive recent periods of data */
struct tmc_1way_aggr {
  tmc_flow_hops_t *tc1w_classHopsp; /* FlowClass config data from OneWayDelay*/
  int              tc1w_statsTimer; /* timer for regular stats reporting */
  int16_t          tc1w_reportDelay;/* delay from start of period before
                                                        outputing a record */
  int16_t          tc1w_reportPeriod;/* duration of output records */
  int8_t           tc1w_periodsCnt; /* number of successive records to keep
                                       in active window:
                                             == (delay + period -1) / period */
  tmc1w_data_t     tmc1w_window[1]; /* dynamically allocated table of
                                                   tmc1w_periodsCnt elements */
};
```

2.4 TRAFMON COMMON CORE C DATA STRUCTURES

2.4.1 Probe PDU structures

MODULE:

`tmon_PDU.h + tmon_core.h` (debugging decoding and dump in `tmon_PDU.c`)

See section 4.4 below.

2.4.2 Histograms and Metrics

MODULE:

`tmon_metric.h + tmon_core.h / tmon_metric.c`

These data structures intend to store the synthetic values characterising any time-aggregate metric or histogram slice thereof.

See also section 4.4.8 below.

```
/*
 * METRIC TYPES
 * ============
 * A. DELAY METRICS
 * ++++++++++++++++
 * Five possible instance types of probe-aggregated delay histograms:
 *
 * ==> index in table inside a flow instance record in probe.
 */
#define TMON_MTR_DELAY_1WAY       0 /* packet latency between probing
                                                                interfaces */
#define TMON_MTR_DELAY_2W_RSPDR   1 /* round-trip time towards responder */
#define TMON_MTR_DELAY_2W_INITR   2 /* round-trip time towards initiatior */
#define TMON_MTR_DELAY_INTERDGRM  3 /* inter-datagram delay */
#define TMON_MTR_DELAY_INTERFRAG  4 /* inter-fragment (of a same datagram)
                                                                    delay */
#define TMON_MTR_DELAY_COUNT      5
/*
 * B. OTHER METRICS
 * ++++++++++++++++
 *
 * TBD
 */
/*
 * Possibly Aggregated Metric Types: index to string
 */
char *TmonMetricName[] = {
  "One-way Delay",                        /* TMON_MTR_DELAY_1WAY */
  "Two-way Delay towards Responder",      /* TMON_MTR_DELAY_2W_RSPDR */
  "Two-way Delay towards Initiator",      /* TMON_MTR_DELAY_2W_INITR */
  "Inter-Datagram Delay",                 /* TMON_MTR_DELAY_INTERDGRM */
  "Delay inter Incomplete Fragments"      /* TMON_MTR_DELAY_INTERFRAG */
};

/*
 * Generic definition of an histogram
 */
typedef struct tmon_histospec {
  int32_t  hsp_lowBound; /* upper bound of lowest slice of delays histogram */
  int32_t  hsp_highBound;/* lower bound of top slice of delays histogram */
  uint8_t  hsp_sliceCnt; /* number of slices in histogram, incl. '< lowBound'*/
                         /*                      and including '>= highBound'*/
} tmon_histospec_t;

/*
 * Cover structure of a metric instance
 */
typedef struct tmon_metric {
  uint64_t           mtr_flowID;  /* ID of the measured Flow Instance */
  tmon_histoslice_t *mtr_sliceTab;/* Table of histogram slice values */
                                  /* Type of each value, hence of englobing
                                   * slice strcture, depends on metric type */
```

```
  int64_t            mtr_lastReset;/* start of time aggregation interval */
  int64_t            mtr_lastUpdate;/* to detect metric instance obsolescence*/
  tmon_histospec_t   mtr_spec;    /* stored the metric histogram description */
  uint16_t           mtr_Period;  /* Lowest periodPattern in seconds */
  uint8_t            mtr_Type;    /* TMON_MTR_XXX_YYY */
} tmon_metric_t;
/*
 * Generic aggregated statitics values of an histogram slice
 */
typedef struct tmon_histoslice {
     int32_t  sl_lower;/* bucket includes this lower  bound, maybe INT32_MIN*/
     int32_t  sl_upper;/* value just above the bucket bound, maybe INT32_MAX*/
     int32_t  sl_min;  /* actual minimal value within this bucket population*/
     int32_t  sl_max;  /* actual maximal value within this bucket population*/
     int64_t  sl_pop;  /* actual population covered by this slice bucket */
    uint64_t  sl_sumsq;/* sum of the square of all delays within this bucket*/
     int32_t  sl_sum;  /* sum of all delays of members of this bucket */
     int32_t  sl_avg;  /* average value within this bucket population */
     int8_t   sl_num;  /* Slice number (starting at 1) */
} tmon_histoslice_t;
```

2.4.3 Efficient Flexible Dictionary and BTree

MODULEs:

tmon_dict.h + tmon_core.h / tmon_dict.c

tmon_btree.h + tmon_core.h / tmon_btree.c

This is combined data structure that permits:

- efficient insertion of a new element at its ordered position,
- efficient retrieval of an existing element,
- a combination of the two above,
- travel at left or right of a retrieved element of the ordered list of members, with potential extraction, and optional release, of the specified predecessor/successor
- “near” retrieval of the element which is immediately below or above a non existing key

The dictionary is made

- of a balanced 2 tree, where intermediate nodes have 2 or 3 children, and where leaf nodes have 1, 2 or 3 anchors of elements.
- and of a doubly linked list of anchors of elements.

BTree:

```
/*
 * Implementation of a 2 BTree data structure
 *
 * Generalized from Aho, Hopcroft, Ullman.
 *  'The Design and Analysis of Computer Algorithms, Addison-Wesley 1974
```

```
 *  Section 4.10 Dictionaries and Priority Queues.
 *
 * In the algorithm above, the tree has nodes for leafs.
 * But this is redundant with the maxLeft, maxCenter of the leafs fathers,
 * except for the right leaf if it exists. Therefore, the last layer
 * of the tree is omitted and the right pointer is overloaded to contain
 * the rigth key in nodes that are fathers of leafs
 *
 * Invariants:
 *    leftp = centerp = 0 if childs are leaves
 *    maxLeft = highest key of left subtree or the key of left leaf
 *      if leftp = 0
 *    maxCenter = highest key of center subtree or the key of center
 *      leaf if leftp = 0
 *    leftp = the left subtree if not 0
 *    centerp = the center subtree if not 0
 *    rightp = the right subtree if it exists or the key of right leaf
 *      if leftp = 0
 *    Special condition arise for a single element tree,
 *    otherwise, nodes have always at least left and center
 *    subtrees if they are not father of leaves.
 *
 *    Leaves are in increasing order from left to right
 *      ==> maxLeft < maxCenter for each node
 */
typedef struct tmon_btree_node {
  struct tmon_btree_node *leftp;
  struct tmon_btree_node *centerp;
  struct tmon_btree_node *rightp;
  void              *maxLeft;
  void              *maxCenter;
  } tmon_btree_node_t;

/*
 *  Routines exported out of this module
 *
 *  When given non-NULL pointers to allocated pointers, one of the
 *       *leftp or *rightp is filled with the direct neighbour element
 *       that was already stored
 */
extern void *TmonBSearch(void *key, void **treep, int(*compar)(void *,void *),
                                 void **leftp ,void **rightp);
extern void *TmonBDelete(void *key, void **treep, int(*compar)(void *,void *));
extern void  TmonBWalk(void *treep, void (*action)(void*, VISIT, int));
extern void *TmonBFind(void *key, void *treep, int (*compar)(void *,void *),
                                  void **leftp, void **rightp);
extern void  TmonBDestroy(void **treep);
```

Dictionary:

```
/*
 * Dictionary anchor
 */
typedef struct tmon_dict {
  unsigned int   eltCnt;          /* Number of elements in the dictionary */
  int       (*cmp)(void*,void*);/* Comparison routine
                           *  defining elts order relationship:
                           * ARGS ARE POINTERS TO USER'S
```

```
                            * void* ELEMENTS, EVEN THOUGH DECLARED
                            * AS void* THEMSELVES
                            */
  void        *treep;         /* BTree as dictionary index
                            * whose elements are pointers to
                            * tmon_dict_elt_t
                            */
  tmon_dict_elt_t elts;         /* A fake element as list anchor:
                            * list head pointed to by elts.nextp
                            * list tail pointed to by elts.prevp
                            *
                            * Invariants:
                            * last->nextp  == &elts
                            * first->prevp == &elts
                            * Empty list:
                            *   elts.nextp == elts.prevp == &elts
                            */
} tmon_dict_t;
/*
 * Element anchor in the dictionary
 */
typedef struct tmon_dict_elt {
  void            *elt;  /* Needs to be first field in anchor!!
                          * Such that a pointer to an anchor is also
                          * a pointer to a void* user's element
                          */
  struct tmon_dict_elt *nextp;
  struct tmon_dict_elt *prevp;
} tmon_dict_elt_t;
/*
 *  Routines exported out of this module
 */
tmon_dict_t *TmonDictCreate(int (*compar)(void*, void*));
void  TmonDictDestroy(tmon_dict_t** dictp, void (*optFreeElt)(void*));
              /* double reference to dict for nullifying the ref */
void *TmonDictAdd(void *elt, tmon_dict_t *dictp);
void *TmonDictFind(void *eltKey, tmon_dict_t *dictp);
tmon_dict_elt_t *TmonDictNear(void *eltKey, tmon_dict_t *dictp,
               tmon_dict_elt_t **leftEltp, tmon_dict_elt_t **RightEltp);
void *TmonDictExtract(void *eltKey ,tmon_dict_t *dictp);
tmon_dict_elt_t *TmonDictNext(tmon_dict_elt_t *curEltp, tmon_dict_t *dictp,
                        int doFreeCur, void (*optFreeElt)(void*));
tmon_dict_elt_t *TmonDictPrev(tmon_dict_elt_t *curEltp, tmon_dict_t *dictp,
                        int doFreeCur, void (*optFreeElt)(void*));
unsigned int    TmonDictSize(tmon_dict_t *dictp);
```

Typical use of “near” retrieval:

```
/*
 *    A pointer to tmon_dict_el_t (i.e. a tmon_dict_el_t*) can be
 *    de-referenced in such a way to obtain the dictionary element
 *    that the user has stored (seen as a void* by the tmon_dict API):
 *    {
 *      my_element_t *myEltp;
 *      my_element_t  myKey;
 *      tmon_dict_t  *myDictp;
 *      tmon_dict_t  *found;
 *      tmon_dict_t  *atLeftp;
```

```
 *       tmon_dict_t  *atRightp;
 *
 *       myDict = TmonDictCreate(myCmp);
 *
 *       ...
 *
 *       myKey = ...
 *       myEltp = &myKey;
 *
 *       atLeftp = atRightp = 0;
 *       found = TmonDictNear(myDictp, myEltp, &atLeftp, &atRightp);
 *       if(!found) {
 *         if(atLeftp) {
 *           myEltp = *(my_element_t**)atLeftp;
 *           ...
 *         }
 *       } else {
 *           myEltp = *(my_element_t**)found;
 *           ...
 *       }
 *       ...
 *       found = TmonDictNext(myDictp, myEltp, 0, 0);
 *       if(found) {
 *         myEltp = *(my_element_t**)found;
 *         ...
 *       }
 *     }
 */
```

For travel, an existing element must first be found by “near” searching. At each step, the caller may decide to extract the current element while retrieving the previous or next one, by asserting the `doFreeCur` Boolean. Furthermore, the caller can also give a releasing routine as argument `optFreeElt`, which will be invoked on the just released element.

2.4.4 Circular Buffers

MODULEs:

Storing fixed-size pointers to elements:

`tmon_circ_bufp.h + tmon_core.h / tmon_circ_bufp.c`

Storing variable-size elements data (with copy-in/copy-out):

`tmon_circ_buf.h + tmon_core.h / tmon_circ_buf.c`

Support parallelism (multi-threading/multiprocessing) between ONE reader and ONE writer, without need of semaphores.

For the Circular FIFO buffer for fixed-size pointers to exterior data:

```
/*
 * = The maximum number of slots in the buffer is fixed after
 *   initialisation.
```

```
 * = The buffer is circular: when head/tail reaches the size of buffer,
 *   it wraps back to is base; inversely when it decreases to the base, it
 *   jumps back to the end of the buffer memory.
 * = It is also possible to POP (UN-APPEND) back the latest inserted chunk
 *   from the tail
 *   (e.g. when there is no room enough to accomodate a linked sequence
 *    of data chunks)
 *
 * MULTI-PROCESSING:
 *
 * - The writer process is the owner and sole modifier of the tail pointer
 * - The reader process is the owner and sole modifier of the head pointer
 *
 *   Upon race condition, the most critical event would be that the buffer
 *   is declared full while a slot has just been released, or declared empty
 *   while a chunk has just been inserted.
 *
 * Note: UN-APPEND is SAFE, but CAN FAIL
 *       The reader may have already consumed the block before the writer
 *       triggers its UNDO
 */
typedef struct tmon_circ_pbuf {
  void    **tcpb_slotsTab; /* Address of the start of array of pointers     */
  uint16_t  tcpb_slotsMax; /* Selectable maximum # of slots in buffer       */
                           /* frozen after the buffer has been initialised  */
  uint16_t  tcpb_slotsTail;/* Free slot for next pointer elt when Tail!=Head*/
  uint16_t  tcpb_slotsHead;/* Next pointer element when Head+1!=Tail        */
} tmon_circ_pbuf_t;
/*
 * Module interface routines
 */
tmon_circ_pbuf_t *TmonCircPBufCreate(int16_t, void *(*)(size_t));
int16_t           TmonCircPBufFreeSlots(tmon_circ_pbuf_t*);
int               TmonCircPBufIsFull(tmon_circ_pbuf_t*, int16_t);
int               TmonCircPBufIsEmpty(tmon_circ_pbuf_t*, int16_t);
int               TmonCircPBufNoSlotUsed(tmon_circ_pbuf_t*);
int               TmonCircPBufAppend(tmon_circ_pbuf_t*, void*);
void*             TmonCircPBufUnAppend(tmon_circ_pbuf_t*);
void*             TmonCircPBufNext(tmon_circ_pbuf_t*);
```

For the Circular FIFO buffer with copy-in/copy-out of variable length data, two ring structures are created: one with the fixed-size lengths of every buffered elements, the other with variable slots size, storing the elements data themselves:

```
/*
 * = Maximum size of an element data chunk is 0x7fff=32K
 * = Each data chunk in the buffer is variable in length;
 * = Respective sizes (uint16_t) of stored chunks are saved in a sibling
 *   area. Thanks to the fixed size (uint16_t) of the elements in this
 *   area, it is possible to know where starts (or ends) the chunk at head
 *   (at tail). Using a separate area for the chunk sizes is the least
 *   consuming way to know the boundaries of the chunks.
 * = The size of the buffer is fixed after initialisation.
 * = The maximum number of slots in the buffer is also fixed after
 *   initialisation.
 * = The buffer is circular: when head/tail reaches the size of buffer,
 *   it wraps back to is base; inversely when it decreases to the base, it
```

```
 *   jumps back to the end of the buffer memory.
 * = NEXT operation occurs in two steps:
 *     - the size of the oldest chunk at head is returned to permit
 *       the user to allocate a destination memory
 *     - the data chunk is copied-out to the destination memory and
 *       room is released at head.
 * = It is also possible to POP (UN-APPEND) back the latest inserted chunk
 *   from the tail
 *   (e.g. when there is no room enough to accomodate a linked sequence
 *    of data chunks)
 *
 * MULTI-PROCESSING:
 *
 * - The writer process is the owner and sole modifier of the tail pointer
 * - The reader process is the owner and sole modifier of the head pointer
 *
 *   Upon race condition, the most critical event would be that the buffer
 *   is declared full while a slot has just been released, or declared empty
 *   while a chunk has just been inserted.
 *
 * Note: UN-APPEND is SAFE, but CAN FAIL
 *       The reader may have already consumed the block before the writer
 *       triggers its UNDO
 */

typedef struct tmon_circ_buf {
  void     *tcb_dataBasep; /* Address of the start of data buffer memory     */
  uint16_t *tcb_slotsTab;  /* Address of the start of sibling array of sizes */
  uint32_t  tcb_dataSize;  /* Selectable length of the buffer memory         */
                           /* frozen after the buffer has been initialised   */
  uint32_t  tcb_dataTail;  /* Offset from dataBasep where to append the next */
                           /* data chunk: first free when dataTail!=dataHead */
  uint32_t  tcb_dataHead;  /* offset from dataBasep where oldest chunk has   */
                           /* been saved: start of first busy chunk          */
  uint16_t  tcb_slotsMax;  /* Selectable maximum # of slots in buffer        */
                           /* frozen after the buffer has been initialised   */
  uint16_t  tcb_slotsTail; /* Equiv. of dataTail inside the array of sizes   */
  uint16_t  tcb_slotsHead; /* Equiv. of dataHead inside the array of sizes   */
} tmon_circ_buf_t;

/*
 * Module interface routines
 */
tmon_circ_buf_t *TmonCircBufCreate(int32_t, int16_t, void *(*)(size_t));
int32_t          TmonCircBufFreeSize(tmon_circ_buf_t*);
int16_t          TmonCircBufFreeSlots(tmon_circ_buf_t*);
int              TmonCircBufIsFull(tmon_circ_buf_t*, int32_t, int16_t);
int              TmonCircBufIsEmpty(tmon_circ_buf_t*, int16_t);
int              TmonCircBufNoSlotUsed(tmon_circ_buf_t*);
int              TmonCircBufAppend(tmon_circ_buf_t*, int16_t, const void*);
int              TmonCircBufUnAppend(tmon_circ_buf_t*);
int16_t          TmonCircBufNextSize(tmon_circ_buf_t*);
int16_t          TmonCircBufNext(tmon_circ_buf_t*, int16_t, void*);
```

2.4.5 Hash Table

MODULE:

tmon_hash.h + tmon_core.h / tmon_hash.c

```
/*
 * Generic hash table data structure where entries are doubly-linked lists
 * of elements.
 *
 * Elements are remembered in anchor structures, starting with two pointers
 * (next and prev), that are part of doubly-linked list for a same hash value.
 *
 * The table itself is just an array of such anchors: pairs of pointers to
 * first and last of the list of elements with same hash key, fake elt pointer
 * indicating the main handle of the hash table data structure itself.
 *
 * Its size is a power of two, whose exponent value is assigned at creation
 * time and stored in a header structure together with the table anchor.
 *
 * The hash function is expected to return an array of bytes (max. 2) long
 * enough to cover the number of bits == power of table size, so the the index
 * is obtained by masking and shifting inside a uint16_t.
 *
 * Inside the doubly linked list, elements are kept in ascending order.
 * In a table slot, anchor of a list:
 * + nextp points to the head of list, nextp->prevp points to the table slot
 * + prevp points to the tail of list, prevp->nextp points to the table slot
 * + eltp ponts to the anchor heading structure of the hash data structure
 *      itself.
 * Therefore, being given the handle of the hash and any element, it is
 * possible to find its predecessor (either previous in list or tail of
 * previous non-empty slot with lower hash value) or successor (next in list
 * or head of next non-empty slot with higher hash value)
 * So an exhaustive travel (only partly ordered) is possible, asking for
 * predecessor of successor of a retrieved element.
 */

/*
 * Hash table anchor
 */
typedef struct tmon_hash {
  tmon_hashelt_t *hash_tab;          /* The actual hash table:
                                      *  nextp is the head of list
                                      *  prevp is the tail of list
                                      *  When list is EMPTY, both pointers
                                      *  contain the ADDRESS OF THEIR OWN ANCHOR
                                      *  eltp points to the tmon_hash_t itself.
                                      */
  int           (*cmp)(void*,void*);/* Comparison routine
                                      *     defining elts order relationship
                                      */
  uint8_t      *(*hash)(void*);      /* Hash function:
                                      *  Being given the address of an element,
                                      *  returns an array of bytes whose
                                      *  N leftmost bits are used as index
                                      *  in the hash table (N is the exponent
```

```
                                        *  of the table size, saved in hash_bits
                                        */
  unsigned int    eltCnt;              /* Number of stored elements */
  uint16_t        size;                /* Number of slots in hash_tab */
                                       /*           == 1 << hash_bits */
  uint8_t         hash_bits;           /*
                                        * Number of bits used from hash function
                                        * result.
                                        * The hash table is 2^hash_bits long.
                                        * MUST BE AT MOST 16 (max.2 bytes of hash,
                                        * => hash table of max. 65536 slots)
                                        */
} tmon_hash_t;
/*
 * Generic starting content of element in the data structure
 */
typedef struct tmon_hashelt {
  struct tmon_hashelt *nextp;
  struct tmon_hashelt *prevp;
  void                *eltp;
        /*
         * The actual user element.
         * For slots in the actual hash table (array), eltp points to
         * the heading tmon_hash_t of the data structure (handle)
         */
} tmon_hashelt_t;
```

Typical way of use:

```
/*
 *    typedef struct my_element {
 *      char myKey[YYY]
 *      char myData[ZZZ]
 *    } my_element_t;
 *    . . .
 *    {
 *      my_element_t    myKey;
 *      my_element_t   *foundp;
 *      my_element_t   *newOrNotp;
 *      tmon_hashelt_t *anchorp;
 *      tmon_hash_t    *myHashp;
 *      uint8_t         myHashBits = 8; // one byte hash, 256 hash table slots
 *      void          (*myfree)(void*) = free; // de-allocate an element
 *
 *      // Init
 *      myHashp = TmonHashCreate(myHashBits, myCmp, myHashFun);
 *      ...
 *
 *      // Retrieve or not found
 *      myKey.myKey = ...
 *
 *      found = TmonHashFind( &myKey, myHashp );
 *      if( found ) {
 *         ... found->myData[...] ...; // consume stored data
 *      }
 *      ...
 *
 *      // Retrieve or insert
```

```
 *       newOrNotp = malloc(sizeof(my_element_t));
 *       if( !newOrNotp ) exit(2);
 *
 *       newOrNotp->myKey = ...;
 *
 *       foundp = (my_element_t*)TmonHashSearch( newOrNotp, myHashp);
 *       if( foundp != newOrNotp) { // was already stored
 *         free(newOrNotp);
 *         ... foundp->myData[...] ...; // consume stored data
 *       } else {
 *           newOrNotp->myData[...] = ...; // data to be remembered with key;
 *           ...
 *       }
 *       ...
 *
 *       // Read-only travel
 *       for( anchorp = (my_element_t*)TmonHasNext(NULL, myHashp, 0, NULL);
 *            anchorp;
 *            anchorp = (my_element_t*)TmonHasNext(foundp, myHashp, 0, NULL);
 *         foundp = anchorp->eltp;
 *         ...
 *       }
 *       ...
 *
 *       // Travel with selective removal
 *       anchorp = (my_element_t*)TmonHasNext(NULL, myHashp, 0, NULL);
 *       while( anchorp ) {
 *         foundp = anchorp->eltp;
 *         if( ... foundp->myData[...] ... ) {
 *
 *           // found element must be EXTRACTED BUT NOT RELEASED
 *           anchorp = (my_element_t*)TmonHasNext(anchorp, myHashp, 1, NULL);
 *
 *         } else if( ... foundp->myData[...] ... ) {
 *
 *           // found element must be EXTRACTED AND RELEASED
 *           anchorp = (my_element_t*)TmonHasNext(anchorp, myHashp, 1, myfree);
 *
 *         } else {
 *           // found element must be KEPT
 *           anchorp = (my_element_t*)TmonHasNext(anchorp, myHashp, 0, 0);
 *         }
 *       }
 *     }
 */
```

2.4.6 Timers

MODULE:

`tmon_timer.h + tmon_core.h / tmon_timer.c`

Use of signal based timers in a C program is often annoying as the timeout routine executes by breaking the main program progress at any, potentially unstable, point. This

could induce parallelism in mono-thread program, hence the need to heavily protect critical sections against timeout routine potential intrusive impact.

Hence the generic `tmon_timer` module is always invoked in pure synchronism by the main program, which has to explicitly poll the the time advance at quite regular but fully controlled stable points in the code (inside the main loop).

A timer is made of a pointer to the call-back timeout routine, an opaque argument and a delay in milliseconds. When launched, the timer provides a handle that can be used to deactivate it before time exhaustion.

```
/*
 * DEFAULT max number of concurrent timeout
 *      used only when argument of TmonTimerInit() is negative or null
 */
#ifndef TMON_TIMER_POOL_SZ
#define TMON_TIMER_POOL_SZ  1000
#endif /*TMON_TIMER_POOL_SZ*/

/*
 * structure used to implement a list of
 * functions to be run in variable delays
 */
typedef struct tmon_timer {
  struct timeval     tTimerDelay;   /* absolute timeout stamp */
  int                tTimerId;      /* entry id in the list */
  void              *tTimerFunArgs; /* argument to routine */
  void             (*tTimerFunc)(void*);   /* timeout routine */
  struct tmon_timer *tTimerNext;
  } tmon_timer_t;
/*
 * Timeout list entry point:
 *    The list is ordered by growing timeout values.
 *    Each entry delay is the absolute stamp as a struct timeval.
 *    The entry also stores the function called at timeout, and its
 *      argument, and an identifier to remove it.
 */
tmon_timer_t   timerFirst; /* entry point of the list of active timers */

static tmon_timer_t *timerPool =0; /* array of timeout entries */
static tmon_timer_t *timerFree =0; /* free list of timeout entries */

/*
 * Generic timeout routine: asserts the flag pointed to by the argument
 *                          passed at timer arm time, so that the user
 *                          can easilly test whether the timer has already
 *                          fired or not
 */
void  TmonTimerSetFlag(void *flagp);
```

3. TRAFMON DATABASE PROCESSING AND REPORTING

3.1 DATABASE SCHEMA

For experimented performance reasons, the MySQL database management system (DBMS) has been selected instead of PostgreSQL (used in previous TrafMon). Selected version is 5.6.31 which correctly implements two key features:

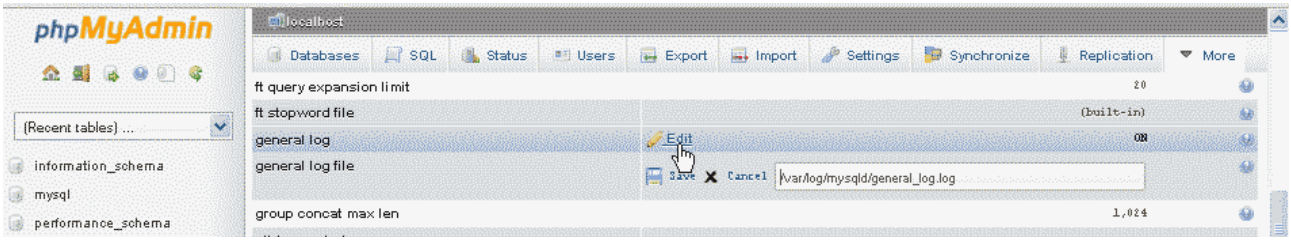
- **Tables partitioning**: permits to split a logical long table in several physical partitions, each being assigned a range of possible values for a key column field (e.g. one day of a DATETIME key field). This permits efficiently work on data belonging to a given period of time (especially the continuously updated current day) and also to efficiently DROP an aged-out daily data set all at once, without need to resort to rather slow selective DELETE operations.
- **Fractional DATETIME timestamps**: the granularity of trafMon observations is the *millisecond*.

MySQL allows to connect securely using SSL but at this time the database regular loading script use a UNIX socket preventing network utilization and the BIRT reporting tool (running locally) connects through a TCP connection to the server, only listening to localhost.

Different levels of logging are enabled in MySQL:

The detailed logging of the DBMS activity, including user denied and full dump of the “as executed” SQL statements is called **general log:**

- System Variable `general_log` is a Boolean that dynamically controls whether such **logging is enabled/disabled**
- System Variable `general_log file` is the **full pathname** of the general log file.



Error logging give information about errors that occur while the server is running (also server start and stop):

- System Variable ‘`log_error_verbosity`’ variable allows tuning of verbosity levels: **errors only, errors and warnings and errors, warnings, notes**.

Slow queries can also been detected and logged (see section 3.6.2 below).

The database, *trafMon* by default, or any one whose name starts with the prefix, is made of several tables, some being fixed, others being instantiated per aggregated time scale (**minute, hour, day**).

At their finest granularity (**1m**), those persistent measurements table are physically partitioned *on a daily basis*: this permits to cluster those data that are naturally close to each order, enhancing retrieval performances; and especially, this permits to efficiently drop aged-out detailed data chunks. But tables with coarser granularity are also partitioned: **8 days** chunks for **1h**, **31 days** chunks for **1d**. Every partition is named "pYYYYMMDD", except the last one, called pFuture, which should always be empty, but would receive data inserted, for any reason, about dates after the theoretical upper bound of the last dynamically created.

During the regular data loading (`trafMon_loader.py` see section 3.4.2), **temporary tables** are created for the bulk loading of fresh data: the sampled output log files from the collector (and the pre-processed optional NetFlow data).

Being persistent or temporary, all tables have their template counterpart in the *trafMon_template* separate pre-defined database. This database also contains all trafMon stored procedures, and the table assigning a service name to known TCP/UDP port number (with optional precedence). This last can be updated by the trafMon tool administrator.

After those bulk loads which imports data for a new day, corresponding new day partitions are created in the corresponding persistent tables (those at one minute or those with individual data records – tcp connections, file transfers).

The counters data are first aggregated in the temporary tables, whose templates are named ***xxx_aggr_tmp_template***. When needed, the database and target persistent tables are automatically created, based on their template schema from *trafMon_template.**xxx_aggr_template***.

The aggregate tables globally provide a ***population*** and, for each measurement field, the ***_sum***, ***_sumSquare***, ***_min*** and ***_max***. This allows for further aggregating the data over longer time period without losing the statistical representation (average and variance).

Distinction must be made between measurement by time unit (accumulated counter over a fixed time interval) and aggregation of individually measured values (as is the case for 2way delays).

For *delays*, the **population** is the number of individual measurements made over the interval. Therefore:

- the average for a given histogram slice and time interval is such that

```
_average = _sum / population
```

- the standard deviation (whatever it means in case of Poisson distribution) is also based on `population`:

```
_sigma = SQRT( ((population * _sumSquare) - (_sum * _sum))
                                  / (population * (population-1)) )
```

The 95 percentile for could be approximated by

```
[_min .. _average+2*_sigma]
```

For *counters*, the **population** is the number of 1min intervals during which one of the counter in the table has actually been incremented (traffic has actually been observed). Hence the population relative to the aggregated time interval (e.g. 60 minutes for hourly aggregation) gives an idea of the representativeness of the reported counter metric: *observed only over 100*(population/interval) % of the reported time*. The total population, for which to report the average, is the total duration of the reported time interval:

- the covered % of reported time period

```
coverage = 100 * population / interval
```

- the average for a time interval is such that

```
_average = _sum / interval
```

- the standard deviation (whatever it means in case of Poisson distribution – see) is also based on `population`:

```
_sigma = SQRT( ((interval * _sumSquare) - (_sum * _sum))
                                        / (interval * (interval -1)) )
```

The 95 percentile is approximated by

```
[_average*_sigma .. _average+2*_sigma]
```

3.1.1 Persistent Tables Templates

The structure of the several **persistent tables** is given below:

```
--
-- Table structure for table `aggr_exists_template`
--

DROP TABLE IF EXISTS `aggr_exists_template`;
CREATE TABLE IF NOT EXISTS `aggr_exists_template` (
  `flowID` varchar(200) NOT NULL,
  `ipctAggr` bit(1) NOT NULL DEFAULT b'0',
  `tcpctAggr` bit(1) NOT NULL DEFAULT b'0',
  `udpctAggr` bit(1) NOT NULL DEFAULT b'0',
  `ftpctAggr` bit(1) NOT NULL DEFAULT b'0',
  `icmpctAggr` bit(1) NOT NULL DEFAULT b'0',
  `ipszAggr` bit(1) NOT NULL DEFAULT b'0',
  `tcpcon` bit(1) NOT NULL DEFAULT b'0',
  `ftpxfr` bit(1) NOT NULL DEFAULT b'0',
  `twoway` bit(1) NOT NULL DEFAULT b'0',
  `oneway` bit(1) NOT NULL DEFAULT b'0',
  `lossctAggr` bit(1) NOT NULL DEFAULT b'0',
  PRIMARY KEY (`flowID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------
```

```
--
-- Table structure for table `eventtable_template`
--

DROP TABLE IF EXISTS `eventtable_template`;
CREATE TABLE IF NOT EXISTS `eventtable_template` (
  `time` datetime NOT NULL,
  `type` varchar(100) DEFAULT NULL,
  `severity` varchar(20) DEFAULT NULL,
  `entity` varchar(20) NOT NULL,
  `probeInterface` varchar(50) DEFAULT NULL,
  `flowName` varchar(200) DEFAULT NULL,
  `eventMessage` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`time`,`entity`),
  KEY `time` (`time`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `eventtable_fix_template`
--

DROP TABLE IF EXISTS `eventtable_fix_template`;
CREATE TABLE IF NOT EXISTS `eventtable_fix_template` (
  `time` datetime NOT NULL,
  `type` varchar(100) DEFAULT NULL,
  `severity` varchar(20) DEFAULT NULL,
  `entity` varchar(20) NOT NULL,
  `probeInterface` varchar(50) DEFAULT NULL,
  `flowName` varchar(200) DEFAULT NULL,
  `eventMessage` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`time`,`entity`),
  KEY `time` (`time`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(time)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `flowtable_template`
--

DROP TABLE IF EXISTS `flowtable_template`;
CREATE TABLE IF NOT EXISTS `flowtable_template` (
  `flowID` varchar(200) NOT NULL,
  `address1` varchar(18) NOT NULL,
  `address2` varchar(18) DEFAULT NULL,
  `port1` smallint(5) unsigned DEFAULT NULL,
  `port2` smallint(5) unsigned DEFAULT NULL,
  `protocol` enum('tcp','udp','icmp') DEFAULT NULL,
  `direction` enum('<','>','=') NOT NULL,
  `size` int(10) unsigned DEFAULT NULL,
  `ToSType` tinyint(3) unsigned DEFAULT NULL,
  `ToSValue` tinyint(3) unsigned DEFAULT NULL,
  `TimeToLive` tinyint(3) unsigned DEFAULT NULL,
```

```
  `DontFragment` bit(1) DEFAULT NULL,
  `MoreFragment` bit(1) DEFAULT NULL,
  `fragmentNumber` int(10) unsigned DEFAULT NULL,
  `fragmentOffset` int(10) unsigned DEFAULT NULL,
  `icmpType` int(10) unsigned DEFAULT NULL,
  `tcpType` int(10) unsigned DEFAULT NULL,
  `probeInterface` varchar(30) DEFAULT NULL,
  `comment` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`flowID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ----------------------------------------------------------

--
-- Table structure for table `ftpcttable_aggr_template`
--

DROP TABLE IF EXISTS `ftpcttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `ftpcttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(10) unsigned NOT NULL,
  `startedSessions_sum` float unsigned NOT NULL,
  `startedSessions_sumSquare` float unsigned NOT NULL,
  `startedSessions_min` int(10) unsigned NOT NULL,
  `startedSessions_max` int(10) unsigned NOT NULL,
  `cleanClosedSession_sum` float unsigned NOT NULL,
  `cleanClosedSession_sumSquare` float unsigned NOT NULL,
  `cleanClosedSession_min` int(10) unsigned NOT NULL,
  `cleanClosedSession_max` int(10) unsigned NOT NULL,
  `dirtyClosedSessions_sum` float unsigned NOT NULL,
  `dirtyClosedSessions_sumSquare` float unsigned NOT NULL,
  `dirtyClosedSessions_min` int(10) unsigned NOT NULL,
  `dirtyClosedSessions_max` int(10) unsigned NOT NULL,
  `encryptedSessions_sum` float unsigned NOT NULL,
  `encryptedSessions_sumSquare` float unsigned NOT NULL,
  `encryptedSessions_min` int(10) unsigned NOT NULL,
  `encryptedSessions_max` int(10) unsigned NOT NULL,
  `noLoginSessions_sum` float unsigned NOT NULL,
  `noLoginSessions_sumSquare` float unsigned NOT NULL,
  `noLoginSessions_min` int(10) unsigned NOT NULL,
  `noLoginSessions_max` int(10) unsigned NOT NULL,
  `noCmdSessions_sum` float unsigned NOT NULL,
  `noCmdSessions_sumSquare` float unsigned NOT NULL,
  `noCmdSessions_min` int(10) unsigned NOT NULL,
  `noCmdSessions_max` int(10) unsigned NOT NULL,
  `noFileXferSessions_sum` float unsigned NOT NULL,
  `noFileXferSessions_sumSquare` float unsigned NOT NULL,
  `noFileXferSessions_min` int(10) unsigned NOT NULL,
  `noFileXferSessions_max` int(10) unsigned NOT NULL,
  `fileXferSessions_sum` float unsigned NOT NULL,
  `fileXferSessions_sumSquare` float unsigned NOT NULL,
  `fileXferSessions_min` int(10) unsigned NOT NULL,
  `fileXferSessions_max` int(10) unsigned NOT NULL,
  `activeConnections_sum` float unsigned NOT NULL,
  `activeConnections_sumSquare` float unsigned NOT NULL,
  `activeConnections_min` int(10) unsigned NOT NULL,
```

```
  `activeConnections_max` int(10) unsigned NOT NULL,
  `passiveConnections_sum` float unsigned NOT NULL,
  `passiveConnections_sumSquare` float unsigned NOT NULL,
  `passiveConnections_min` int(10) unsigned NOT NULL,
  `passiveConnections_max` int(10) unsigned NOT NULL,
  `dirListCount_sum` float unsigned NOT NULL,
  `dirListCount_sumSquare` float unsigned NOT NULL,
  `dirListCount_min` int(10) unsigned NOT NULL,
  `dirListCount_max` int(10) unsigned NOT NULL,
  `fileGetOK_sum` float unsigned NOT NULL,
  `fileGetOK_sumSquare` float unsigned NOT NULL,
  `fileGetOK_min` int(10) unsigned NOT NULL,
  `fileGetOK_max` int(10) unsigned NOT NULL,
  `filePutOK_sum` float unsigned NOT NULL,
  `filePutOK_sumSquare` float unsigned NOT NULL,
  `filePutOK_min` int(10) unsigned NOT NULL,
  `filePutOK_max` int(10) unsigned NOT NULL,
  `fileGetFailures_sum` float unsigned NOT NULL,
  `fileGetFailures_sumSquare` float unsigned NOT NULL,
  `fileGetFailures_min` int(10) unsigned NOT NULL,
  `fileGetFailures_max` int(10) unsigned NOT NULL,
  `filePutFailures_sum` float unsigned NOT NULL,
  `filePutFailures_sumSquare` float unsigned NOT NULL,
  `filePutFailures_min` int(10) unsigned NOT NULL,
  `filePutFailures_max` int(10) unsigned NOT NULL,
  `xferRestarts_sum` float unsigned NOT NULL,
  `xferRestarts_sumSquare` float unsigned NOT NULL,
  `xferRestarts_min` int(10) unsigned NOT NULL,
  `xferRestarts_max` int(10) unsigned NOT NULL,
  `xferAborts_sum` float unsigned NOT NULL,
  `xferAborts_sumSquare` float unsigned NOT NULL,
  `xferAborts_min` int(10) unsigned NOT NULL,
  `xferAborts_max` int(10) unsigned NOT NULL,
  `failedLogins_sum` float unsigned NOT NULL,
  `failedLogins_sumSquare` float unsigned NOT NULL,
  `failedLogins_min` int(10) unsigned NOT NULL,
  `failedLogins_max` int(10) unsigned NOT NULL,
  `cipherFailures_sum` float unsigned NOT NULL,
  `cipherFailures_sumSquare` float unsigned NOT NULL,
  `cipherFailures_min` int(10) unsigned NOT NULL,
  `cipherFailures_max` int(10) unsigned NOT NULL,
  `commandFailures_sum` float unsigned NOT NULL,
  `commandFailures_sumSquare` float unsigned NOT NULL,
  `commandFailures_min` int(10) unsigned NOT NULL,
  `commandFailures_max` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `ftpcttable_template`
--
```

```
DROP TABLE IF EXISTS `ftpcttable_template`;
CREATE TABLE IF NOT EXISTS `ftpcttable_template` (
  `flowID` varchar(200) NOT NULL,
  `ftpTimestamp` datetime NOT NULL,
  `ftpInterval` int(10) unsigned NOT NULL,
  `startedSessions` int(10) unsigned NOT NULL,
  `cleanClosedSession` int(10) unsigned NOT NULL,
  `dirtyClosedSessions` int(10) unsigned NOT NULL,
  `encryptedSessions` int(10) unsigned NOT NULL,
  `noLoginSessions` int(10) unsigned NOT NULL,
  `noCmdSessions` int(10) unsigned NOT NULL,
  `noFileXferSessions` int(10) unsigned NOT NULL,
  `fileXferSessions` int(10) unsigned NOT NULL,
  `activeConnections` int(10) unsigned NOT NULL,
  `passiveConnections` int(10) unsigned NOT NULL,
  `dirListCount` int(10) unsigned NOT NULL,
  `fileGetOK` int(10) unsigned NOT NULL,
  `filePutOK` int(10) unsigned NOT NULL,
  `fileGetFailures` int(10) unsigned NOT NULL,
  `filePutFailures` int(10) unsigned NOT NULL,
  `xferRestarts` int(10) unsigned NOT NULL,
  `xferAborts` int(10) unsigned NOT NULL,
  `failedLogins` int(10) unsigned NOT NULL,
  `cipherFailures` int(10) unsigned NOT NULL,
  `commandFailures` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`ftpTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ftpxfrtable_template`
--

DROP TABLE IF EXISTS `ftpxfrtable_template`;
CREATE TABLE IF NOT EXISTS `ftpxfrtable_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `duration` int(11) unsigned NOT NULL,
  `ctlSessionTime` datetime NOT NULL,
  `clientAddress` varchar(15) NOT NULL,
  `clientDataPort` smallint(5) unsigned NOT NULL,
  `clientControlPort` smallint(5) unsigned NOT NULL,
  `serverAddress` varchar(15) NOT NULL,
  `serverDataPort` smallint(5) unsigned NOT NULL,
  `serverControlPort` smallint(5) unsigned NOT NULL,
  `fileDirection` enum('GET','PUT') DEFAULT NULL,
  `fileName` varchar(300) NOT NULL,
  `workDir` varchar(300) NOT NULL,
  `skippedFileOffset` bigint(20) unsigned NOT NULL,
  `fileSize` bigint(20) unsigned NOT NULL,
  `payloadBytes` bigint(20) unsigned NOT NULL,
  `transfertType` enum('ASCII','BINARY','EBCDIC') DEFAULT NULL,
  `transfertMode` enum('Stream','Block','CompressedBlock') DEFAULT NULL,
  `connectionMode` enum('Active','Passive') DEFAULT NULL,
  `userName` varchar(20) NOT NULL,
  `connectionState` enum('SYN','DATA','FIN','CLOSED') DEFAULT NULL,
```

```
  `probeInterface` varchar(50) NOT NULL,
  `interfaceDescription` varchar(255) NOT NULL,
  PRIMARY KEY
(`flowID`,`firstSeenTime`,`clientAddress`,`clientDataPort`,`serverAddress`,`serv
erDataPort`,`probeInterface`),
  KEY `firstSeenTime` (`firstSeenTime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ftpxfrtable_fix_template`
--

DROP TABLE IF EXISTS `ftpxfrtable_fix_template`;
CREATE TABLE IF NOT EXISTS `ftpxfrtable_fix_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `duration` int(11) unsigned NOT NULL,
  `ctlSessionTime` datetime NOT NULL,
  `clientAddress` varchar(15) NOT NULL,
  `clientDataPort` smallint(5) unsigned NOT NULL,
  `clientControlPort` smallint(5) unsigned NOT NULL,
  `serverAddress` varchar(15) NOT NULL,
  `serverDataPort` smallint(5) unsigned NOT NULL,
  `serverControlPort` smallint(5) unsigned NOT NULL,
  `fileDirection` enum('GET','PUT') DEFAULT NULL,
  `fileName` varchar(300) NOT NULL,
  `workDir` varchar(300) NOT NULL,
  `skippedFileOffset` bigint(20) unsigned NOT NULL,
  `fileSize` bigint(20) unsigned NOT NULL,
  `payloadBytes` bigint(20) unsigned NOT NULL,
  `transfertType` enum('ASCII','BINARY','EBCDIC') DEFAULT NULL,
  `transfertMode` enum('Stream','Block','CompressedBlock') DEFAULT NULL,
  `connectionMode` enum('Active','Passive') DEFAULT NULL,
  `userName` varchar(20) NOT NULL,
  `connectionState` enum('SYN','DATA','FIN','CLOSED') DEFAULT NULL,
  `probeInterface` varchar(50) NOT NULL,
  `interfaceDescription` varchar(255) NOT NULL,
  PRIMARY KEY
(`flowID`,`firstSeenTime`,`clientAddress`,`clientDataPort`,`serverAddress`,`serv
erDataPort`,`probeInterface`),
  KEY `firstSeenTime` (`firstSeenTime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(firstSeenTime)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `ftp_filesxfr_template`
--

DROP TABLE IF EXISTS `ftp_filesxfr_template`;
CREATE TABLE IF NOT EXISTS `ftp_filesxfr_template` (
  `firstSeenTime` datetime NOT NULL,
  `serverAddress` varchar(15) NOT NULL,
```

```
  `location` varchar(100) DEFAULT 'N/A',
  `activity` varchar(100) DEFAULT 'N/A',
  `country` varchar(100) DEFAULT 'N/A',
  `fileName` varchar(300) NOT NULL,
  `fileSize` bigint(20) unsigned NOT NULL DEFAULT '0',
  `nbOfTranfers` bigint(20) NOT NULL DEFAULT '0',
  PRIMARY KEY (`firstSeenTime`,`serverAddress`,`fileName`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(firstSeenTime)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `hopstable_template`
--

DROP TABLE IF EXISTS `hopstable_template`;
CREATE TABLE IF NOT EXISTS `hopstable_template` (
  `classID` int(10) unsigned NOT NULL,
  `className` varchar(30) NOT NULL,
  `classDesc` varchar(100) NOT NULL,
  `timestamp` datetime NOT NULL,
  `cfgVersion` smallint(10) unsigned NOT NULL,
  `cfgStart` datetime NOT NULL,
  `hopCount` tinyint(10) unsigned NOT NULL,
  `hop1Name` varchar(30) DEFAULT NULL,
  `hop2Name` varchar(30) DEFAULT NULL,
  `hop3Name` varchar(30) DEFAULT NULL,
  `hop4Name` varchar(30) DEFAULT NULL,
  `hop5Name` varchar(30) DEFAULT NULL,
  `hop6Name` varchar(30) DEFAULT NULL,
  `hop7Name` varchar(30) DEFAULT NULL,
  `hop8Name` varchar(30) DEFAULT NULL,
  `hop9Name` varchar(30) DEFAULT NULL,
  `hop10Name` varchar(30) DEFAULT NULL,
  PRIMARY KEY (`classID`,`cfgVersion`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `icmpcttable_aggr_template`
--

DROP TABLE IF EXISTS `icmpcttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `icmpcttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
```

```
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `echoRequests_sum` float unsigned NOT NULL,
  `echoRequests_sumSquare` float unsigned NOT NULL,
  `echoRequests_min` bigint(20) unsigned NOT NULL,
  `echoRequests_max` bigint(20) unsigned NOT NULL,
  `echoReplies_sum` float unsigned NOT NULL,
  `echoReplies_sumSquare` float unsigned NOT NULL,
  `echoReplies_min` bigint(20) unsigned NOT NULL,
  `echoReplies_max` bigint(20) unsigned NOT NULL,
  `fragmentationNeeded_sum` float unsigned NOT NULL,
  `fragmentationNeeded_sumSquare` float unsigned NOT NULL,
  `fragmentationNeeded_min` bigint(20) unsigned NOT NULL,
  `fragmentationNeeded_max` bigint(20) unsigned NOT NULL,
  `sourceQuench_sum` float unsigned NOT NULL,
  `sourceQuench_sumSquare` float unsigned NOT NULL,
  `sourceQuench_min` bigint(20) unsigned NOT NULL,
  `sourceQuench_max` bigint(20) unsigned NOT NULL,
  `timeToLiveExpired_sum` float unsigned NOT NULL,
  `timeToLiveExpired_sumSquare` float unsigned NOT NULL,
  `timeToLiveExpired_min` bigint(20) unsigned NOT NULL,
  `timeToLiveExpired_max` bigint(20) unsigned NOT NULL,
  `reassemblyTimeout_sum` float unsigned NOT NULL,
  `reassemblyTimeout_sumSquare` float unsigned NOT NULL,
  `reassemblyTimeout_min` bigint(20) unsigned NOT NULL,
  `reassemblyTimeout_max` bigint(20) unsigned NOT NULL,
  `unReached_sum` float unsigned NOT NULL,
  `unReached_sumSquare` float unsigned NOT NULL,
  `unReached_min` bigint(20) unsigned NOT NULL,
  `unReached_max` bigint(20) unsigned NOT NULL,
  `redirect_sum` float unsigned NOT NULL,
  `redirect_sumSquare` float unsigned NOT NULL,
  `redirect_min` bigint(20) unsigned NOT NULL,
  `redirect_max` bigint(20) unsigned NOT NULL,
  `otherIcmpErrors_sum` float unsigned NOT NULL,
  `otherIcmpErrors_sumSquare` float unsigned NOT NULL,
  `otherIcmpErrors_min` bigint(20) unsigned NOT NULL,
  `otherIcmpErrors_max` bigint(20) unsigned NOT NULL,
  `otherIcmpInfoPackets_sum` float unsigned NOT NULL,
  `otherIcmpInfoPackets_sumSquare` float unsigned NOT NULL,
  `otherIcmpInfoPackets_min` bigint(20) unsigned NOT NULL,
  `otherIcmpInfoPackets_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- ----------------------------------------------------------

--
-- Table structure for table `icmpcttable_template`
--

DROP TABLE IF EXISTS `icmpcttable_template`;
CREATE TABLE IF NOT EXISTS `icmpcttable_template` (
  `flowID` varchar(200) NOT NULL,
  `icmpTimestamp` datetime NOT NULL,
  `icmpInterval` int(10) unsigned NOT NULL,
```

```
  `probeChecksumFailures` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped` bigint(20) unsigned NOT NULL,
  `echoRequests` bigint(20) unsigned NOT NULL,
  `echoReplies` bigint(20) unsigned NOT NULL,
  `fragmentationNeeded` bigint(20) unsigned NOT NULL,
  `sourceQuench` bigint(20) unsigned NOT NULL,
  `timeToLiveExpired` bigint(20) unsigned NOT NULL,
  `reassemblyTimeout` bigint(20) unsigned NOT NULL,
  `unReached` bigint(20) unsigned NOT NULL,
  `redirect` bigint(20) unsigned NOT NULL,
  `otherIcmpErrors` bigint(20) unsigned NOT NULL,
  `otherIcmpInfoPackets` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`icmpTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ------------------------------------------------------

--
-- Table structure for table `ipcttable_aggr_template`
--

DROP TABLE IF EXISTS `ipcttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `ipcttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `totalBytes_sum` float unsigned NOT NULL,
  `totalBytes_sumSquare` float unsigned NOT NULL,
  `totalBytes_min` bigint(20) unsigned NOT NULL,
  `totalBytes_max` bigint(20) unsigned NOT NULL,
  `probeReassemblyTimeout_sum` float unsigned NOT NULL,
  `probeReassemblyTimeout_sumSquare` float unsigned NOT NULL,
  `probeReassemblyTimeout_min` bigint(20) unsigned NOT NULL,
  `probeReassemblyTimeout_max` bigint(20) unsigned NOT NULL,
  `probeFragmentOverlap_sum` float unsigned NOT NULL,
  `probeFragmentOverlap_sumSquare` float unsigned NOT NULL,
  `probeFragmentOverlap_min` bigint(20) unsigned NOT NULL,
  `probeFragmentOverlap_max` bigint(20) unsigned NOT NULL,
  `icmpCount_sum` float unsigned NOT NULL,
  `icmpCount_sumSquare` float unsigned NOT NULL,
  `icmpCount_min` bigint(20) unsigned NOT NULL,
  `icmpCount_max` bigint(20) unsigned NOT NULL,
  `udpCount_sum` float unsigned NOT NULL,
  `udpCount_sumSquare` float unsigned NOT NULL,
  `udpCount_min` bigint(20) unsigned NOT NULL,
  `udpCount_max` bigint(20) unsigned NOT NULL,
  `tcpCount_sum` float unsigned NOT NULL,
  `tcpCount_sumSquare` float unsigned NOT NULL,
  `tcpCount_min` bigint(20) unsigned NOT NULL,
  `tcpCount_max` bigint(20) unsigned NOT NULL,
  `otherProtocolCount_sum` float unsigned NOT NULL,
  `otherProtocolCount_sumSquare` float unsigned NOT NULL,
  `otherProtocolCount_min` bigint(20) unsigned NOT NULL,
  `otherProtocolCount_max` bigint(20) unsigned NOT NULL,
  `bitrate` float unsigned NOT NULL DEFAULT '0',
  `ftpPassiveEstimatedBytes` float unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`flowID`,`rangeStart`),
```

```
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `ipcttable_template`
--

DROP TABLE IF EXISTS `ipcttable_template`;
CREATE TABLE IF NOT EXISTS `ipcttable_template` (
  `flowID` varchar(200) NOT NULL,
  `ipctTimestamp` datetime NOT NULL,
  `ipctInterval` int(10) unsigned NOT NULL,
  `perDatagram` char(1) NOT NULL,
  `totalBytes` float unsigned NOT NULL,
  `sizeBucketCount` int(10) unsigned NOT NULL,
  `probeReassemblyTimeout` bigint(20) unsigned NOT NULL,
  `probeFragmentOverlap` bigint(20) unsigned NOT NULL,
  `icmpCount` bigint(20) unsigned NOT NULL,
  `udpCount` bigint(20) unsigned NOT NULL,
  `tcpCount` bigint(20) unsigned NOT NULL,
  `otherProtocolCount` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`ipctTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ipinfotable_template`
--

DROP TABLE IF EXISTS `ipinfotable_template`;
CREATE TABLE IF NOT EXISTS `ipinfotable_template` (
  `IP` varchar(18) NOT NULL,
  `location` varchar(100) DEFAULT NULL,
  `activity` varchar(100) DEFAULT NULL,
  `country` varchar(100) DEFAULT 'Unknown',
  `city` varchar(100) DEFAULT 'Unknown',
  `ASN` varchar(200) DEFAULT '',
  `DNS` varchar(200) DEFAULT 'Unknown',
  `comments` varchar(200) DEFAULT '',
  PRIMARY KEY (`IP`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ipsztable_aggr_template`
--

DROP TABLE IF EXISTS `ipsztable_aggr_template`;
CREATE TABLE IF NOT EXISTS `ipsztable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
```

```
  `lower` smallint(5) unsigned NOT NULL,
  `upper` smallint(5) unsigned NOT NULL,
  `minimum` smallint(5) unsigned NOT NULL,
  `maximum` smallint(5) unsigned NOT NULL,
  `average` float unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `sum` float unsigned NOT NULL,
  `sumOfSquares` float unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`lower`,`upper`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `ipsztable_template`
--

DROP TABLE IF EXISTS `ipsztable_template`;
CREATE TABLE IF NOT EXISTS `ipsztable_template` (
  `flowID` varchar(200) NOT NULL,
  `ipszTimestamp` datetime NOT NULL,
  `ipszInterval` int(10) unsigned NOT NULL,
  `perDatagram` char(1) NOT NULL,
  `lower` smallint(5) unsigned NOT NULL,
  `upper` smallint(5) unsigned NOT NULL,
  `minimum` smallint(5) unsigned NOT NULL,
  `maximum` smallint(5) unsigned NOT NULL,
  `average` float unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `sum` bigint(20) unsigned NOT NULL,
  `sumOfSquares` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`ipszTimestamp`,`lower`,`upper`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `matchingtable_template`
--

DROP TABLE IF EXISTS `matchingtable_template`;
CREATE TABLE IF NOT EXISTS `matchingtable_template` (
  `collectorFlowID` varchar(200) NOT NULL,
  `flowID` varchar(200) NOT NULL,
  PRIMARY KEY (`collectorFlowID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `metrictable_template`
--

DROP TABLE IF EXISTS `metrictable_template`;
```

```
CREATE TABLE IF NOT EXISTS `metrictable_template` (
  `flowID` varchar(200) NOT NULL,
  `metrictype` varchar(11) NOT NULL,
  `metricSubType` varchar(20) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `lower` int(11) NOT NULL,
  `upper` int(11) NOT NULL,
  PRIMARY KEY (`flowID`,`metrictype`,`metricSubType`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ------------------------------------------------------

--
-- Table structure for table `activityvolumetable_aggr_template`
--

DROP TABLE IF EXISTS `activityvolumetable_aggr_template`;
CREATE TABLE IF NOT EXISTS `activityvolumetable_aggr_template` (
  `rangeStart` datetime NOT NULL,
  `address1` varchar(18) CHARACTER SET latin1 NOT NULL DEFAULT '',
  `address2` varchar(18) CHARACTER SET latin1 NOT NULL DEFAULT '',
  `sPro` varchar(50) COLLATE latin1_general_ci NOT NULL DEFAULT 'No Match',
  `pro` varchar(5) CHARACTER SET latin1 NOT NULL,
  `in_bytes` bigint(20) unsigned DEFAULT '0',
  `out_bytes` bigint(20) unsigned DEFAULT '0',
  `in_bitRate` double unsigned DEFAULT '0',
  `out_bitRate` double unsigned DEFAULT '0',
  `in_ipBytes` bigint(20) unsigned DEFAULT '0',
  `out_ipBytes` bigint(20) unsigned DEFAULT '0',
  `in_payloadBytes` bigint(20) unsigned DEFAULT '0',
  `out_payloadBytes` bigint(20) unsigned DEFAULT '0',
  `in_firstSegmentsPayload` bigint(20) unsigned DEFAULT '0',
  `out_firstSegmentsPayload` bigint(20) unsigned DEFAULT '0',
  `in_retransmittedPayloadBytes` bigint(20) unsigned DEFAULT '0',
  `out_retransmittedPayloadBytes` bigint(20) unsigned DEFAULT '0',
  `in_effectivePayload` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_effectivePayload` decimal(65,4) unsigned DEFAULT '0.0000',
  `protocolEfficiency` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_avgLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_avgLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_avgMaxWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_avgMaxWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_maxLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_maxLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `failedLogins_sum` bigint(20) unsigned DEFAULT '0',
  `noLoginSessions_sum` bigint(20) unsigned DEFAULT '0',
  `noCmdSessions_sum` bigint(20) unsigned DEFAULT '0',
  `noFileXferSessions_sum` bigint(20) unsigned DEFAULT '0',
  `fileXferSessions_sum` bigint(20) unsigned DEFAULT '0',
  `commandFailures_sum` bigint(20) unsigned DEFAULT '0',
  `encryptedSessions_sum` bigint(20) unsigned DEFAULT '0',
  `dirListCount_sum` bigint(20) unsigned DEFAULT '0',
  `fileGetOK_sum` bigint(20) unsigned DEFAULT '0',
  `filePutOK_sum` bigint(20) unsigned DEFAULT '0',
  `fileGetFailures_sum` bigint(20) unsigned DEFAULT '0',
  `filePutFailures_sum` bigint(20) unsigned DEFAULT '0',
  `activity1` varchar(100) COLLATE latin1_general_ci NOT NULL DEFAULT 'N/A',
```

```
  `location1` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `country1` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `activity2` varchar(100) COLLATE latin1_general_ci NOT NULL DEFAULT 'N/A',
  `location2` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `country2` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `dns1` varchar(200) COLLATE latin1_general_ci DEFAULT 'N/A',
  `dns2` varchar(200) COLLATE latin1_general_ci DEFAULT 'N/A',
  PRIMARY KEY (`rangeStart`,`address1`,`address2`,`sPro`,`pro`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `activityvolumetable_netflow_aggr_template`
--

DROP TABLE IF EXISTS `activityvolumetable_netflow_aggr_template`;
CREATE TABLE IF NOT EXISTS `activityvolumetable_netflow_aggr_template` (
  `rangeStart` datetime NOT NULL,
  `address1` varchar(18) CHARACTER SET latin1 NOT NULL DEFAULT '',
  `address2` varchar(18) CHARACTER SET latin1 NOT NULL DEFAULT '',
  `sPro` varchar(50) COLLATE latin1_general_ci NOT NULL DEFAULT 'No Match',
  `pro` varchar(5) CHARACTER SET latin1 NOT NULL,
  `in_bytes` bigint(20) unsigned DEFAULT '0',
  `out_bytes` bigint(20) unsigned DEFAULT '0',
  `in_bitRate` double unsigned DEFAULT '0',
  `out_bitRate` double unsigned DEFAULT '0',
  `in_ipBytes` bigint(20) unsigned DEFAULT '0',
  `out_ipBytes` bigint(20) unsigned DEFAULT '0',
  `in_payloadBytes` bigint(20) unsigned DEFAULT '0',
  `out_payloadBytes` bigint(20) unsigned DEFAULT '0',
  `in_firstSegmentsPayload` bigint(20) unsigned DEFAULT '0',
  `out_firstSegmentsPayload` bigint(20) unsigned DEFAULT '0',
  `in_retransmittedPayloadBytes` bigint(20) unsigned DEFAULT '0',
  `out_retransmittedPayloadBytes` bigint(20) unsigned DEFAULT '0',
  `in_effectivePayload` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_effectivePayload` decimal(65,4) unsigned DEFAULT '0.0000',
  `protocolEfficiency` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_avgLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_avgLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_avgMaxWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_avgMaxWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `in_maxLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `out_maxLastWindow` decimal(65,4) unsigned DEFAULT '0.0000',
  `failedLogins_sum` bigint(20) unsigned DEFAULT '0',
  `noLoginSessions_sum` bigint(20) unsigned DEFAULT '0',
  `noCmdSessions_sum` bigint(20) unsigned DEFAULT '0',
  `noFileXferSessions_sum` bigint(20) unsigned DEFAULT '0',
  `fileXferSessions_sum` bigint(20) unsigned DEFAULT '0',
  `commandFailures_sum` bigint(20) unsigned DEFAULT '0',
  `encryptedSessions_sum` bigint(20) unsigned DEFAULT '0',
  `dirListCount_sum` bigint(20) unsigned DEFAULT '0',
  `fileGetOK_sum` bigint(20) unsigned DEFAULT '0',
  `filePutOK_sum` bigint(20) unsigned DEFAULT '0',
```

```
  `fileGetFailures_sum` bigint(20) unsigned DEFAULT '0',
  `filePutFailures_sum` bigint(20) unsigned DEFAULT '0',
  `activity1` varchar(100) COLLATE latin1_general_ci NOT NULL DEFAULT 'N/A',
  `location1` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `country1` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `activity2` varchar(100) COLLATE latin1_general_ci NOT NULL DEFAULT 'N/A',
  `location2` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `country2` varchar(100) COLLATE latin1_general_ci DEFAULT 'N/A',
  `dns1` varchar(200) COLLATE latin1_general_ci DEFAULT 'N/A',
  `dns2` varchar(200) COLLATE latin1_general_ci DEFAULT 'N/A',
  PRIMARY KEY (`rangeStart`,`address1`,`address2`,`sPro`,`pro`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_general_ci
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `netflowtable_aggr_template`
--

DROP TABLE IF EXISTS `netflowtable_aggr_template`;
CREATE TABLE IF NOT EXISTS `netflowtable_aggr_template` (
  `interface_` int(11) NOT NULL,
  `sIP` varchar(18) NOT NULL,
  `dIP` varchar(18) NOT NULL,
  `pro` varchar(20) NOT NULL,
  `sPort` int(10) NOT NULL,
  `dPort` int(10) NOT NULL,
  `bitRate` float unsigned NOT NULL,
  `packets` int(10) unsigned DEFAULT NULL,
  `bytes` bigint(20) unsigned DEFAULT NULL,
  `rangeStart` datetime NOT NULL,
  PRIMARY KEY (`interface_`,`sIP`,`dIP`,`pro`,`sPort`,`dPort`,`rangeStart`),
  KEY `sPort` (`sPort`,`dPort`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `netflowtable_template`
--

DROP TABLE IF EXISTS `netflowtable_template`;
CREATE TABLE IF NOT EXISTS `netflowtable_template` (
  `idx` int(11) NOT NULL AUTO_INCREMENT,
  `sIP` varchar(18) NOT NULL,
  `dIP` varchar(18) NOT NULL,
  `sPort` smallint(5) unsigned NOT NULL,
  `dPort` smallint(5) unsigned NOT NULL,
  `pro` varchar(20) NOT NULL,
  `packets` int(10) unsigned DEFAULT NULL,
  `bytes` bigint(20) unsigned DEFAULT NULL,
```

```
  `sTime` datetime NOT NULL,
  `eTime` datetime NOT NULL,
  `durat` float unsigned NOT NULL DEFAULT '0',
  `sen` smallint(5) unsigned NOT NULL,
  PRIMARY KEY (`idx`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

-- --------------------------------------------------------

--
-- Table structure for table `onewaycttable_aggr_template`
--

DROP TABLE IF EXISTS `onewaycttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `onewaycttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `lost_sum` float unsigned NOT NULL,
  `lost_sumSquare` float unsigned NOT NULL,
  `lost_min` bigint(20) unsigned NOT NULL,
  `lost_max` bigint(20) unsigned NOT NULL,
  `partlyMissed_sum` float unsigned NOT NULL,
  `partlyMissed_sumSquare` float unsigned NOT NULL,
  `partlyMissed_min` bigint(20) unsigned NOT NULL,
  `partlyMissed_max` bigint(20) unsigned NOT NULL,
  `dropped_sum` float unsigned NOT NULL,
  `dropped_sumSquare` float unsigned NOT NULL,
  `dropped_min` bigint(20) unsigned NOT NULL,
  `dropped_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `onewaycttable_template`
--

DROP TABLE IF EXISTS `onewaycttable_template`;
CREATE TABLE IF NOT EXISTS `onewaycttable_template` (
  `flowID` varchar(200) NOT NULL,
  `oneWayTimestamp` datetime NOT NULL,
  `interval` int(11) NOT NULL,
  `lost` bigint(20) unsigned NOT NULL,
  `partlyMissed` bigint(20) unsigned NOT NULL,
  `dropped` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`oneWayTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `onewaydelaytable_template`
--
```

```
DROP TABLE IF EXISTS `onewaydelaytable_template`;
CREATE TABLE IF NOT EXISTS `onewaydelaytable_template` (
  `flowID` varchar(200) NOT NULL,
  `timestamp` datetime NOT NULL,
  `flowClass` mediumint(8) unsigned NOT NULL,
  `signature` varchar(20) NOT NULL,
  `size` int(10) unsigned NOT NULL,
  `fragmentNumber` int(10) unsigned NOT NULL,
  `hop1FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop1LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10LastMSec` bigint(20) unsigned DEFAULT NULL,
  `packetID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`packetID`,`timestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1
/*!50500 PARTITION BY RANGE  COLUMNS(timestamp)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `onewaylatencytable_aggr_template`
--

DROP TABLE IF EXISTS `onewaylatencytable_aggr_template`;
CREATE TABLE IF NOT EXISTS `onewaylatencytable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `perDatagram` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;
```

```
-- ----------------------------------------------------------

--
-- Table structure for table `onewaylatencytable_template`
--

DROP TABLE IF EXISTS `onewaylatencytable_template`;
CREATE TABLE IF NOT EXISTS `onewaylatencytable_template` (
  `flowID` varchar(200) NOT NULL,
  `timestamp` datetime NOT NULL,
  `interval` int(11) NOT NULL,
  `perDatagram` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`timestamp`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ----------------------------------------------------------

--
-- Table structure for table `onewaylosttable_template`
--

DROP TABLE IF EXISTS `onewaylosttable_template`;
CREATE TABLE IF NOT EXISTS `onewaylosttable_template` (
  `flowID` varchar(200) NOT NULL,
  `timestamp` datetime NOT NULL,
  `flowClass` mediumint(8) unsigned NOT NULL,
  `signature` varchar(20) NOT NULL,
  `size` int(10) unsigned NOT NULL,
  `fragmentNumber` int(10) unsigned NOT NULL,
  `hop1FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop1LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10LastMSec` bigint(20) unsigned DEFAULT NULL,
```

```
  `packetID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`packetID`,`timestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1
/*!50500 PARTITION BY RANGE  COLUMNS(timestamp)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- ------------------------------------------------------

--
-- Table structure for table `onewaymisstable_template`
--

DROP TABLE IF EXISTS `onewaymisstable_template`;
CREATE TABLE IF NOT EXISTS `onewaymisstable_template` (
  `flowID` varchar(200) NOT NULL,
  `timestamp` datetime NOT NULL,
  `flowClass` mediumint(8) unsigned NOT NULL,
  `signature` varchar(20) NOT NULL,
  `size` int(10) unsigned NOT NULL,
  `fragmentNumber` int(10) unsigned NOT NULL,
  `hop1FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop1LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop2LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop3LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop4LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop5LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop6LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop7LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop8LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop9LastMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10FirstMSec` bigint(20) unsigned DEFAULT NULL,
  `hop10LastMSec` bigint(20) unsigned DEFAULT NULL,
  `packetID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`packetID`,`timestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1
/*!50500 PARTITION BY RANGE  COLUMNS(timestamp)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- ------------------------------------------------------

--
-- Table structure for table `tcpcontable_aggr_counters_template`
--

DROP TABLE IF EXISTS `tcpcontable_aggr_counters_template`;
CREATE TABLE IF NOT EXISTS `tcpcontable_aggr_counters_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `ipBytesAB` bigint(20) DEFAULT NULL,
```

```
  `ipBytesBA` bigint(20) DEFAULT NULL,
  `payloadBytesAB` bigint(20) DEFAULT NULL,
  `payloadBytesBA` bigint(20) DEFAULT NULL,
  `firstSegmentsPayloadAB` bigint(20) DEFAULT NULL,
  `firstSegmentsPayloadBA` bigint(20) DEFAULT NULL,
  `retransmittedPayloadBytesAB` bigint(20) DEFAULT NULL,
  `retransmittedPayloadBytesBA` bigint(20) DEFAULT NULL,
  `effectivePayloadAB` float DEFAULT NULL,
  `effectivePayloadBA` float DEFAULT NULL,
  `avgLastWindowAB` float DEFAULT NULL,
  `avgLastWindowBA` float DEFAULT NULL,
  `avgMaxWindowAB` float unsigned NOT NULL DEFAULT '0',
  `avgMaxWindowBA` float unsigned NOT NULL DEFAULT '0',
  `maxLastWindowAB` float DEFAULT NULL,
  `maxLastWindowBA` float DEFAULT NULL,
  PRIMARY KEY (`flowID`,`firstSeenTime`),
  KEY `firstSeenTime` (`firstSeenTime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(firstSeenTime)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `tcpcontable_template`
--

DROP TABLE IF EXISTS `tcpcontable_template`;
CREATE TABLE IF NOT EXISTS `tcpcontable_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `addressA` varchar(15) NOT NULL DEFAULT '',
  `portA` smallint(5) unsigned NOT NULL,
  `addressB` varchar(15) NOT NULL DEFAULT '',
  `portB` smallint(5) unsigned NOT NULL,
  `state` enum('SYN','DATA','FIN','CLOSED') DEFAULT NULL,
  `initiator` enum('A','B') DEFAULT NULL,
  `terminator` enum('A','B') DEFAULT NULL,
  `reset` enum('no','A','B','A+B') DEFAULT NULL,
  `tcpOptions` varchar(255) DEFAULT NULL,
  `probeInterface` varchar(100) NOT NULL DEFAULT '',
  `interfaceDescription` varchar(255) DEFAULT NULL,
  `segmentsAB` int(10) unsigned NOT NULL,
  `ipBytesAB` bigint(20) unsigned NOT NULL,
  `payloadBytesAB` bigint(20) unsigned NOT NULL,
  `firstSegmentsAB` int(10) unsigned NOT NULL,
  `firstSegmentsPayloadAB` bigint(20) unsigned NOT NULL,
  `retransmittedSegmentsAB` int(10) unsigned NOT NULL,
  `retransmittedPayloadBytesAB` int(10) unsigned NOT NULL,
  `emptyAckAB` int(10) unsigned NOT NULL,
  `wouldAckNextAB` int(10) unsigned NOT NULL,
  `firstWindowAB` smallint(5) unsigned NOT NULL,
  `lastWindowAB` int(10) unsigned NOT NULL,
  `maxWindowAB` int(10) unsigned NOT NULL,
  `lastSeenAB` datetime DEFAULT NULL,
  `durationAB` int(10) DEFAULT NULL,
  `segmentsBA` int(10) unsigned NOT NULL,
```

```
  `ipBytesBA` bigint(20) unsigned NOT NULL,
  `payloadBytesBA` bigint(20) unsigned NOT NULL,
  `firstSegmentsBA` int(10) unsigned NOT NULL,
  `firstSegmentsPayloadBA` bigint(20) unsigned NOT NULL,
  `retransmittedSegmentsBA` int(10) unsigned NOT NULL,
  `retransmittedPayloadBytesBA` int(10) unsigned NOT NULL,
  `emptyAckBA` int(10) unsigned NOT NULL,
  `wouldAckNextBA` int(10) unsigned NOT NULL,
  `firstWindowBA` smallint(5) unsigned NOT NULL,
  `lastWindowBA` int(10) unsigned NOT NULL,
  `maxWindowBA` int(10) unsigned NOT NULL,
  `lastSeenBA` datetime DEFAULT NULL,
  `durationBA` int(10) DEFAULT NULL,
  PRIMARY KEY
(`flowID`,`firstSeenTime`,`addressA`,`portA`,`addressB`,`portB`,`probeInterface`
)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ------------------------------------------------------------


--
-- Table structure for table `tcpcontable_fix_template`
--

DROP TABLE IF EXISTS `tcpcontable_fix_template`;
CREATE TABLE IF NOT EXISTS `tcpcontable_fix_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `addressA` varchar(15) NOT NULL DEFAULT '',
  `portA` smallint(5) unsigned NOT NULL,
  `addressB` varchar(15) NOT NULL DEFAULT '',
  `portB` smallint(5) unsigned NOT NULL,
  `state` enum('SYN','DATA','FIN','CLOSED') DEFAULT NULL,
  `initiator` enum('A','B') DEFAULT NULL,
  `terminator` enum('A','B') DEFAULT NULL,
  `reset` enum('no','A','B','A+B') DEFAULT NULL,
  `tcpOptions` varchar(255) DEFAULT NULL,
  `probeInterface` varchar(100) NOT NULL DEFAULT '',
  `interfaceDescription` varchar(255) DEFAULT NULL,
  `segmentsAB` int(10) unsigned NOT NULL,
  `ipBytesAB` bigint(20) unsigned NOT NULL,
  `payloadBytesAB` bigint(20) unsigned NOT NULL,
  `firstSegmentsAB` int(10) unsigned NOT NULL,
  `firstSegmentsPayloadAB` bigint(20) unsigned NOT NULL,
  `retransmittedSegmentsAB` int(10) unsigned NOT NULL,
  `retransmittedPayloadBytesAB` int(10) unsigned NOT NULL,
  `emptyAckAB` int(10) unsigned NOT NULL,
  `wouldAckNextAB` int(10) unsigned NOT NULL,
  `firstWindowAB` smallint(5) unsigned NOT NULL,
  `lastWindowAB` int(10) unsigned NOT NULL,
  `maxWindowAB` int(10) unsigned NOT NULL,
  `lastSeenAB` datetime DEFAULT NULL,
  `durationAB` int(10) DEFAULT NULL,
  `segmentsBA` int(10) unsigned NOT NULL,
  `ipBytesBA` bigint(20) unsigned NOT NULL,
  `payloadBytesBA` bigint(20) unsigned NOT NULL,
  `firstSegmentsBA` int(10) unsigned NOT NULL,
```

```
  `firstSegmentsPayloadBA` bigint(20) unsigned NOT NULL,
  `retransmittedSegmentsBA` int(10) unsigned NOT NULL,
  `retransmittedPayloadBytesBA` int(10) unsigned NOT NULL,
  `emptyAckBA` int(10) unsigned NOT NULL,
  `wouldAckNextBA` int(10) unsigned NOT NULL,
  `firstWindowBA` smallint(5) unsigned NOT NULL,
  `lastWindowBA` int(10) unsigned NOT NULL,
  `maxWindowBA` int(10) unsigned NOT NULL,
  `lastSeenBA` datetime DEFAULT NULL,
  `durationBA` int(10) DEFAULT NULL,
  PRIMARY KEY
(`flowID`,`firstSeenTime`,`addressA`,`portA`,`addressB`,`portB`,`probeInterface`
)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(firstSeenTime)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------

--
-- Table structure for table `tcpcttable_aggr_template`
--

DROP TABLE IF EXISTS `tcpcttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `tcpcttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `retransmits_sum` float unsigned NOT NULL,
  `retransmits_sumSquare` float unsigned NOT NULL,
  `retransmits_min` bigint(20) unsigned NOT NULL,
  `retransmits_max` bigint(20) unsigned NOT NULL,
  `latePackets_sum` float unsigned NOT NULL,
  `latePackets_sumSquare` float unsigned NOT NULL,
  `latePackets_min` int(10) unsigned NOT NULL,
  `latePackets_max` int(10) unsigned NOT NULL,
  `connectionStartCount_sum` float unsigned NOT NULL,
  `connectionStartCount_sumSquare` float unsigned NOT NULL,
  `connectionStartCount_min` int(10) unsigned NOT NULL,
  `connectionStartCount_max` int(10) unsigned NOT NULL,
  `connectionCleanCloseCount_sum` float unsigned NOT NULL,
  `connectionCleanCloseCount_sumSquare` float unsigned NOT NULL,
  `connectionCleanCloseCount_min` int(10) unsigned NOT NULL,
  `connectionCleanCloseCount_max` int(10) unsigned NOT NULL,
  `connectionDirtyCloseCount_sum` float unsigned NOT NULL,
  `connectionDirtyCloseCount_sumSquare` float unsigned NOT NULL,
  `connectionDirtyCloseCount_min` int(10) unsigned NOT NULL,
  `connectionDirtyCloseCount_max` int(10) unsigned NOT NULL,
  `ftpControlConnections_sum` float unsigned NOT NULL,
```

```
  `ftpControlConnections_sumSquare` float unsigned NOT NULL,
  `ftpControlConnections_min` int(10) unsigned NOT NULL,
  `ftpControlConnections_max` int(10) unsigned NOT NULL,
  `ftpFileTransfers_sum` float unsigned NOT NULL,
  `ftpFileTransfers_sumSquare` float unsigned NOT NULL,
  `ftpFileTransfers_min` int(10) unsigned NOT NULL,
  `ftpFileTransfers_max` int(10) unsigned NOT NULL,
  `httpFileTransfers_sum` float unsigned NOT NULL,
  `httpFileTransfers_sumSquare` float unsigned NOT NULL,
  `httpFileTransfers_min` int(10) unsigned NOT NULL,
  `httpFileTransfers_max` int(10) unsigned NOT NULL,
  `otherConnections_sum` float unsigned NOT NULL,
  `otherConnections_sumSquare` float unsigned NOT NULL,
  `otherConnections_min` int(10) unsigned NOT NULL,
  `otherConnections_max` int(10) unsigned NOT NULL,
  `synPackets_sum` float unsigned NOT NULL,
  `synPackets_sumSquare` float unsigned NOT NULL,
  `synPackets_min` int(10) unsigned NOT NULL,
  `synPackets_max` int(10) unsigned NOT NULL,
  `synAckPackets_sum` float unsigned NOT NULL,
  `synAckPackets_sumSquare` float unsigned NOT NULL,
  `synAckPackets_min` int(10) unsigned NOT NULL,
  `synAckPackets_max` int(10) unsigned NOT NULL,
  `finPackets_sum` float unsigned NOT NULL,
  `finPackets_sumSquare` float unsigned NOT NULL,
  `finPackets_min` int(10) unsigned NOT NULL,
  `finPackets_max` int(10) unsigned NOT NULL,
  `resetPackets_sum` float unsigned NOT NULL,
  `resetPackets_sumSquare` float unsigned NOT NULL,
  `resetPackets_min` int(10) unsigned NOT NULL,
  `resetPackets_max` int(10) unsigned NOT NULL,
  `ftpCtlPacket_sum` float unsigned NOT NULL,
  `ftpCtlPacket_sumSquare` float unsigned NOT NULL,
  `ftpCtlPacket_min` int(10) unsigned NOT NULL,
  `ftpCtlPacket_max` int(10) unsigned NOT NULL,
  `ftpFileXferPacket_sum` float unsigned NOT NULL,
  `ftpFileXferPacket_sumSquare` float unsigned NOT NULL,
  `ftpFileXferPacket_min` int(10) unsigned NOT NULL,
  `ftpFileXferPacket_max` int(10) unsigned NOT NULL,
  `ftpDirListPackets_sum` float unsigned NOT NULL,
  `ftpDirListPackets_sumSquare` float unsigned NOT NULL,
  `ftpDirListPackets_min` int(10) unsigned NOT NULL,
  `ftpDirListPackets_max` int(10) unsigned NOT NULL,
  `httpPackets_sum` float unsigned NOT NULL,
  `httpPackets_sumSquare` float unsigned NOT NULL,
  `httpPackets_min` int(10) unsigned NOT NULL,
  `httpPackets_max` int(10) unsigned NOT NULL,
  `otherProtoPackets_sum` float unsigned NOT NULL,
  `otherProtoPackets_sumSquare` float unsigned NOT NULL,
  `otherProtoPackets_min` int(10) unsigned NOT NULL,
  `otherProtoPackets_max` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- ----------------------------------------------------------
```

```
--
-- Table structure for table `tcpcttable_template`
--

DROP TABLE IF EXISTS `tcpcttable_template`;
CREATE TABLE IF NOT EXISTS `tcpcttable_template` (
  `flowID` varchar(200) NOT NULL,
  `tcpTimestamp` datetime NOT NULL,
  `tcpInterval` int(10) unsigned NOT NULL,
  `probeChecksumFailures` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped` bigint(20) unsigned NOT NULL,
  `retransmits` bigint(20) unsigned NOT NULL,
  `latePackets` int(10) unsigned NOT NULL,
  `connectionStartCount` int(10) unsigned NOT NULL,
  `connectionCleanCloseCount` int(10) unsigned NOT NULL,
  `connectionDirtyCloseCount` int(10) unsigned NOT NULL,
  `ftpControlConnections` int(10) unsigned NOT NULL,
  `ftpFileTransfers` int(10) unsigned NOT NULL,
  `httpFileTransfers` int(10) unsigned NOT NULL,
  `otherConnections` int(10) unsigned NOT NULL,
  `synPackets` int(10) unsigned NOT NULL,
  `synAckPackets` int(10) unsigned NOT NULL,
  `finPackets` int(10) unsigned NOT NULL,
  `resetPackets` int(10) unsigned NOT NULL,
  `ftpCtlPacket` int(10) unsigned NOT NULL,
  `ftpFileXferPacket` int(10) unsigned NOT NULL,
  `ftpDirListPackets` int(10) unsigned NOT NULL,
  `httpPackets` int(10) unsigned NOT NULL,
  `otherProtoPackets` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`tcpTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `twowaydelaytable_aggr_template`
--

DROP TABLE IF EXISTS `twowaydelaytable_aggr_template`;
CREATE TABLE IF NOT EXISTS `twowaydelaytable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `withInitiator` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`withInitiator`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- --------------------------------------------------------
```

```
--
-- Table structure for table `twowaydelaytable_template`
--

DROP TABLE IF EXISTS `twowaydelaytable_template`;
CREATE TABLE IF NOT EXISTS `twowaydelaytable_template` (
  `flowID` varchar(200) NOT NULL,
  `timestamp` datetime NOT NULL,
  `interval` int(11) NOT NULL,
  `withInitiator` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`timestamp`,`withInitiator`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `udpcttable_aggr_template`
--

DROP TABLE IF EXISTS `udpcttable_aggr_template`;
CREATE TABLE IF NOT EXISTS `udpcttable_aggr_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `emptyDatagrams_sum` float unsigned NOT NULL,
  `emptyDatagrams_sumSquare` float unsigned NOT NULL,
  `emptyDatagrams_min` bigint(20) unsigned NOT NULL,
  `emptyDatagrams_max` bigint(20) unsigned NOT NULL,
  `snmpCount_sum` float unsigned NOT NULL,
  `snmpCount_sumSquare` float unsigned NOT NULL,
  `snmpCount_min` bigint(20) unsigned NOT NULL,
  `snmpCount_max` bigint(20) unsigned NOT NULL,
  `dnsCount_sum` float unsigned NOT NULL,
  `dnsCount_sumSquare` float unsigned NOT NULL,
  `dnsCount_min` bigint(20) unsigned NOT NULL,
  `dnsCount_max` bigint(20) unsigned NOT NULL,
  `ntpCount_sum` float unsigned NOT NULL,
  `ntpCount_sumSquare` float unsigned NOT NULL,
  `ntpCount_min` bigint(20) unsigned NOT NULL,
  `ntpCount_max` bigint(20) unsigned NOT NULL,
  `otherServiceCount_sum` float unsigned NOT NULL,
```

```
  `otherServiceCount_sumSquare` float unsigned NOT NULL,
  `otherServiceCount_min` bigint(20) unsigned NOT NULL,
  `otherServiceCount_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50500 PARTITION BY RANGE  COLUMNS(rangeStart)
(PARTITION pFuture VALUES LESS THAN MAXVALUE ENGINE = InnoDB) */;

-- ----------------------------------------------------------

--
-- Table structure for table `udpcttable_template`
--

DROP TABLE IF EXISTS `udpcttable_template`;
CREATE TABLE IF NOT EXISTS `udpcttable_template` (
  `flowID` varchar(200) NOT NULL,
  `udpTimestamp` datetime NOT NULL,
  `udpInterval` int(10) unsigned NOT NULL,
  `probeChecksumFailures` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped` bigint(20) unsigned NOT NULL,
  `emptyDatagrams` bigint(20) unsigned NOT NULL,
  `snmpCount` bigint(20) unsigned NOT NULL,
  `dnsCount` bigint(20) unsigned NOT NULL,
  `ntpCount` bigint(20) unsigned NOT NULL,
  `otherServiceCount` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`udpTimestamp`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

3.1.2 Temporary Input Tables Templates

The template structure of the **temporary input tables**, for bulk load, is given below:

```
--
-- Table structure for table `ftpcttable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `ftpcttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `ftpcttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(10) unsigned NOT NULL,
  `startedSessions_sum` float unsigned NOT NULL,
  `startedSessions_sumSquare` float unsigned NOT NULL,
  `startedSessions_min` int(10) unsigned NOT NULL,
  `startedSessions_max` int(10) unsigned NOT NULL,
  `cleanClosedSession_sum` float unsigned NOT NULL,
  `cleanClosedSession_sumSquare` float unsigned NOT NULL,
  `cleanClosedSession_min` int(10) unsigned NOT NULL,
  `cleanClosedSession_max` int(10) unsigned NOT NULL,
  `dirtyClosedSessions_sum` float unsigned NOT NULL,
  `dirtyClosedSessions_sumSquare` float unsigned NOT NULL,
  `dirtyClosedSessions_min` int(10) unsigned NOT NULL,
  `dirtyClosedSessions_max` int(10) unsigned NOT NULL,
  `encryptedSessions_sum` float unsigned NOT NULL,
```

```
  `encryptedSessions_sumSquare` float unsigned NOT NULL,
  `encryptedSessions_min` int(10) unsigned NOT NULL,
  `encryptedSessions_max` int(10) unsigned NOT NULL,
  `noLoginSessions_sum` float unsigned NOT NULL,
  `noLoginSessions_sumSquare` float unsigned NOT NULL,
  `noLoginSessions_min` int(10) unsigned NOT NULL,
  `noLoginSessions_max` int(10) unsigned NOT NULL,
  `noCmdSessions_sum` float unsigned NOT NULL,
  `noCmdSessions_sumSquare` float unsigned NOT NULL,
  `noCmdSessions_min` int(10) unsigned NOT NULL,
  `noCmdSessions_max` int(10) unsigned NOT NULL,
  `noFileXferSessions_sum` float unsigned NOT NULL,
  `noFileXferSessions_sumSquare` float unsigned NOT NULL,
  `noFileXferSessions_min` int(10) unsigned NOT NULL,
  `noFileXferSessions_max` int(10) unsigned NOT NULL,
  `fileXferSessions_sum` float unsigned NOT NULL,
  `fileXferSessions_sumSquare` float unsigned NOT NULL,
  `fileXferSessions_min` int(10) unsigned NOT NULL,
  `fileXferSessions_max` int(10) unsigned NOT NULL,
  `activeConnections_sum` float unsigned NOT NULL,
  `activeConnections_sumSquare` float unsigned NOT NULL,
  `activeConnections_min` int(10) unsigned NOT NULL,
  `activeConnections_max` int(10) unsigned NOT NULL,
  `passiveConnections_sum` float unsigned NOT NULL,
  `passiveConnections_sumSquare` float unsigned NOT NULL,
  `passiveConnections_min` int(10) unsigned NOT NULL,
  `passiveConnections_max` int(10) unsigned NOT NULL,
  `dirListCount_sum` float unsigned NOT NULL,
  `dirListCount_sumSquare` float unsigned NOT NULL,
  `dirListCount_min` int(10) unsigned NOT NULL,
  `dirListCount_max` int(10) unsigned NOT NULL,
  `fileGetOK_sum` float unsigned NOT NULL,
  `fileGetOK_sumSquare` float unsigned NOT NULL,
  `fileGetOK_min` int(10) unsigned NOT NULL,
  `fileGetOK_max` int(10) unsigned NOT NULL,
  `filePutOK_sum` float unsigned NOT NULL,
  `filePutOK_sumSquare` float unsigned NOT NULL,
  `filePutOK_min` int(10) unsigned NOT NULL,
  `filePutOK_max` int(10) unsigned NOT NULL,
  `fileGetFailures_sum` float unsigned NOT NULL,
  `fileGetFailures_sumSquare` float unsigned NOT NULL,
  `fileGetFailures_min` int(10) unsigned NOT NULL,
  `fileGetFailures_max` int(10) unsigned NOT NULL,
  `filePutFailures_sum` float unsigned NOT NULL,
  `filePutFailures_sumSquare` float unsigned NOT NULL,
  `filePutFailures_min` int(10) unsigned NOT NULL,
  `filePutFailures_max` int(10) unsigned NOT NULL,
  `xferRestarts_sum` float unsigned NOT NULL,
  `xferRestarts_sumSquare` float unsigned NOT NULL,
  `xferRestarts_min` int(10) unsigned NOT NULL,
  `xferRestarts_max` int(10) unsigned NOT NULL,
  `xferAborts_sum` float unsigned NOT NULL,
  `xferAborts_sumSquare` float unsigned NOT NULL,
  `xferAborts_min` int(10) unsigned NOT NULL,
  `xferAborts_max` int(10) unsigned NOT NULL,
  `failedLogins_sum` float unsigned NOT NULL,
  `failedLogins_sumSquare` float unsigned NOT NULL,
```

```
  `failedLogins_min` int(10) unsigned NOT NULL,
  `failedLogins_max` int(10) unsigned NOT NULL,
  `cipherFailures_sum` float unsigned NOT NULL,
  `cipherFailures_sumSquare` float unsigned NOT NULL,
  `cipherFailures_min` int(10) unsigned NOT NULL,
  `cipherFailures_max` int(10) unsigned NOT NULL,
  `commandFailures_sum` float unsigned NOT NULL,
  `commandFailures_sumSquare` float unsigned NOT NULL,
  `commandFailures_min` int(10) unsigned NOT NULL,
  `commandFailures_max` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ------------------------------------------------------

--
-- Table structure for table `icmpcttable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `icmpcttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `icmpcttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `echoRequests_sum` float unsigned NOT NULL,
  `echoRequests_sumSquare` float unsigned NOT NULL,
  `echoRequests_min` bigint(20) unsigned NOT NULL,
  `echoRequests_max` bigint(20) unsigned NOT NULL,
  `echoReplies_sum` float unsigned NOT NULL,
  `echoReplies_sumSquare` float unsigned NOT NULL,
  `echoReplies_min` bigint(20) unsigned NOT NULL,
  `echoReplies_max` bigint(20) unsigned NOT NULL,
  `fragmentationNeeded_sum` float unsigned NOT NULL,
  `fragmentationNeeded_sumSquare` float unsigned NOT NULL,
  `fragmentationNeeded_min` bigint(20) unsigned NOT NULL,
  `fragmentationNeeded_max` bigint(20) unsigned NOT NULL,
  `sourceQuench_sum` float unsigned NOT NULL,
  `sourceQuench_sumSquare` float unsigned NOT NULL,
  `sourceQuench_min` bigint(20) unsigned NOT NULL,
  `sourceQuench_max` bigint(20) unsigned NOT NULL,
  `timeToLiveExpired_sum` float unsigned NOT NULL,
  `timeToLiveExpired_sumSquare` float unsigned NOT NULL,
  `timeToLiveExpired_min` bigint(20) unsigned NOT NULL,
  `timeToLiveExpired_max` bigint(20) unsigned NOT NULL,
  `reassemblyTimeout_sum` float unsigned NOT NULL,
  `reassemblyTimeout_sumSquare` float unsigned NOT NULL,
  `reassemblyTimeout_min` bigint(20) unsigned NOT NULL,
  `reassemblyTimeout_max` bigint(20) unsigned NOT NULL,
```

```
  `unReached_sum` float unsigned NOT NULL,
  `unReached_sumSquare` float unsigned NOT NULL,
  `unReached_min` bigint(20) unsigned NOT NULL,
  `unReached_max` bigint(20) unsigned NOT NULL,
  `redirect_sum` float unsigned NOT NULL,
  `redirect_sumSquare` float unsigned NOT NULL,
  `redirect_min` bigint(20) unsigned NOT NULL,
  `redirect_max` bigint(20) unsigned NOT NULL,
  `otherIcmpErrors_sum` float unsigned NOT NULL,
  `otherIcmpErrors_sumSquare` float unsigned NOT NULL,
  `otherIcmpErrors_min` bigint(20) unsigned NOT NULL,
  `otherIcmpErrors_max` bigint(20) unsigned NOT NULL,
  `otherIcmpInfoPackets_sum` float unsigned NOT NULL,
  `otherIcmpInfoPackets_sumSquare` float unsigned NOT NULL,
  `otherIcmpInfoPackets_min` bigint(20) unsigned NOT NULL,
  `otherIcmpInfoPackets_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ipcttable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `ipcttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `ipcttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `totalBytes_sum` float unsigned NOT NULL,
  `totalBytes_sumSquare` float unsigned NOT NULL,
  `totalBytes_min` bigint(20) unsigned NOT NULL,
  `totalBytes_max` bigint(20) unsigned NOT NULL,
  `probeReassemblyTimeout_sum` float unsigned NOT NULL,
  `probeReassemblyTimeout_sumSquare` float unsigned NOT NULL,
  `probeReassemblyTimeout_min` bigint(20) unsigned NOT NULL,
  `probeReassemblyTimeout_max` bigint(20) unsigned NOT NULL,
  `probeFragmentOverlap_sum` float unsigned NOT NULL,
  `probeFragmentOverlap_sumSquare` float unsigned NOT NULL,
  `probeFragmentOverlap_min` bigint(20) unsigned NOT NULL,
  `probeFragmentOverlap_max` bigint(20) unsigned NOT NULL,
  `icmpCount_sum` float unsigned NOT NULL,
  `icmpCount_sumSquare` float unsigned NOT NULL,
  `icmpCount_min` bigint(20) unsigned NOT NULL,
  `icmpCount_max` bigint(20) unsigned NOT NULL,
  `udpCount_sum` float unsigned NOT NULL,
  `udpCount_sumSquare` float unsigned NOT NULL,
  `udpCount_min` bigint(20) unsigned NOT NULL,
  `udpCount_max` bigint(20) unsigned NOT NULL,
  `tcpCount_sum` float unsigned NOT NULL,
  `tcpCount_sumSquare` float unsigned NOT NULL,
  `tcpCount_min` bigint(20) unsigned NOT NULL,
  `tcpCount_max` bigint(20) unsigned NOT NULL,
  `otherProtocolCount_sum` float unsigned NOT NULL,
  `otherProtocolCount_sumSquare` float unsigned NOT NULL,
  `otherProtocolCount_min` bigint(20) unsigned NOT NULL,
```

```
  `otherProtocolCount_max` bigint(20) unsigned NOT NULL,
  `bitrate` float unsigned NOT NULL DEFAULT '0',
  `ftpPassiveEstimatedBytes` float unsigned NOT NULL DEFAULT '0',
  PRIMARY KEY (`flowID`,`rangeStart`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `ipsztable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `ipsztable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `ipsztable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `lower` smallint(5) unsigned NOT NULL,
  `upper` smallint(5) unsigned NOT NULL,
  `minimum` smallint(5) unsigned NOT NULL,
  `maximum` smallint(5) unsigned NOT NULL,
  `average` float unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `sum` float unsigned NOT NULL,
  `sumOfSquares` float unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`lower`,`upper`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `netflowtable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `netflowtable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `netflowtable_aggr_tmp_template` (
  `interface_` int(11) NOT NULL,
  `sIP` varchar(18) NOT NULL,
  `dIP` varchar(18) NOT NULL,
  `pro` varchar(20) NOT NULL,
  `sPort` int(10) NOT NULL,
  `dPort` int(10) NOT NULL,
  `bitRate` float unsigned NOT NULL,
  `packets` int(10) unsigned DEFAULT NULL,
  `bytes` bigint(20) unsigned DEFAULT NULL,
  `rangeStart` datetime NOT NULL,
  PRIMARY KEY (`interface_`,`sIP`,`dIP`,`pro`,`sPort`,`dPort`,`rangeStart`),
  KEY `sPort` (`sPort`,`dPort`),
  KEY `rangeStart` (`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `onewaycttable_aggr_tmp_template`
--
```

```
DROP TABLE IF EXISTS `onewaycttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `onewaycttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `lost_sum` float unsigned NOT NULL,
  `lost_sumSquare` float unsigned NOT NULL,
  `lost_min` bigint(20) unsigned NOT NULL,
  `lost_max` bigint(20) unsigned NOT NULL,
  `partlyMissed_sum` float unsigned NOT NULL,
  `partlyMissed_sumSquare` float unsigned NOT NULL,
  `partlyMissed_min` bigint(20) unsigned NOT NULL,
  `partlyMissed_max` bigint(20) unsigned NOT NULL,
  `dropped_sum` float unsigned NOT NULL,
  `dropped_sumSquare` float unsigned NOT NULL,
  `dropped_min` bigint(20) unsigned NOT NULL,
  `dropped_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `onewaylatencytable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `onewaylatencytable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `onewaylatencytable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `perDatagram` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `tcpcontable_aggr_counters_tmp_template`
--

DROP TABLE IF EXISTS `tcpcontable_aggr_counters_tmp_template`;
CREATE TABLE IF NOT EXISTS `tcpcontable_aggr_counters_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `firstSeenTime` datetime NOT NULL,
  `ipBytesAB` bigint(20) DEFAULT NULL,
  `ipBytesBA` bigint(20) DEFAULT NULL,
  `payloadBytesAB` bigint(20) DEFAULT NULL,
  `payloadBytesBA` bigint(20) DEFAULT NULL,
  `firstSegmentsPayloadAB` bigint(20) DEFAULT NULL,
```

```
  `firstSegmentsPayloadBA` bigint(20) DEFAULT NULL,
  `retransmittedPayloadBytesAB` bigint(20) DEFAULT NULL,
  `retransmittedPayloadBytesBA` bigint(20) DEFAULT NULL,
  `effectivePayloadAB` float DEFAULT NULL,
  `effectivePayloadBA` float DEFAULT NULL,
  `avgLastWindowAB` float DEFAULT NULL,
  `avgLastWindowBA` float DEFAULT NULL,
  `avgMaxWindowAB` float unsigned NOT NULL DEFAULT '0',
  `avgMaxWindowBA` float unsigned NOT NULL DEFAULT '0',
  `maxLastWindowAB` float DEFAULT NULL,
  `maxLastWindowBA` float DEFAULT NULL,
  PRIMARY KEY (`flowID`,`firstSeenTime`),
  KEY `firstSeenTime` (`firstSeenTime`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- ------------------------------------------------------

--
-- Table structure for table `udpcttable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `udpcttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `udpcttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `emptyDatagrams_sum` float unsigned NOT NULL,
  `emptyDatagrams_sumSquare` float unsigned NOT NULL,
  `emptyDatagrams_min` bigint(20) unsigned NOT NULL,
  `emptyDatagrams_max` bigint(20) unsigned NOT NULL,
  `snmpCount_sum` float unsigned NOT NULL,
  `snmpCount_sumSquare` float unsigned NOT NULL,
  `snmpCount_min` bigint(20) unsigned NOT NULL,
  `snmpCount_max` bigint(20) unsigned NOT NULL,
  `dnsCount_sum` float unsigned NOT NULL,
  `dnsCount_sumSquare` float unsigned NOT NULL,
  `dnsCount_min` bigint(20) unsigned NOT NULL,
  `dnsCount_max` bigint(20) unsigned NOT NULL,
  `ntpCount_sum` float unsigned NOT NULL,
  `ntpCount_sumSquare` float unsigned NOT NULL,
  `ntpCount_min` bigint(20) unsigned NOT NULL,
  `ntpCount_max` bigint(20) unsigned NOT NULL,
  `otherServiceCount_sum` float unsigned NOT NULL,
  `otherServiceCount_sumSquare` float unsigned NOT NULL,
  `otherServiceCount_min` bigint(20) unsigned NOT NULL,
  `otherServiceCount_max` bigint(20) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
-- --------------------------------------------------------

--
-- Table structure for table `twowaydelaytable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `twowaydelaytable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `twowaydelaytable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `withInitiator` char(1) NOT NULL,
  `sliceNum` tinyint(3) unsigned NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `minimum` int(10) NOT NULL,
  `maximum` int(10) NOT NULL,
  `average` float NOT NULL,
  `sum` float NOT NULL,
  `sumOfSquares` float NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`,`withInitiator`,`sliceNum`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- --------------------------------------------------------

--
-- Table structure for table `tcpcttable_aggr_tmp_template`
--

DROP TABLE IF EXISTS `tcpcttable_aggr_tmp_template`;
CREATE TABLE IF NOT EXISTS `tcpcttable_aggr_tmp_template` (
  `flowID` varchar(200) NOT NULL,
  `rangeStart` datetime NOT NULL,
  `population` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_sum` float unsigned NOT NULL,
  `probeChecksumFailures_sumSquare` float unsigned NOT NULL,
  `probeChecksumFailures_min` bigint(20) unsigned NOT NULL,
  `probeChecksumFailures_max` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_sum` float unsigned NOT NULL,
  `probeChecksumSkipped_sumSquare` float unsigned NOT NULL,
  `probeChecksumSkipped_min` bigint(20) unsigned NOT NULL,
  `probeChecksumSkipped_max` bigint(20) unsigned NOT NULL,
  `retransmits_sum` float unsigned NOT NULL,
  `retransmits_sumSquare` float unsigned NOT NULL,
  `retransmits_min` bigint(20) unsigned NOT NULL,
  `retransmits_max` bigint(20) unsigned NOT NULL,
  `latePackets_sum` float unsigned NOT NULL,
  `latePackets_sumSquare` float unsigned NOT NULL,
  `latePackets_min` int(10) unsigned NOT NULL,
  `latePackets_max` int(10) unsigned NOT NULL,
  `connectionStartCount_sum` float unsigned NOT NULL,
  `connectionStartCount_sumSquare` float unsigned NOT NULL,
  `connectionStartCount_min` int(10) unsigned NOT NULL,
  `connectionStartCount_max` int(10) unsigned NOT NULL,
  `connectionCleanCloseCount_sum` float unsigned NOT NULL,
  `connectionCleanCloseCount_sumSquare` float unsigned NOT NULL,
  `connectionCleanCloseCount_min` int(10) unsigned NOT NULL,
  `connectionCleanCloseCount_max` int(10) unsigned NOT NULL,
  `connectionDirtyCloseCount_sum` float unsigned NOT NULL,
```

```
  `connectionDirtyCloseCount_sumSquare` float unsigned NOT NULL,
  `connectionDirtyCloseCount_min` int(10) unsigned NOT NULL,
  `connectionDirtyCloseCount_max` int(10) unsigned NOT NULL,
  `ftpControlConnections_sum` float unsigned NOT NULL,
  `ftpControlConnections_sumSquare` float unsigned NOT NULL,
  `ftpControlConnections_min` int(10) unsigned NOT NULL,
  `ftpControlConnections_max` int(10) unsigned NOT NULL,
  `ftpFileTransfers_sum` float unsigned NOT NULL,
  `ftpFileTransfers_sumSquare` float unsigned NOT NULL,
  `ftpFileTransfers_min` int(10) unsigned NOT NULL,
  `ftpFileTransfers_max` int(10) unsigned NOT NULL,
  `httpFileTransfers_sum` float unsigned NOT NULL,
  `httpFileTransfers_sumSquare` float unsigned NOT NULL,
  `httpFileTransfers_min` int(10) unsigned NOT NULL,
  `httpFileTransfers_max` int(10) unsigned NOT NULL,
  `otherConnections_sum` float unsigned NOT NULL,
  `otherConnections_sumSquare` float unsigned NOT NULL,
  `otherConnections_min` int(10) unsigned NOT NULL,
  `otherConnections_max` int(10) unsigned NOT NULL,
  `synPackets_sum` float unsigned NOT NULL,
  `synPackets_sumSquare` float unsigned NOT NULL,
  `synPackets_min` int(10) unsigned NOT NULL,
  `synPackets_max` int(10) unsigned NOT NULL,
  `synAckPackets_sum` float unsigned NOT NULL,
  `synAckPackets_sumSquare` float unsigned NOT NULL,
  `synAckPackets_min` int(10) unsigned NOT NULL,
  `synAckPackets_max` int(10) unsigned NOT NULL,
  `finPackets_sum` float unsigned NOT NULL,
  `finPackets_sumSquare` float unsigned NOT NULL,
  `finPackets_min` int(10) unsigned NOT NULL,
  `finPackets_max` int(10) unsigned NOT NULL,
  `resetPackets_sum` float unsigned NOT NULL,
  `resetPackets_sumSquare` float unsigned NOT NULL,
  `resetPackets_min` int(10) unsigned NOT NULL,
  `resetPackets_max` int(10) unsigned NOT NULL,
  `ftpCtlPacket_sum` float unsigned NOT NULL,
  `ftpCtlPacket_sumSquare` float unsigned NOT NULL,
  `ftpCtlPacket_min` int(10) unsigned NOT NULL,
  `ftpCtlPacket_max` int(10) unsigned NOT NULL,
  `ftpFileXferPacket_sum` float unsigned NOT NULL,
  `ftpFileXferPacket_sumSquare` float unsigned NOT NULL,
  `ftpFileXferPacket_min` int(10) unsigned NOT NULL,
  `ftpFileXferPacket_max` int(10) unsigned NOT NULL,
  `ftpDirListPackets_sum` float unsigned NOT NULL,
  `ftpDirListPackets_sumSquare` float unsigned NOT NULL,
  `ftpDirListPackets_min` int(10) unsigned NOT NULL,
  `ftpDirListPackets_max` int(10) unsigned NOT NULL,
  `httpPackets_sum` float unsigned NOT NULL,
  `httpPackets_sumSquare` float unsigned NOT NULL,
  `httpPackets_min` int(10) unsigned NOT NULL,
  `httpPackets_max` int(10) unsigned NOT NULL,
  `otherProtoPackets_sum` float unsigned NOT NULL,
  `otherProtoPackets_sumSquare` float unsigned NOT NULL,
  `otherProtoPackets_min` int(10) unsigned NOT NULL,
  `otherProtoPackets_max` int(10) unsigned NOT NULL,
  PRIMARY KEY (`flowID`,`rangeStart`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
-- ------------------------------------------------------

--
-- Table structure for table `flowtable_template`
--
```

3.2 DATABASE STORED PROCEDURES

3.2.1 Protocol Details Aggregates Update

Several SQL stored procedures are created to implement routines invoked by the loading and aggregating script (see section 3.4 below).

Following routines, defined in the **`trafMon_template`** database, permit to aggregate just loaded raw input data to update/complete the detailed per minute aggregate as well as the further aggregates per hour and per day:

```
CREATE PROCEDURE `Aggr_ftpcttable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_ftpcttable_next_level`(IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_icmpcttable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_icmpcttable_next_level`(IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_ipcttable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_ipcttable_next_level`(IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_ipsztable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_ipsztable_next_level`(IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_onewaycttable _first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_onewaycttable _next_level` (IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_onewaylatencytable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_onewaylatencytable_next_level` (IN `_dbName` VARCHAR(64),
            IN `_srcTableSuffix` VARCHAR(25), IN `_dstTableSuffix` VARCHAR(25),
                                              IN `_aggrInterval` INT)
CREATE PROCEDURE `Aggr_tcpcontable_counters_first_level`(
                                                    IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_tcpcontable_counters_next_level`(
                      IN `_dbName` VARCHAR(64),IN `srcTableSuffix` VARCHAR(25),
                      IN `dstTableSuffix` VARCHAR(25), IN `aggrInterval` INT)
CREATE PROCEDURE `Aggr_tcpcttable_first_level`(IN `_dbName` VARCHAR(64))
```

```
CREATE PROCEDURE `Aggr_tcpcttable_next_level`(IN `_dbName` VARCHAR(64),
                        IN `_dbName` VARCHAR(64),IN `srcTableSuffix` VARCHAR(25),
                        IN `dstTableSuffix` VARCHAR(25), IN `aggrInterval` INT)
CREATE PROCEDURE `Aggr_twowaydelaytable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_twowaydelaytable_next_level`(IN `_dbName` VARCHAR(64),
                        IN `_dbName` VARCHAR(64),IN `srcTableSuffix` VARCHAR(25),
                        IN `dstTableSuffix` VARCHAR(25), IN `aggrInterval` INT)
CREATE PROCEDURE `Aggr_udpcttable_first_level`(IN `_dbName` VARCHAR(64))
CREATE PROCEDURE `Aggr_udpcttable_next_level`(IN `_dbName` VARCHAR(64),
                        IN `_dbName` VARCHAR(64),IN `srcTableSuffix` VARCHAR(25),
                        IN `dstTableSuffix` VARCHAR(25), IN `aggrInterval` INT)
```

PROCESS:

Those primary aggregation routines **`Aggr_xxxtable_first_level()`** perform a first aggregation pass, at one minute granularity, from temporary table *`xxx`*`table_tmp` into temporary aggregate table *`xxx`*`table_aggr_1m_tmp`.

It then **UPDATE** the records in permanent table *`xxx`*`table_aggr_1m` that `INNER JOIN`s those in temporary aggregate table *`xxx`*`table_aggr_1m_tmp` (same `FlowID/rangeStart`).

After that, it **INSERT** the other records from temporary aggregate table *`xxx`*`table_aggr_1m_tmp` which do not exist yet (LEFT JOIN gives NULL for corresponding `FlowID/rangeStart`), into permanent table *`xxx`*`table_aggr_1m`.

Finally further aggregation is obtained (the same way) via

- `CALL Aggr_ftpcttable_next_level('1m','1h',3600)`, for permanent table `ftpcttable_aggr_1h`
- `CALL Aggr_ftpcttable_next_level('1h','1d',3600)`, for permanent table `ftpcttable_aggr_1d`

Note that for the optional NetFlow data, the process is slightly different:

> As all the data about 1 hour (or successive hours) are systematically (re-)extracted from the SiLK log files. Newly reported NetFlow records could provide flow volumes over minute slots in the past that were not yet present, while the prior loaded data are expected to be present again in the fresh data.

Hence the `PROCEDURE` **`Aggr_netflowtable_first_level`**`(IN `_dbName`` `VARCHAR(64))` performs a `REPLACE INTO netflowtable_aggr_1m`, overwriting pre-aggregated data for the target 1 minute time slots.

And the next level `PROCEDURE` **`Aggr_netflowtable_next_level`**`(IN `_dbName`` `VARCHAR(64), IN `_dstTableSuffix` VARCHAR(25), IN `_aggrInterval` INT)` aggregates directly from the 1 minute TEMPORARY TABLE `netflowtable_tmp_aggr`. Hence the lack of `srcTableSuffix` argument.

- Get_update_ip_info_address

```
CREATE PROCEDURE `Get_update_ip_info_Addresses`(IN `_dbName` VARCHAR(64))
```

This procedure retrieves the newly found IP addresses in order for the `trafMon_updateIpInfo.py` script to query the DNS server and update only the new addresses. It call was first migrated to `trafMon_loader.py`, to avoid locking and table access timeout; then it has been replaced there by a more efficient code, focusing only at the freshly loaded IP addresses.

3.2.2 Partitioning Process

During the aggregation process, a partition for the newly inserted data is also created if needed. The period that a partition covers depends on the granularity of the table: **1m** aggregation tables have partitions for **each day** of data, **1h** tables have partitions for each **8 days** of data and **1d** tables for each **31 days** of data.

Each kind of aggregation table (1m, 1h and 1d) has its own stored procedure to create a new partition: partition_create is used for the 1m tables, partition_create_1h for the 1h tables and partition_create_1d for the 1d tables:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `partition_create`(`SCHEMANAME` VARC
HAR(64), `TABLENAME` VARCHAR(64), `REQUESTED_DATE` INT)
CREATE DEFINER=`root`@`localhost` PROCEDURE `partition_create_1h`(`SCHEMANAME` V
ARCHAR(64), `TABLENAME` VARCHAR(64), `REQUESTED_DATE` INT)
CREATE DEFINER=`root`@`localhost` PROCEDURE `partition_create_1d`(`SCHEMANAME` V
ARCHAR(64), `TABLENAME` VARCHAR(64), `REQUESTED_DATE` INT)
```

A stored procedure manages database clean-up by dropping partitions older than a given number of days, using the following procedures:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `partition_drop`(`SCHEMANAME` VARCHA
R(64), `TABLENAME` VARCHAR(64), `NB_DAYS` INT)
```

3.2.3 Data Preparation Procedures

- Ipct_tcp_no_match_update

```
CREATE PROCEDURE `Ipct_tcp_no_match_update`(IN `_dbName` VARCHAR(64))
```

This procedure is now replaced by Update_ftp_data_in_ipcttable()

Maps volume of actual FTP data transfer connections to the flow corresponding to the FTP control connections

- Update_ftp_data_in_ipcttable

```
CREATE PROCEDURE `update_ftp_data_in_ipcttable`( IN `_dbName` VARCHAR(64),
                        `_rangeStart` VARCHAR(100), `_rangeEnd` VARCHAR(100) )
```

Invoked by crontab, once per day just before Aggr_activityvolumetable_first_level(), to count Passive data connections as part of their FTP flows.

Sum-up the volume of FTP passive transfers as part of the corresponding FTP flow identified by its control connection. This procedure is called to update a column, ftpPassiveEstimatedBytes, in the ipcttable_aggr tables. This column contains, for each flow, the quantity of bytes which belongs to FTP passive transfers.

This information is complex to obtain while generating a report, so it is calculated once per day by this procedure and inserted in the corresponding tables to make sure the correct amount of FTP bytes are reported in a fast and easy way.

- Aggr_ipcttable_activity

```
CREATE PROCEDURE `Aggr_ipcttable_activity`(IN `_dbName` VARCHAR(64),
                    IN `_tableSuffix` VARCHAR(25), _requestedDate DATE)
```

Replaced by Update_activity_aggr_volume_data()

- Update_activity_aggr_volume_data

```
CREATE PROCEDURE `Update_activity_aggr_volume_data`(IN `_dbName` VARCHAR(64),
                IN `_TableName` VARCHAR(100), IN `_rangeStart` VARCHAR(100),
                IN `_rangeEnd` VARCHAR(100), IN `_GUID` VARCHAR(100),
                IN `_activity` VARCHAR(100), IN `_location` VARCHAR(100),
                IN `_ip` VARCHAR(100))
```

Utility routine called by Aggr_activityvolumetable_first_level() and by Aggr_activityvolumetable_first_level() to prepare yesterday data for the synthesis reporting (on Activity/Location/peer Country traffic volumes)

Extract the IP volumes and TCP and FTP meaningful counters as well as the qualified source and destination addresses (activity, location, country, DNS)

- Aggr_activityvolumetable_first_level

```
CREATE PROCEDURE `Aggr_activityvolumetable_first_level`(
                IN `_dbName` VARCHAR(64), IN `_rangeStart` VARCHAR(100),
                                          IN `_rangeEnd` VARCHAR(100))
```

For the given range of days (up to yesterday), or for yesterday when both `rangeStart` and `rangeEnd` are NULL, call the Update_activity_aggr_volume_data() based on the ipcttable_aggr_1h (flow volumes detected by the probes) to produce all the base data necessary for the synthesis reports at 1 hour granularity: into activityvolumetable_aggr_1h.

This table is the source of synthesis reports covering one day.

Then calls Aggr_activityvolumetable_next_level() (see below) to aggregate those hourly data per day (adding or replacing the values for the specified range of days).

This procedure must be invoked explicitly (e.g. via `crontab`), typically once a day during quiet time at night, with `rangeStart` = `rangeEnd` = NULL (case insensitive, but without quotes).

- Aggr_activityvolumetable_next_level

```
CREATE PROCEDURE `Aggr_activityvolumetable_next_level`(IN `_dbName` VARCHAR(64),
                IN `_rangeStart` VARCHAR(100), IN `_rangeEnd` VARCHAR(100))
```

Use activityvolumetable_aggr_1h, supposedly updated by invoking Aggr_activityvolumetable_first_level() procedure, to aggregate (and possibly replace) the daily values in activityvolumetable_aggr_1d.

This table is the source of synthesis reports covering one day.

- Aggr_activityvolumetable_netflow_first_level

```
CREATE PROCEDURE `Aggr_activityvolumetable_netflow_first_level`(
                 IN `_dbName` VARCHAR(64), IN `_rangeStart` VARCHAR(100),
                                           IN `_rangeEnd` VARCHAR(100))
```

For the given range of days (up to yesterday), or for yesterday when both `rangeStart` and `rangeEnd` are NULL, call the Update_activity_aggr_volume_data() based on the netflow_aggr_1h (flow volumes provided by SiLK logs) to produce all the base data necessary for the synthesis reports at 1 hour granularity: into activityvolumetable_netflow_aggr_1h.

This table is the source of synthesis reports covering one day.

Then calls Aggr_activityvolumetable_netflow_next_level() (see below) to aggregate those hourly data per day (adding or replacing the values for the specified range of days).

This procedure must be invoked explicitly (e.g. via `crontab`), typically once a day during quiet time at night (dissociated from the execution of Aggr_activityvolumetable_first_level()), with `rangeStart` = `rangeEnd` = NULL (case insensitive, but without quotes).

- Aggr_activityvolumetable_netflow_next_level

```
CREATE PROCEDURE `Aggr_activityvolumetable_netflow_next_level`(
                 IN `_dbName` VARCHAR(64),  IN `_rangeStart` VARCHAR(100),
                 IN `_rangeEnd` VARCHAR(100))
```

Use activityvolumetable_netflow_aggr_1h, supposedly updated by invoking Aggr_activityvolumetable_netflow_first_level() procedure, to aggregate (and possibly replace) the daily values in activityvolumetable_netflow_aggr_1d.

This table is the source of synthesis reports covering one day.

3.2.4 Additional Stored Procedures

- Init_tables_if_necessary

```
CREATE PROCEDURE `init_tables_if_necessary`(`_dbName` VARCHAR(64),
                                            `tablePrefix` VARCHAR(64) )
```

This procedure is executed once at the beginning of nearly every aggregation procedures (*xxx*_first_level()) to make sure that all the destination tables exist (they are created if not) and that they are partitioned as expected (a partition with including today is created if not).

- Drop_procedure

```
CREATE  PROCEDURE `Drop_working_table `()
```

This procedure drops persistent but working tables generated by other procedures (volume aggregation, etc). The procedure only drops tables that are older than at least 10 minutes and with a certain pattern (*name starting with an underscore*) specific to those tables.

This is applied to all databases whose name starts with *trafMon* or *tmon*.

This cleanup procedure has to be called manually: typically via a daily `crontab` task. It is not dangerous to invoke (no loss of information) and light in processing (simply drops entire tables at once).

3.2.5 Data Computations upon Report Generation

- Make_intervals

```
CREATE  PROCEDURE `Make_intervals`(IN `_dbName` VARCHAR(64),
                  `startDate` TIMESTAMP, `endDateDate` TIMESTAMP,
                  `intVal` INT, `unitVal` VARCHAR(10), `endBool` VARCHAR(10))
```

This procedure generates a set of dates with regular intervals; this set of dates is then used by the FTP Summary report to provide the user with easy to choose intervals for the report generation.

- Report_sum

```
CREATE  PROCEDURE `Report_sum`( IN `_dbName` VARCHAR(64),
                  `_ACTION` VARCHAR(100), `_GUID` VARCHAR(100),
                  `_TABLENAME` VARCHAR(100), `_RANGESTART` VARCHAR(100),
                  `_RANGEEND` VARCHAR(100), `_FLOW` VARCHAR(100) )
```

This procedure is used by all the protocol counters details reports to compute the sum of all specific counters during the defined time span for the defined flow (summary table at top of report). The flow value is an SQL expression, part of WHERE clause, which allows for a precise Flow or for a group of Flows that match the input template.

- Activity_aggr_volume

```
CREATE PROCEDURE `Activity_aggr_volume`(IN `_dbName` VARCHAR(64),
                  IN `_TableName` VARCHAR(100), IN `_rangeStart` VARCHAR(100),
                  IN `_rangeEnd` VARCHAR(100), IN `_GUID` VARCHAR(100),
                  IN `_activity` VARCHAR(100), IN `_location` VARCHAR(100),
                  IN `_ip` VARCHAR(100))
```

Replaced by Activity_aggr_volume_fast()

This procedure is used to obtain a large variety of information, from either NetFlow or the trafMon probes: volumes, bitrates, counters about TCP connections, FTP transfers, etc. The counters are aggregated depending on the filters. Note that this procedure works with any combination of specified _site/_mission/_ip (but can be slow at times), which is not the case with the following one.

- Activity_aggr_volume_fast

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `Activity_aggr_volume_fast`(
                        IN `_granularity` VARCHAR(100),
                        IN `_rangeStart` VARCHAR(100),
                        IN `_rangeEnd` VARCHAR(100) ), IN `_GUID` VARCHAR(100),
                        IN `_activity` VARCHAR(100), IN `_location` VARCHAR(100),
                        IN `_ip` VARCHAR(100) , IN `_srcTable` VARCHAR(100) )
```

This procedure works in the same way as Activity_aggr_volume() (it returns the same table with the same information), but it does so by querying dedicated tables, activityvolumetable_aggr_1d and activityvolumetable_aggr_1h. As such, it is extremely fast, contrary to other procedures which outputs the same kind of information. The results of this procedure are used to generate reports.

The `_srcTable` decides on the use of NetFlow data (= 'netflow') instead of default probe data.

This procedure is the main BIRT Data Set of all synthesis report templates.

3.3 NETFLOW DATA COLLECTION

NetFlow v5 or v9 (or sFlow or IPFIX) observations are automatically received and logged on the collector system via utilities provided by CERT® of SiLK™. The service `rwflowpack` is running on the collector and listen on the port 9991 (configurable). The main configuration file is located in `/var/silk/sensor.conf` (path configured in `/usr/local/etc/rwflowpack.conf`).

NetFlow PDUs are logged in a tree of binary files root at `/var/silk/data/` (path configurable) and retrieved via the script `trafMon_FormatNetFlow.py`. This script pulls the observations for a specific range of time (via the `rwfilter` and `rwcut` SiLK utilities).

```
## Load data from NetFlow SiLK logs every hour
15 * * * * python /opt/trafMon/bin/trafMon_FormatNetFlow.py -H 1
                                          >> /var/log/trafMon/cron.log 2>&1
```

Those observations are stored in the collector directory and formatted as text in order to be processed by the `trafMon_loader.py` script.

As those observations are not pre-aggregated by the trafMon collector software, they have to be separated in per-minute observations.

The NetFlow observations do not provide extensive per-application details: they are limited to IP, port, protocol, number of packets, number of bytes, start time, end time and duration.

NetFlow observations are used to provide volume distribution between hosts/locations/activities.

3.4 DATABASE REGULAR LOADING AND AGGREGATING PYTHON SCRIPT

One script, *trafMon_loader.py*, is to be regularly executed, via `crontab`, as in

```
## Load collector data files into the DB
##    only when trafMon_loader.py returns with status 0,
##       then trafMon_updateIpInfo.py - partial - is executed to resolve
##                                          IP addresses with no DNS name
*/10 * * * * python /opt/trafMon/bin/trafMon_loader.py -p /var/trafMon/collector
                                                  >>/var/log/trafMon/cron.log 2>&1
   && python /opt/trafMon/bin/trafMon_updateIpInfo.py -p /etc/trafMon/ipInfo.ini
                                                  >>/var/log/trafMon/cron.log 2>&1
```

The script does the following steps:

- Moves all pending collector output files to a working directory (`/var/trafMon/collector/work/`), which are then merged as one per type (under`/var/trafMon/collector/mergedFiles/`) and also archived in a compressed tarfile (`/var/trafMon/collector/done/`*YYYMMDDThhmm*`.tbz`).
- Prepare the NetFlow collected data with intervals of a minute.
- For each type of observations, creates one merged file by concatenating all corresponding per-minute chunk files.
- It then bulk loads (with replace) definitional data (flow instances, metric histogram slices and flow class hops lists), with potential replace upon clash of key, directly into their permanent definitional tables.
- Then bulk loads, with replace, all other files into corresponding temporary tables.
- For TCP connections, FTP file transfers and 1way individual observations, temporary records are inserted, with replace, into their respective persistent table. The replace statement ensures that only the last update results are kept for a given connection or transfer record. The replace has no effect on 1way observations, whose numeric key is not explicit in inserted data: auto-increment.
- For each other temporary tables, it invokes to corresponding aggregation stored procedure (see section 3.2.1 above), which updates/complements the aggregate at 1 minute, then also updates/inserts the further aggregates at 1 day and at 1 month.

The script use a UNIX socket to communicate with MySQL, therefore the network is not used and no password is sent through the network (in the deployment baseline).

Because the bulk load is directed to temporary tables and is accompanied by smart aggregation and update/extension of per-day aggregate tables, it is not implemented via invocation of the **mysqlimport** command, but directly via the LOAD DATA SQL statement. The Python script relies on the **MySQL connector for Python** software library.

3.4.1 IP Addresses Geolocation

Just after each execution of the `trafMon_loader.py` script, another script is executed, which is dedicated to resolving newly found IP addresses, `trafMon_updateIpInfo.py`.

This script fetches all non-resolved IP addresses from the database and query its DNS server: if the server resolves the IP address, the script updates it, afer all addresses have been queried. Otherwise, the IP is kept in place of a proper DNS resolution.

Once a week or less frequently, the script is also executed with `--all` IP addresses as input. It results in a full update of all the IPs with newly resolved DNS.

On top of that, this script also tag all IP addresses with other kind of information:

- A country
- A location
- An activity
- A city
- An ASN
- Some comments

These data can come from two different places: an offline geolocation database (GeoLite2™ from MaxMind® Inc.) pre-loaded on the system, and a configuration input file, provided as an argument to the script, which provides information about known IP addresses for which the geolocation process is not needed.

At the end of its execution, this script stores its results into a dedicated table, ***ipinfotable***, by doing a REPLACE into it.

3.4.2 Database Regular Aggregation

For preparing and alleviating the report generation work induced by the synthesis reports (see below), fully decorated volume tables are prepared once a day, on the basis of probe produced IP volume data (and some FTP and TCP observations) and, optionally, on the basis of NetFlow volume data:

```
## Update of ipcttable with re-assigning data bytes of passive FTP transfers to
## the flow with the FTP control connection
##                      (null from/to arguments means do for yesterday)
## THEN
## fill activity/location volume table every day with yesterday's data
##         (must be executed *AFTER* update_ftp_data_in_ipcttable every day)
## the user trafmon must have a $HOME/.my.cnf with permission 0400 for auto
## login as the tmon_birt read-only/call-only user into MySQL:
## % cat ~/.my.cnf
## [mysql]
## user=tmon_birt
## password=xxxx
```

```
47 0 * * * mysql -e 'CALL trafMon_template.Update_ftp_data_in_ipcttable(
                     "trafMon", null, null)' >>/var/log/trafMon/cron.log 2>&1 ;
           mysql -e 'CALL trafMon_template.Aggr_activityvolumetable_first_level(
                     "trafMon", null, null)' >>/var/log/trafMon/cron.log 2>&1
```

and optionally for NetFlow:

```
#
## Fill NetFlow volume table every day
2 3  * * * mysql -e 'CALL
trafMon_template.Aggr_activityvolumetable_netflow_first_level(
                     "trafMon", null, null)' >> /var/log/trafMon/cron.log 2>&1
```

3.4.3 Database Partitions and Efficient Clean-up

When creating entries for a new day in persistent tables, a new dedicated partition is created if needed. The partition covers a different period of time depending on the granularity of the table (every day for 1 second, every 8 days for 1 hour, every 31 days for 1 day).

Therefore in normal operations, detailed data are partitioned into separate physical sub-tables. Partitions have two advantages:

- Segmenting the data has a performance impact, avoiding to fetch old irrelevant data when querying a recent period of time;
- Destroying ancient data does not involve an expensive per row DELETE WHERE statement: it is replaced by a simple and efficient DROP PARTITION.

A cleanup database procedure, **`Partition_drop()`**, is provided that preserves only the last `_keepDays` (at least) in the given `_dbName._tableName`. The trafMon tool administrator may carefully automate the gradual cleanup of ancient file grain data, via `crontab`:

```
## DO THIS WITH CARE: drop the partitions with quickly growing fine grain
##                    data at 1 minute and that are about 90 days ago
##                    e.g. the IP size histograms
## every 3 day at 7:16
#16 7 */3 * * mysql -e 'CALL trafMon_template.Partition_drop(
                                        "trafMon", "ipsz_table_aggr_1m", 90)'
                                               >> /var/factseo/log/cron.log 2>&1
```

In addition, working tables with intermediate results, which cannot be declared temporary , nor can be removed by the tasks that produce them, must be regularly dropped by `crontab`:

```
## CLEAN DATABASE of the '_xxx' working tables
34 * * * * mysql -e 'CALL trafMon_template.Drop_working_tables()'
                                               >> /var/factseo/log/cron.log 2>&1
```

3.5 DATABASE USERS

Two users have been created in MySQL database for different purposes described below.
These users identifies themselves with passwords specified in the following file:

```
[DatabaseURL]
db_name = trafMon
db_url = unix:///var/lib/mysql/mysql.sock
#db_name = trafMon_2
#db_url = tcp://db_server:3306

[TMonloaderCredentials]
trafMon_loader_user = tmon_db
trafMon_loader_pwd = GuvfVfZlErnq/JevgrgensZbaQOHfreCnffjbeq

[TMonReportCredentials]
trafMon_report_user = tmon_birt
trafMon_report_pwd = GuvfVfZlErnq-BaylgensZbaQOHfreCnffjbeq
```

Passwords are (rot13) encoded and a decode function is applied in each script using them.
MySQL encrypts passwords stored in the user table using its own algorithm.

3.5.1 Database management user

For purposes of regular loading, user ‘tmon_db’ has been created.
The following privileges has been assigned:

- Global privileges: FILE (loading data from text file)
- For database trafMon*xxx* (trafMon% SQL pattern):
  - SELECT,INSERT,UPDATE,DELETE,EXECUTE (stored procedures calls)
  - CREATE,ALTER,DROP,CREATE TEMPORARY TABLES
- For database tmon_template: SELECT (temporary tables template, covered by above)
- For database information_schema: SELECT (partitions management)

3.5.2 Database reporting user

For purposes of BIRT reporting tool, user ‘**tmon_birt’** has been created.
The only privileges allowed are SELECT and EXECUTE on trafMon*xxx* (trafMon% SQL pattern) databases.

3.6 BIRT REPORTING

3.6.1 Selected Tools

The reporting is implemented through the use of public-domain Eclipse/BIRT reporting tool. This tool also exists as a commercial product from Open Text Corp. (formerly Actuate Corp.).

Using the same template and fed by the same data BIRT Engine is able to produce indifferently Web reports (HTML) and electronic documents (PDF, WORD doc, OpenOffice doc).

BIRT can produce reports mixing texts, simple variables, tables and charts from a wide palette. Multiple data sets can be shown in a single chart, possibly with multiple Y axes. XY plots permit to represent, in the axes, linear, logarithmic or date/time values. Furthermore, its can present , in a same report, a mix of data sets extracted from several different data sources, possibly for totally different types:

- All types of public domain or commercial SQL relational DBMS,
- XML files,
- data streams,
- ...

In addition to the data retrieved from the data source (e.g. SQL queries results), BIRT implements the JavaScript language that allows to further transform and or aggregate the input data.

A given report template does not freeze the data it presents. Through the use of report parameters, selected at run time, the template can be applied to custom-selected data.

Any object in the report can also be an hyperlink, either inside the report itself, or towards another (sub)report, or to any URL. Cross report hyperlinks also allow to pass data items from the calling report as values for sub-report parameters.

Report templates can be available with a Web-based BIRT runtime Engine, installed on a Java Web server product, for the on-demand generation of reports with custom-defined parameter values.

For trafMon the public-domain Apache Tomcat7 has been selected. Its security has been enforced through the use of SSL HTTPS encrypted connection (with self-signed certificate). And HTTP Basic authentication has been activated to force users to provide username/password, through the privacy protected connection.

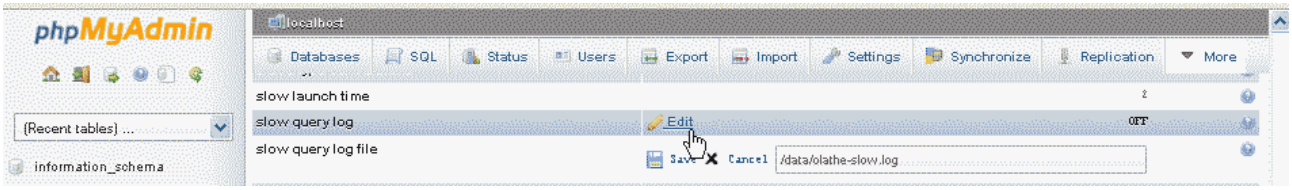
3.6.2 Expert User

In addition to the normal user basic access to the interactive Web reports, an Apache HTTP server with ***phpMyAdmin*** public domain application is also installed, allowing an **expert user** to custom query directly the data tables and, according to the database

authorisation profile of the username he mentions in its Web-based login form, the expert user will be allowed to modify the trafMon data and even to administer the DBMS. With the necessary privileges, he will be able to kill a (too resource-consuming) query in progress too.

Currently, with MySQL version 5.6.31, slow queries part of report generation, but also launched by the expert user, can optionally be detected by the server and written in the `slow_query_log`:

- System Variable `slow_query_log` is a Boolean that dynamically controls whether such **logging is enabled/disabled**
- System Variable `slow_query_log_file` http://dev.mysql.com/doc/refman/5.6/en/server-system-variables.html - sysvar_general_log is the **full pathname** of the slow log.



- System Variable `long_query_time` is the **threshold in seconds**

Later, with version above **MySQL 5.7.4**, it will be possible to specify a **maximum time for executing a query**. This will permit to protect the DBMS from inconsequently huge queries launched via the expert user interface.

3.6.3 BIRT Report Templates

All reports refer to a common Data Source, by default the MySQL database **`trafMon`** accessed, through the JDBC connector for Java, locally via a Unix socket pathname, using the account **tmon_birt** and its encrypted password. This is specified in the BIRT library file `Library/trafMonDb.rptlibrary`.

3.6.3.1 Protocol Details Reports

An initial set of reports have been implemented, focusing at the detailed protocol observations collected by the trafMon probes.

- one report, `FTP_Summary.rptdesign`, is standalone and presents top-10 synthesis views of the FTP File Transfers
- A set of reports implement a coherent presentation through navigation:
  - This first one is the Flow Instance report, with tabular presentation. It exists
  - There is one sub-report per protocol counters: `IP_Counters.rptdesign`, `ICMP_Counters.rptdesign`, `UDP_Counters.rptdesign`, `TCP_Counters.rptdesign`, `FTP_Counters.rptdesign`

- There is one sub-report for IP Size Distribution: `IP_Size_distribution.rptdesign`
- There is one sub-report for Round-trip Delays: `TwoWayDelays.rptdesign`
- There are two sub-report for One-Way observations: `OneWayLatencies.rptdesign` and `OneWay_Counters.rptdesign`

This set of reports derives from a common page. They share two common BIRT libraries, the DataSource for accessing the database (`Library/trafMonDb.rptlibrary`)and several Data Source queries and parameters definitions (`Library/trafMonLibrary.rptlibrary`). And they share a common javascript library with the per-report ad hoc routines, `Scripts/FormatFileSize.js`.

3.6.3.2 Synthesis Reports

This implementation also provides a set of reports presenting the traffic dissemination between hosts / sites and missions:

- MissionManagerReport.rptdesign
- OperatorReport.rptdesign
- OperatorReport_conv.rptdesign

Besides the access to the Data Source (MySQL database instance and login), those reports also encompass hyperlinks to each/other, referred by the `urlBirt` parameter (the *host:port* head of the Apache Tomcat URI, e.g. “https://127.0.0.1:8443/”) and by the `url` parameter (the Apache HTTPD URI for accessing the trafMon Menu Bar for synthesis reports, e.g. https://127.0.0.1/trafmon/#!/volume, in order to re-display the top menu bar after following the hyperlink) Those values are defined in `Library/url.rptlibrary`.

3.6.3.3 Report Template Editor

Edition of report templates is achieved via the BIRT Report Designer GUI tool, part of the Eclipse environment.

- On the CentOS Linux collector computer, the pre-configured Eclipse with BIRT Report Designer is typically installed under `/opt/eclipse/`. The user must execute /opt/eclipse/eclipse command to launch the designer.
- An **All-in-one** version of Eclipse with BIRT Designer for **Windows** PC (32 or 64 bit) and for **Mac OS X** (32 or 64 bit) is also available for download via the URL **https://www.eclipse.org/birt/**

3.6.4 Apache Tomcat Environment for On-demand Generation of trafMon Reports

The trafMon URL for directly accessing a given BIRT report is structured as

https://*trafMonServer*:8443**/birt/**run?__**report=trafMon_reports/**xxx.rptdesign

This means that the BIRT runtime environment has been installed in the directory

`/var/lib/tomcat/webapps`**`/birt/`**

This also means that the .rptdesign files are stored in the sub-directory:

`/var/lib/tomcat/webapps`**`/birt/trafMon_reports/`**

and of course with its tree sub-directories (`Library/` and `Scripts/`) as per above.

When using **run** in the report URL, the user is presented with a continuous potentially very long page he must scroll through.

When using **frameset** instead of run in the report URL, the user is presented the BIRT Viewer application. This allows to see paginated report and to navigate on a per-page basis. Additional icons permit to, among others, export the report in HTML or PDF or MS-WORD or OpenOffice document format. And the user is also able to export the actual data from a report object (e.g. a chart).

Due to the interactive menu bar (see section 3.6.5 below), the set of report parameters has been more or less unified and generalised. Hence, it is not that intuitive to execute the reports directly, some parameters (especially the start/end Day are redundant vs the start/end Time. Furthermore, the (limited) capability for the BIRT parameters to query themselves the database for a list of possible values has been deactivated in the current version, because replaced by the JavaScript bar of dynamic menus and its PHP queries

When accessing the FTP_Summary.rptdesign report, the user must provide the DBname, the Granularity (Minute, hour or Day), the StartTime and EndTime. StartDay and EndDay (used by the Menu Bar to pre-determine the rough time span) can be left NULL.

When accessing all other protocol details reports, the list of report parameters reflect also the choices in the Menu Bar to apply for an individual Flow instance, or to sum-up the per-flow figures matching one or more of the flow identification components: IP1, IP2, Direction and/or probe Interface (with the risk of double counting). The parameters are:

- `DBname`,
- `TableName`: even though this is implied by the report type, it is the (radix) of) the data table that provides the values populating the report instance,
- Either `flowID` != “none”
- Or (`flowID` == “none”):
  - `IP1` and/or,
  - `IP2` and/or,
  - `Direction`: “<”, “>”, “<>” (bi-directional) or other (don’t care), and/or,
  - `Interface`,
- `Granularity`: “Minute”, “Hour”, or other (means “Day”),

- `StartTime`,
- `EndTime`,
- `StartDay` (unused, expoited by the Menu Bar when selecting the `StartTime`)
- `EndDate` (unused, expoited by the Menu Bar when selecting the `EndTime`)

When accessing the synthesis reports, the parameters are:

- `DBname`,
- `activity`: “any” or a valid Activity name,
- `location`: “any” or a valid Location name,
- `host`: “any” or a valid hot IP address,
- `source`: “probe” or other (means “netflow”),
- `view`: “IP” (host IP address) or other (means “DNS” hostname),
- `granularity`: “Months”, “Hours”, or other (means “Days”),
- `top` (integer > 0),
- `treshold` (spelling typo, should be threshold): integer value in bit/sec,
- `rangeStart,`
- `rangeEnd,`
- `url` (from library, see 3.6.3.2 above),
- `urlBirt` (from library, see 3.6.3.2 above)

3.6.5 Apache Httpd Environment for On-demand Generation of trafMon Reports

A web menu is accessible at https://*trafMonServer*/trafMon, this open the Menu Bar for synthesis reports, with URI replaced by https://*trafMonServer*/trafMon/#!/volume. When switching to “Detailed”, the URI for details reports is used: https://*trafMonServer*/trafMon/#!/birt.

This menu provides easy access to the different parameters of all the reports. The purpose of this menu is to ease the navigation/drill-down generation of multiples reports and to provide a simple way to refine input parameters.

The web server is hosted on the central trafMon server and the files are located in `/var/www/html/`, the queries to the database are made by php and the client-based interaction are managed via JavaScript (AngularJS and JQuery).

The report generation is made by updating an iFrame inside the page and calling the Tomcat report generation.

The different uses of this menu are further explained in the User Manual.

3.6.5.1 trafMon_web App Source Structure

```
$ ls -R trafMon_web/
trafMon_web/:
README  app

trafMon_web/app:
404.html  index.html  php  scripts  styles  views

trafMon_web/app/php:
Activity.php  Host.php  IP.php  IP1.php  IP2.php  Location.php  database.php
dateend.php  datestart.php  flow.php  host_info.php  include.php  interface.php
rangeend.php  rangestart.php

trafMon_web/app/scripts:
app.js  controllers  directives  filters  services

trafMon_web/app/scripts/controllers:
about.js  birt.js  main.js  volume.js

trafMon_web/app/scripts/directives:
mydirective.js  ngdate.js

trafMon_web/app/scripts/filters:
unique.js

trafMon_web/app/scripts/services:
birturl.js  phpurl.js

trafMon_web/app/styles:
Supernice.css  custom.css  jquery-ui.min.css

trafMon_web/app/views:
about.html  birt.html  main.html  volume.html
```

The `trafMon_web/app/scripts/`**`app.js`** maps the URI to the right menu:

```
/**
 * @ngdoc overview
 * @name trafMonWebApp
 * @description
 * # trafMonWebApp
 *
 * Main module of the application.
 */
angular
  .module('trafMonWebApp', [
    'ngAnimate',
    'ngAria',
    'ngCookies',
    'ngMessages',
```

```
    'ngResource',
    'ngRoute',
    'ngSanitize',
    'ngMaterial',
    'angularSpinner'
  ])
  .config(function ($routeProvider) {
    $routeProvider
      .when('/volume', {
        templateUrl: 'views/volume.html',
        controller: 'VolumeCtrl',
        controllerAs: 'volume'
      })
      .when('/birt', {
        templateUrl: 'views/birt.html',
        controller: 'BirtCtrl',
        controllerAs: 'birt'
      })
      .when('/about', {
        templateUrl: 'views/about.html',
        controller: 'AboutCtrl',
        controllerAs: 'about'
      })
      .otherwise({
        redirectTo: '/volume'
      });
  })
  .constant('PHP_URL', 'https://127.0.0.1/');
```

The `trafMon_web/app/scripts/services/`**`birturl.js`** defined the URL prefix for invoking the BIRT report generation:

```
/*
 * TO BE ADAPTED TO YOUR ENVIRONMENT
 */
/**
 * @ngdoc service
 * @name trafMonWebApp.birtUrl
 * @description
 * # birtUrl
 * Constant in the trafMonWebApp.
 */
angular.module('trafMonWebApp')
  .constant('birtUrl', 'https://127.0.0.1:8443/birt/');
```

The various pull-down menus in the Menu Bar are populated via the corresponding SQL queries in the PHP files under `trafMon_web/app/php/`. The file `trafMon_web/app/php/`**`include.php`** contains the configurable definition on how to access the target MySQL database server. The password is encoded in rot13.

The date selection for the synthesis reports is based on JQuery-UI Datepicker widget, which has been integrated in the AngularJS environment, via the directive in `trafMon_web/app/scripts/directives/ngdate.js`.

The JavaScript code follows the AngularJS View/Controller conventions. Data are handled as JSON structures populated by the PHP code.

3.6.5.2 trafMon_web App Packages Dependencies.

The trafMon web App can be put under Yeoman/Bower/Grunt/Usemin control. Usemin allows the production of a rather cryptic compact variant of all the JavaScript code, which downloads more quickly in the browser. Grunt allows to run the Web App from the development environment, to test/debug/tune/fix the code.

The *trafMon Installation Guide* explains the simplest way to download the necessary third-party public domain modules and to produce a running app (although not “minimized”):

```
trafmon % cd /opt/trafMon/trafMon_web/app/
trafmon % cat .bowerrc
{
  "directory": "bower_components"
}
trafmon % bower install angular angular-animate angular-aria angular-cookies
trafmon % bower install angular-material angular-messages angular-resource
trafmon % bower install angular-route angular-sanitize angular-spinner
trafmon % bower install angular-touch bootstrap jquery jquery-ui spin.js
trafmon % ls bower_components/
angular           angular-material  angular-route     bootstrap
angular-animate   angular-messages  angular-sanitize  jquery
angular-aria      angular-mocks     angular-spinner   jquery-ui
angular-cookies   angular-resource  angular-touch     spin.js
```

Verify that the pathnames in `index.html` are correctly referring inside the substructure of every installed package under `bower_components/`.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>trafMon Reports Viewer</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <!-- Place favicon.ico and apple-touch-icon.png in the root directory -->
    <!-- build:css(.) styles/vendor.css -->
    <!-- bower:css -->
    <link rel="stylesheet" href="bower_components/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="bower_components/angular-material/angular-material.css"
/>
    <!-- endbower -->
    <!-- endbuild -->
    <!-- build:css(.tmp) styles/main.css -->
    <link rel="stylesheet" href="styles/Supernice.css">
    <link rel="stylesheet" href="styles/custom.css">
    <link rel="stylesheet" href="styles/jquery-ui.min.css">
    <!-- endbuild -->
  </head>
  <body ng-app="trafMonWebApp">
    <div class="container-fluid">
    <div ng-view=""></div>
    </div>
    <iframe class="iframe-basic-volume" id="frame" src="" width="100%"
height="100%"></iframe>
    <!-- build:js(.) scripts/vendor.js -->
    <!-- bower:js -->
    <script src="bower_components/jquery/dist/jquery.js"></script>
```

```
    <script src="bower_components/angular/angular.js"></script>
    <script src="bower_components/bootstrap/dist/js/bootstrap.js"></script>
    <script src="bower_components/angular-animate/angular-animate.js"></script>
    <script src="bower_components/angular-aria/angular-aria.js"></script>
    <script src="bower_components/angular-cookies/angular-cookies.js"></script>
    <script src="bower_components/angular-messages/angular-messages.js"></script>
    <script src="bower_components/angular-resource/angular-resource.js"></script>
    <script src="bower_components/angular-route/angular-route.js"></script>
    <script src="bower_components/angular-sanitize/angular-sanitize.js"></script>
    <script src="bower_components/angular-touch/angular-touch.js"></script>
    <script src="bower_components/angular-material/angular-material.js"></script>
    <!-- endbower -->
    <script src="bower_components/jquery-ui/jquery-ui.min.js"></script>
    <script src="bower_components/spin.js/spin.ts"></script>
    <script src="bower_components/angular-spinner/dist/angular-spinner.js"></script>
    <!-- endbuild -->
        <!-- build:js({.tmp,app}) scripts/scripts.js -->
        <script src="scripts/app.js"></script>
        <script src="scripts/controllers/main.js"></script>
        <script src="scripts/controllers/about.js"></script>
        <script src="scripts/controllers/birt.js"></script>
        <script src="scripts/controllers/volume.js"></script>
        <script src="scripts/directives/mydirective.js"></script>
        <script src="scripts/filters/unique.js"></script>
        <script src="scripts/directives/ngdate.js"></script>
        <script src="scripts/services/phpurl.js"></script>
        <script src="scripts/services/birturl.js"></script>
        <!-- endbuild -->
</body>
</html>
```

3.6.6 Apache Tomcat Environment for Batch Generation of trafMon Reports

A second BIRT Runtime can been installed under **`/opt/birt-runtime-xyz/`**.

This is used for the batch report production. More precisely, the generating command is executed via the path **`/opt/tmon/bin/genReport.sh`** which is a link to **`/opt/birt-runtime-xyz/ReportEngine/genReport.sh`.**

```
$ /opt/tmon/bin/genReport.sh --help
Help for ReportRunner

--mode/-m [run|render|runrender] [options] [rptdesign|rptdocument]
        The default mode is runrender.
To see options for run mode, use:
        --help/-h run
To see options for render mode, use:
        --help/-h render
To see options for runrender mode, use:
        --help/-h runrender
Print current message, use --help/-h

$ /opt/tmon/bin/genReport.sh --help render
ReportRunner's RENDER mode:
--mode/-m render [options] <rptdocument file>
```

```
where options could be:
        --format/-f [HTML|PDF]
        --output/-o <target file>
        --page/-n <pageNumber>
        --locale/-l <locale>     default is english
        --config/-c <"configName=configValue">
        --renderOption/-r <"optionName=optionValue">
        --file/-F <file>

1. configs/renderOptions in command line will overide those in file
2. config/renderOption name can not include characters such as ' ', '=', ':'
use "--help/-h configNames" for a list of configurables
use "--help/-h renderOptions" for a list of render options
use "--help/-h file" for options in <file>
```

The script **`trafMon_detailReportGen.py`** relies on the genReport.sh script to produce those protocol details reports listed at the top of the script for the IP addresses matching the expressions in the configuration file /etc/trafMon/report/*FILENAME*:

```
$ ./trafMon_detailReportGen.py --help
Usage: trafMon_detailReportGen.py [options]

Options:
  -h, --help            show this help message and exit
  -l, --localConfig     if -l is specified, db.cred file is fetched from the
                        current directory. Default: /etc/trafMon/cred/
  -f FILENAME, --filename=FILENAME
                        Give a pathname or filename containing IP address
                        patterns in concerned Flow Instances. This file
                        basename is also the root of the tree of generated
                        reports. When relative, the file is fetched from
                        current directory when -l is specified, otherwise from
                        default /etc/trafMon/report/
  -D destination, --destination=destination
                        Destination directory. Default to
                        '/var/trafMon/reports/'.
  -s STARTDATE, --startDate=STARTDATE
                        Give a start date in format: 'YYYY-MM-DD'
  -e ENDDATE, --endDate=ENDDATE
                        Optionally give an end date in format: 'YYYY-MM-DD'
  -t TIMESPAN, --timespan=TIMESPAN
                        Without endDate: choose between 'weekly' or 'monthly'
                        report. With endDate: give any identifier for this
                        type of reports.
  -L LOGDIR, --logFileDirectory=LOGDIR
                        Path to log directory. Default: /var/log/trafMon/
  -T TEMPLATESFOLDER, --reportTemplatesDirectory=TEMPLATESFOLDER
                        Path to trafMon report templates directory. Default:
                        /opt/trafMon/report/
  -g GENREPORT, --genReport_sh=GENREPORT
                        Full pathname to the Birt runtime 'genReport.sh'
                        utility. Default to /opt/trafMon/bin/genReport.sh,
                        which is typically a symbolic link to the BIRT RunTime
                        installation/ReportEngine/genReport.sh
```

This script is primarily aimed at producing the set of reports related to specified set of servers.

The reports are listed in the reportsToTables structure inside the python scripts:

```
reportsToTables = {
    "IP_Counters" : "ipcttable_aggr_",
    "TCP_Counters" : "tcpcttable_aggr_",
    "UDP_Counters" : "udpcttable_aggr_",
    "FTP_Counters" : "ftpcttable_aggr_",
    "ICMP_Counters" : "icmpcttable_aggr_",
    "IP_Size_distribution" : "ipsztable_aggr_",
    "TwoWayDelays" : "twowaydelaytable_aggr_",
    "OneWayDelays" : "onewaylatencytable_aggr_",
    "OneWay_Counters" : "onewaycttable_aggr_",
    "FTP_Details" : "ftpxfrtable",
    "TCP_Details" : "tcpcontable"
}
```

A **Mission FILENAME** for the batch reports generation script, mentioned either as an absolute pathname or relative to **`/etc/tmon/report/`**, contains a **list of IP addresses regular expression patterns**. Reports are generated for all available protocol details templates related to Flow Instances whose one (or both) IP address field matches a pattern from the Mission file.

If optional **ENDDATE** is given, per-day reports are produced from **STARTDATE** (inclusive) to **ENDDATE (excluded).**

Otherwise, the **TIMESPAN** is used to determine boundaries. It is either

- "weekly": per-day reports from Monday to Sunday of the week containing **STARTDATE**
- or “monthly”: per-day reports for the month containing **STARTDATE**

Even when not used to determine date boundaries, a string value must be given to TIMESPAN

Reports are produced under **`/var/trafMon/reports/`*`TIMESPAN`*`/`*`FILENAME`*`/from_`*`STARTDATE`*`_to_`*`ENDDATE`*`/`**.

Typically, invocations of the proxyGenRep.py are scheduled in the user factseo crontab, as in:

```
## Generate detail protocol reports about myServers for last month every month
45 5 1 * * python /opt/trafMon/bin/trafMon_detailReportGen.py -f
             myServers.genAddrs -s `date -d "yesterday" "+%Y-%m-01"` -t monthly
             >> /var/log/trafMon/cron.log 2>&1
```

The script **`trafMon_volumeReportGen.py`** relies on the genReport.sh script to produce the requested synthesis report [manager, operator, conversation]:

```
$ ./trafMon_volumeReportGen.py --help
Usage: trafMon_volumeReportGen.py [options]

Options:
  -h, --help            show this help message and exit
  -d DBname, --db=DBname
                        Database to be used. Default to 'trafMon'.
  -r report, --report=report
                        Type of synthesis report to be generated. Possible
                        choices are: [manager, operator, conversation].
                        Default to 'manager'.
  -D destination, --destination=destination
                        Destination directory. Default to
                        '/var/trafMon/reports/2020/10/28', where the
                        'YYYY/MM/DD' part is the generation time of the report
                        (today).
  -t top, --top=top     Top-N to be used. Possible choices are: [5, 10, 15,
                        20, 25]. Default to top-5.
  -T threshold, --threshold=threshold
                        Threshold bandwidth in b/s to be used. Possible
                        choices are: [0, 1000, 10000, 50000, 100000, 500000].
                        Default to 1000.
  -A activityName, --activity=activityName
                        Activity to be used. Default to 'any'. Use quotes if
                        the activity name include a space.
  -L locationName, --location=locationName
                        Location to be used. Default to 'any'. Use quotes if
                        the location name include a space.
  -H hostName, --host=hostName
                        Host to be used. Default to 'any'. Use quotes if the
                        host name include a space.
  -s startDate, --startDate=startDate
                        Start date to be used (format: 'YYYY-MM-DD'). Default
                        to first day of previous month.
  -e endDate, --endDate=endDate
                        End date to be used (format: 'YYYY-MM-DD'). Default to
                        last day of previous month.
  -l LOGDIR, --logFileDirectory=LOGDIR
                        Path to log directory (default: /var/log/trafMon/)
  -R TEMPLATESFOLDER, --reportTemplatesDirectory=TEMPLATESFOLDER
                        Path to trafMon report templates directory. Default:
                        /opt/trafMon/trafMon_reports/
  -g GENREPORT, --genReport_sh=GENREPORT
                        Full pathname to the Birt runtime 'genReport.sh'
                        utility. Default to /opt/trafMon/bin/genReport.sh,
                        which is typically a symbolic link to the BIRT RunTime
                        installation/ReportEngine/genReport.sh
```

3.7 DATA MAINTENANCE

For permitting smooth preparation/validation/tuning phase, the collector raw observations are voluntarily not removed from the collector computer disk. These are left under **`/var/trafMon/collector/done/`**.

However, the log files are cleaned up after a certain time span:

- on both probe and collector computers, the online functions trace logs (and db loading logs on the collector) **under `/var/log/trafmon/`** are processed via custom **logrotate script**:

```
# cat /etc/logrotate.d/trafmon
/var/log/trafMon/*.log {
  # When some files have been mistakenly created as root,
  # this can perturbate the automated operations, so restore ownership
  firstaction
  chown -R trafmon:trafmon /var/log/trafMon
  endscript

  lastaction
  chown -R trafmon:trafmon /var/log/trafMon
  endscript

  rotate 300000
  # on CentOS 6.x, use daily insead of not yet suppoprted hourly
  hourly
  size 200M
  compress
  delaycompress
  missingok
  notifempty
  create 0644 trafmon trafmon
}}
```

- on the collector computer, a custom **logrotate script** has also been created for the MySQL optional logs stored in their standard location **`/var/log/mysqld/`**:

```
# cat /etc/logrotate.d/mysql
/var/log/mysqld/*.log {
        # create 600 mysql mysql
        notifempty
        daily
        rotate 7
        missingok
        compress
        lastaction
          chown -R mysql:mysql /var/log/mysqld/
          if test -x /usr/bin/mysqladmin && /usr/bin/mysqladmin ping
&>/dev/null
          then
             /usr/bin/mysqladmin flush-logs
          fi
        endscript
}
```

Database regular cleanup is described in section3.4.3 above.

Database back-up and restore can be achieved via the two standard MySQL utilities:

- **mysqldump**
- **mysqlimport**

or via the **phpmyadmin tool: Export and Import tabs**

Computers disk partitions can be efficiently back-ed up and (selectively) restored through the classical Unix utilities:

- **dump**
- **restore**

4. TRAFMON INTERFACE CONTROL DOCUMENTATION

4.1 TRAFMON ONLINE FUNCTIONS XML CONFIGURATION INTERFACE

Being the `tmon_probe` or the `tmon_collector` online programs, the packet capture, observations gathering, measurements processing and regular output of performance data log file activities are governed by a single XML configuration file.

The configuration uses XML for its tags and tag attributes, neglecting text in between tags.

The XML obeys the syntax defined by a DTD called "`tmon.dtd`" which rules out part of the consistency of an XML configuration file and permits its verification upon loading.

The recommended practice is that the XML file refers to its DTD simply by its basename, meaning that the `tmon.dtd` must be located in the same directory as the XML configuration file, on every system where `tmon_probe` and/or `tmon_collector` are running.

```
<!DOCTYPE TrafMonConfig SYSTEM "tmon.dtd" [ ... ]>
```

By default, the configuration file is named `tmon.xml`. And by default, the configuration file and its DTD are read from `/etc/tmon/xml/`:

File *tmon_probe.h*:

```
#define TM_PROBE_CONFIG_PATH   "/etc/tmon"
#define TM_PROBE_XML_DIR       "xml"
```

File *tmon_collector.h*:

```
#define TM_COLL_CONFIG_PATH   "/etc/tmon"
#define TM_COLL_XML_DIR       "xml"
```

Command-line option permits to modify the pathname of the XML configuration file.

```
% ./tmon_probe
USAGE:
tmon_probe [-l] [-c configXML] [-n NEWconfigXML] probeName
        -l (local) means using ./tmon.xml, ./tmon-new.xml and,
                        if it exists, ./tmon_probe.diag
        -c means use the given XML, and nothing else
        -n monitors the given NEW XML for scheduled config update
                        (based on its `startAt' attribute
 If -l is NOT given, /etc/trafMon/xml/tmon.xml is used
 If -l and -n are NOT given, /etc/trafMon/xml/tmon-new.xml is looked at
 If -l is NOT given, or ./tmon_probe.diag doesn't exist,
                        /etc/trafMon/diag/<probeName>.diag is used
% ./tmon_collector
USAGE:
tmon_collector [-l] [-c configXML] [-n NEWconfigXML] collectorName
```

```
        -l (local) means using ./tmon.xml, ./tmon-new.xml and, if it exists,
./tmon_collector.diag
        -c means use the given XML, and nothing else
        -n monitors the given NEW XML for scheduled config update
                        (based on its `startAt' attribute
 If -l is NOT given, /etc/trafMon/xml/tmon.xml is used
 If -l and -n are NOT given, /etc/trafMon/xml/tmon-new.xml is looked at
 If -l is NOT given, or ./tmon_collector.diag doesn't exist,
                        /etc/trafMon/diag/<collectorName>.diag is used
```

Also, this file may mention a pathname where to fine the DTD, instead of only its basename. When just **tmon.dtd** is specified, it must be collocated with the XML configuration file:

```
<!DOCTYPE trafMonConfig SYSTEM "tmon.dtd" [ ... ]>
```

The XML configuration is assigned a version number (<TrafMonConfig> tag **serial** attribute) but also a Date/Time lower bound of validity (TrafMonConfig> tag **startAt** attribute), as in

```
<trafMonConfig serial="100" startAt="2020-08-04 13:29:00" pktSignBytes="3"
  maxTravelTime="30000" >
...

</trafMonConfig>
```

Upon start, the program produces the following kind of log messages:

```
20201012T164400.674881,tmon_probe[14969],WNG,tmon_probe.c:417:main,Auto-
detecting future config. update as './tmon-new.xml': at its 'startAt' time, this
file will be renamed as 'test.xml' and ./tmon_probe probe will auto-restart
20201012T164400.674967,tmon_probe[14969],TR0,tmon_probe.c:441:main,STARTING
probe probe (ID=11) using test.xml version 100 valid since 04 Aug 2020 13:29:00
```

More precisely, the program uses the same pathname as its startup XML configuration to build the name of a potential next update:

- By default /etc/tmon/xml/tmon-new.xml
- or ./tmon-new.xml, when program invoked with **–l option**
- or the pathname given by the **–n option**

While running, the program regularly checks, every minute, for the presence of a file with this name. In case there is one, the XML is parsed (and validated) and the **startAt** inspected. If the file is a correct new version and valid in the future, a timer is launched to let the program automatically re-start itself at the right time on this new version of the configuration, renamed as per the original (itself renamed with “.PREVIOUS” suffix). This restart happens immediately when the correct new version is already valid.

A pseudo new file with same start time and serial number its name is simply appended the “.UNCHANGED” suffix.

4.1.1 Definition of XML Configuration

The configuration is defined by a fully commented DTD file. The DTD enforces the structure and some other constraints. Further detailed constraints and the meaning/effects of configuration items are presented as comment text.

```
<!--
      Copyright (c) 2020 AETHIS s.a./n.v., Belgium. All rights reserved.
      www.trafmon.org

      Licensed under the Apache License, Version 2.0 (the "License");
      you may not use this file except in compliance with the License.
      You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

      Unless required by applicable law or agreed to in writing, software
      distributed under the License is distributed on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
      See the License for the specific language governing permissions and
      limitations under the License.
-->
<!-- trafMon XML DTD tmon.dtd ### Current version $Id:
a13d30010506c1a3f0d1e4c04ef9f41149b9c93a $-->
<!ELEMENT TrafMonConfig (Collector+, Probe+,GranularFlow+,
                                              (FlowClass | ReportLink)+) >
<!ATTLIST TrafMonConfig
                    serial          NMTOKEN   #REQUIRED
                    startAt         CDATA     #REQUIRED
                    pktSignBytes    NMTOKEN   "3"
                    maxTravelTime   NMTOKEN   "10000"
                    pduYoungWindow  NMTOKEN   "10"
>
          <!-- serial:      Config. version ID to which every PDU refers  -->
          <!--                 valid values are [0..255], wrapping        -->
          <!-- startAt:     Universal (UTC) Date/Time in sec at which to  -->
          <!--              switch to this TrafMonConfig serial number    -->
          <!--                Format: as per ISO 8601 date representation -->
          <!--                     YYYY-MM-DD hh:mm:ss                    -->
          <!-- pduCRCSize:  how many bytes of PDU content digest [1..3]   -->
          <!-- pduCRCFunc:  hash function for computing PDU content digest-->
          <!--                CURRENTLY ONLY "MD5" is supported           -->
          <!-- pktSignBytes: signature bytes of IP packet content digest  -->
          <!--                 valid values are [2..10]                   -->
          <!-- maxTravelTime: reasonable boundary, in milliseconds, for   -->
          <!--                any packet to travel through the network.   -->
          <!--                Outside this time window, packet of same    -->
          <!--                ReportFlow and same signature are considered-->
          <!--                different.                                  -->
          <!--                Furthermore, after this duration, an        -->
          <!--                incomplete obs. record, whose missing info  -->
          <!--                are expected from an alive probe, times-out -->
          <!--                in the collector (e.g. packet lost event)   -->
          <!--                 valid values are [100..50000]              -->
          <!-- pduYoungWindow: time window, in seconds in the past,       -->
          <!--                 where the first time in a server received  -->
```

```
          <!--                probe PDU is still considered fresh data   -->
          <!--                valid values are [3..3600]                 -->
<!ELEMENT Collector       (Addr,Output) >
<!ATTLIST Collector       name      ID                          #REQUIRED
                          ID        NMTOKEN                     #REQUIRED
                          descr     CDATA                       #IMPLIED
                          burstRate NMTOKEN                     "10"
>
          <!-- ID:          [256..]                                       -->
          <!-- burstRate:  how much PDU to expect in a burst              -->
          <!--                valid values are [10..65535]               -->
<!ELEMENT Addr            EMPTY >
<!ATTLIST Addr            ip        NMTOKEN                     "0.0.0.0"
                          port      NMTOKEN                     #REQUIRED
                          UDPBufferSize
                                    NMTOKEN                     "0"
>
          <!-- ip:               IP on which to bind to receive PDUs     -->
          <!-- port:             port number on which to listen          -->
          <!-- UDPBufferSize:    kernel buffer size, in kilobytes,       -->
          <!--                                       of UDP socket       -->
          <!--                    valid values are [0..65535]            -->
          <!--                     0 means the kernel default max size   -->
<!ELEMENT Output          EMPTY >
<!ATTLIST Output          dataFile     CDATA          "tmondata-%y%m%d%H%M%S"
                          eventFile    CDATA          "tmonevent-%y%m%d%H%M%S"
                          excepFile    CDATA          "tmonexcep-%y%m%d%H%M%S"
                          period       NMTOKEN                    #IMPLIED
>
          <!-- dataFile:         CSV-file radix name for data output      -->
          <!--                      supports strftime strings (%H%M...)   -->
          <!-- eventFile:        CSV-file radix name for event output     -->
          <!--                      supports strftime strings (%H%M...)   -->
          <!-- excepFile:        CSV-file radix name for exceptions output-->
          <!--                      supports strftime strings (%H%M...)   -->
          <!-- period:           number of minutes during which output    -->
          <!--                   accumulate in a same CVS-file            -->
          <!--                      ONLY VALID where strftime %M present  -->
          <!--                      Used as the modulo on the minute field-->
          <!--                    valid values are [1..59]                -->

<!ELEMENT Probe           ((CapFile | Interface+), PDUSending*, PDUSaving?) >
<!ATTLIST Probe           name         ID                         #REQUIRED
                          ID           NMTOKEN                    #REQUIRED
                          descr        CDATA                      #IMPLIED
>
          <!-- ID:          [0..255]                                      -->

<!ELEMENT CapFile       EMPTY >
<!ATTLIST CapFile         filename     CDATA                      #REQUIRED
                          ID           NMTOKEN                    #REQUIRED
                          expr         CDATA                      #IMPLIED
                          rate         (withDelay|fullSpeed)      "withDelay"
>
          <!-- filename:   Full pathname of packet capture file to read  -->
          <!-- id:         TrafMon-wide unique numeric ID of the probe   -->
          <!--             interface that can distinguish among granular -->
```

```
            <!--              flow instances                                    -->
            <!--                      valid values are [1..65535]               -->
            <!-- expr:        tcpdump-like packet capture filter expression     -->
            <!--                     WHEN VLAN PARTLY TAGS PRESENT              -->
            <!--                          INVOLVE vlan at end of expr           -->
            <!--                     MATCH only IP packets                      -->
            <!--                     DON'T use netmask based criteria           -->
            <!-- rate:        fullSpeed: captured pakets are processed          -->
            <!--                         untouched (with their original capt.-->
            <!--                         time) without waiting between each     -->
            <!--              withDelay: every packet has its capture time      -->
            <!--                         artificially translated by a fixed     -->
            <!--                         amount of time so as if the exact      -->
            <!--                         same traffic behaviour would occur     -->
            <!--                         'now'. The necessary variable delay    -->
            <!--                         is respected before processing each    -->
            <!--                         next packet                            -->

<!ELEMENT Interface         EMPTY >
<!ATTLIST Interface         name          NMTOKEN                    #REQUIRED
                            ID            NMTOKEN                    #REQUIRED
                            descr         CDATA                      #IMPLIED
                            snapLen       NMTOKEN                    "1600"
                            bufPacketCount
                                          NMTOKEN                    "70000"
                            expr          CDATA                      #IMPLIED
>
            <!-- id:          TrafMon-wide unique numeric ID of the probe     -->
            <!--              interface that can distinguish among granular   -->
            <!--              flow instances                                  -->
            <!--                      valid values are [1..65535]             -->
            <!-- snapLen:     maximum Ethernet frame Captured portion         -->
            <!--                      valid values are [125..65535]           -->
            <!--   On Linux:                                                  -->
            <!--       snapLen=125 leads to 122 Eth capture => IP len=108     -->
            <!--                                                              -->
            <!--   NOTE:                                                      -->
            <!--       Even on Ethernet (max MTU = 1500 bytes IP),            -->
            <!--       larger packets can be actually captured due to         -->
            <!--       reassembly being offloaded in the NIC Card             -->
            <!--       Linux Ethtool -k:  LRO - Large Receive Offload or      -->
            <!--                          or GRO - Generic Receive Offload    -->
            <!--       ATTEMPT IS MADE TO DEACTIVATE THIS and the Reception   -->
            <!--       Checksum processing offload upon initiatisation of     -->
            <!--       the probe capture interfaces                           -->
            <!--                                                              -->
            <!-- bufPacketCount:                                              -->
            <!--              How many packets (of ~ snapLen) could be        -->
            <!--              buffered upon traffic burst                     -->
            <!--                      valid values are [1000..1000000]        -->
            <!--                                                              -->
            <!-- expr:        tcpdump-like packet capture filter expression   -->
            <!--                     WHEN VLAN PARTLY TAGS PRESENT            -->
            <!--                          INVOLVE vlan at end of expr         -->
            <!--                     MATCH only IP packets                    -->
            <!--                     DON'T use netmask based criteria         -->
            <!--     LIMITATION: care must been taken with VLAN packets:      -->
```

```
            <!--                  using vlan in expression let the software  -->
            <!--                  use a four bytes offset                    -->
            <!--                  BUT for mixed tagged and untagged traffic  -->
            <!--                  the expression "vlan or udp or tcp" could  -->
            <!--                  fail to work on some systems/NICs          -->

<!ELEMENT PDUSending       (SendTo+) >
<!ATTLIST PDUSending       probeIP      NMTOKEN                    "0.0.0.0"
                           probePort    NMTOKEN                    #REQUIRED
>
            <!-- probeIP:         Source IP for sending PDUs                -->
            <!--                     0.0.0.0 means ANY SOURCE ADDR          -->
            <!-- probePort:       Source port for sending PDUS              -->
<!ELEMENT SendTo           EMPTY >
<!ATTLIST SendTo           collector    IDREF                      #REQUIRED
                           maxPDUSize   NMTOKEN                    "300"
                           maxPDUBuildTime
                                        NMTOKEN                    "300"
                           minTimeGap   NMTOKEN                    "100"
                           heartBeatDelay
                                        NMTOKEN                    "10"
                           timeout      NMTOKEN                    "10"
                           TOMult       NMTOKEN                    "1"
                           TOIncr       NMTOKEN                    "0"
                           retries      NMTOKEN                    "2"
                           breakBorderTime
                                        NMTOKEN                    "3"
                           dropObsFinalTimeout
                                        NMTOKEN                    "60"
>
            <!-- collector:       Name of target Collector                  -->
            <!-- maxPDUSize:      maximum UDP payload, in bytes, of a PDU   -->
            <!--                     valid values are [200..1460]           -->
            <!-- maxPDUBuildTime: maximum duration, in seconds, that a PDU  -->
            <!--                  under construction waits for new records  -->
            <!--                  before being sent                         -->
            <!--                     0 means to send each record in its own -->
            <!--                       PDU, as soon as published            -->
            <!--                     valid values are [0..65535]            -->
            <!-- minTimeGap:      least gap, in msec, between two           -->
            <!--                                              successive PDU -->
            <!--                     valid values are [0..10000]           -->
            <!-- heartBeatDelay:  max silence delay, in sec, before         -->
            <!--                  publishing a possibly empty PDU to server -->
            <!--                     valid values are [1..600]              -->
            <!-- timeout:         first PDU ack timeout in sec              -->
            <!--                     valid values are [1..120]              -->
            <!-- TOMult:          how many times previous ack timeout at    -->
            <!--                     next retry?                            -->
            <!--                     valid values are [1..10]               -->
            <!-- TOIncr:          secs to add to                            -->
            <!--                           (previous ack timeout * TOMult)  -->
            <!--                     valid values are [0..120]              -->
            <!--                                                            -->
            <!-- FOR TYPES OF OBSERVATIONS SUBJECT TO BE DISCARDED:         -->
            <!-- retries:         (maximum PDU send tries) minus 1          -->
            <!--                    ALSO threshold for detecting loss of    -->
```

```
            <!--                                probe connectivity condition  -->
            <!--                          valid values are [0..100]           -->
            <!-- breakBorderTime: time window, in sec, from detection       -->
            <!--                        of probe loss of connectivity, whose  -->
            <!--                        packet observations PDU's are         -->
            <!--                        continuously retried                  -->
            <!--                          valid values are [0..300]           -->
            <!-- dropObsFinalTimeout: probe will anyway discard its         -->
            <!--                           long retried obs after this quite  -->
            <!--                           long delay in MINUTES              -->
            <!--                          valid values are [1..12000]         -->
            <!--                                                 up to 1 week  -->
            <!-- NOTE: retries and breakBorderTime do NOT apply to          -->
            <!--               - Flow Instance Description PDU type          -->
            <!--       which are always retried until PDU contents expire    -->

<!ELEMENT PDUSaving        EMPTY >
<!ATTLIST PDUSaving        filepathname CDATA                    #REQUIRED
                           maxPDUSize   NMTOKEN                  "3000"
>
            <!-- filepathname:     Full pathname of file radix where to     -->
            <!--                   locally save the various types of probe  -->
            <!--                   observation PDU's                        -->
            <!--                      supports strftime strings (%H%M...)   -->
<!-- FACTS-EO FlowClass covers a new TrafMon behaviour where the traffic   -->
<!-- can be independently measured at the different probe/interfaces, and  -->
<!-- the measurements are selectively aggregated and centralised, according-->
<!-- to  Measure directives associated to different FlowClasses.           -->
<!--                                                                       -->
<!-- A FlowClass is assigned one or more Filter expressions to designate to-->
<!-- which sets of packets and protocol exchanges its directives pertain.  -->
<!-- A given Filter Expression is applied On one or more probe/interfaces  -->
<!--                                                                       -->
<!-- Measurements for a FlowClass are segregated per GranularFlow, before  -->
<!-- being centralised to a Server.                                        -->
<!-- The Criteria to discover and discriminate the Granular Flows can be   -->
<!-- applied via <FlowGrain> at the level of a FlowClass, or specialised   -->
<!-- for specific Filter Expression.                                       -->
<!--                                                                       -->
<!-- A given packet can well match several FlowClasses. Hence different    -->
<!-- sets of measurement directives can apply to a same packet and protocol-->
<!-- exchange.                                                             -->

<!ELEMENT GranularFlow     (DistinctIf?,DistinctAddr?,DistinctPort?,GroupBy*) >
<!ATTLIST GranularFlow     name         ID                       #REQUIRED
>
<!ELEMENT DistinctIf       EMPTY >
            <!-- Packets seen at different Probe Interfaces lead to        -->
            <!-- instances of granular flow, even when they produce same   -->
            <!-- results for all other criteria.                           -->
            <!-- NOTE:                                                     -->
            <!--      This may NOT be used when matching BI-DIRECTIONAL    -->
            <!--      traffic flow on the basis of packets captured by a   -->
            <!--      PASSIVE TAP devices: each direction being seen by a  -->
            <!--      separate capture interface.                          -->
<!ELEMENT DistinctAddr     EMPTY >
<!ATTLIST DistinctAddr     field        (src|srcnet|dst|dstnet|srcdst|srcdstnet
```

```
                                    |addr|net|addrpair|netpair) #REQUIRED
                        mask        CDATA                      #IMPLIED
>
            <!-- field:  which fields to preserve in grouping measurements  -->
            <!--  a) UNI-DIRECTIONAL                                        -->
            <!--   src:      keep granularity per source IP address         -->
            <!--   srcnet:   keep granularity per src IP subnet: using mask -->
            <!--   dst:      keep granularity per destination IP address    -->
            <!--   dstnet:   keep granularity per dst IP subnet: using mask -->
            <!--   srcdst:   keep granularity per source/dest. IP addresses -->
            <!--   srcdstnet:keep granularity per src/dst IP subnets: mask  -->
            <!--  b) BI-DIRECTIONAL                                         -->
            <!--   addr:     keep granularity per IP address of 1 peer      -->
            <!--   net:      keep gran. per IP subnet of 1 peer: using mask -->
            <!--   addrpair: keep granularity per pair of IP addresses      -->
            <!--   netpair:  keep granul. per pair of IP subnets: using mask-->
            <!--                                                            -->
            <!-- mask:   subnet mask: "xxx.xxx.xxx.xxx" or "/yy" notation   -->

<!ELEMENT DistinctPort    EMPTY >
<!ATTLIST DistinctPort    field       (sport|dport|sdport
                                       |port|portpair)           #REQUIRED
                          portspec    (alldistinct|privileged)  "alldistinct"
>
            <!-- field:  which fields to preserve in grouping measurements  -->
            <!--  a) UNI-DIRECTIONAL                                        -->
            <!--   sport:    keep granularity of source UDP/TCP port number -->
            <!--                    <=> any:port to any:any                 -->
            <!--   dport:    keep granularity of destin. UDP/TCP port number-->
            <!--                    <=> any:any to any:port                 -->
            <!--   sdport:   keep granularity of src/dst UDP/TCP prt numbers-->
            <!--                    <=> any:port1 to dst:port2              -->
            <!--    EITHER  without <DistinctAddr>                          -->
            <!--    OR ONLY with    <DistinctAddr field=(src|srcnet         -->
            <!--                                        and/or dst|dstnet) -->
            <!--   port: EITHER  with <DistinctAddr field=(src|srcnet) >    -->
            <!--             same as sport:  <=> src:sport to any:any       -->
            <!--         OR      with <DistinctAddr field=(dst|dstnet) >    -->
            <!--             same as dport:  <=> any:any to dst:dport       -->
            <!--         OR      with <DistinctAddr field=(srcdst[net]) >   -->
            <!--             preserves smallest port number:                -->
            <!--               if sport <= dport                            -->
            <!--                  <=> src:sport to dst:any                  -->
            <!--               if sport >  dport                            -->
            <!--                  <=> src:any to dst:dport                  -->
            <!--   portpair: ONLY with    <DistinctAddr field=(src|srcnet   -->
            <!--                                        and/or dst|dstnet) -->
            <!--             same as sdport: <=> src:sport to dst:dport     -->
            <!--  OTHER COMBINATIONS of <DistinctAddr>+<DistinctPort>       -->
            <!--                    ARE NOT ALLOWED (and meaningless)       -->
            <!--  b) BI-DIRECTIONAL                                         -->
            <!--   port: EITHER without <DistinctAddr>                      -->
            <!--             keep granul. of UDP/TCP port number of 1 peer  -->
            <!--               if sport <= dport                            -->
            <!--                  <=> any:sport to any:any                  -->
            <!--               if sport >  dport                            -->
            <!--                  <=> any:dport to any:any                  -->
```

```
            <!--         OR with <DistinctAddr field=(addr|net) >          -->
            <!--             keep granul.  addr:port  to/from   any:any    -->
            <!--                           net:port  to/from   any:any    -->
            <!--                              WHERE addr/net <= peer any  -->
            <!--         OR with <DistinctAddr field=(addrpair|netpair) >  -->
            <!--             keep granul. of address pair and smallest port:-->
            <!--               if port1 <= port2                           -->
            <!--                         addr1:port1 to/from addr2:any     -->
            <!--                          net1:port1 to/from  net2:any     -->
            <!--               if port1  > port2                           -->
            <!--                         addr1:any   to/from addr2:port2   -->
            <!--                          net1:any   to/from  net2:port2   -->
            <!--   portpair:                                               -->
            <!--         EITHER without <DistinctAddr>                     -->
            <!--             keep granul. of both UDP/TCP port numbers     -->
            <!--                   <=> any:port1 to/from any:port2         -->
            <!--                              WHERE port1 <= port2         -->
            <!--         OR with <DistinctAddr field=(addr|net) >          -->
            <!--             keep granul.  addr:port1 to/from   any:port2  -->
            <!--                           net:port1 to/from   any:port2   -->
            <!--                              WHERE addr/net <= peer any   -->
            <!--         OR with <DistinctAddr field=(addrpair|netpair) >  -->
            <!--             keep granul. addr1:port1 to/from addr2:port2  -->
            <!--                           net1:port1 to/from  net2:port2  -->
            <!--                                                           -->
            <!-- portspec:                                                 -->
            <!--  *alldistinct:  keep all values distinct                  -->
            <!--   privileged:  distinguish all service ports<1024        -->
            <!--                 BUT group all ports>=1024 (as 65535)      -->

<!ELEMENT GroupBy          EMPTY >
<!ATTLIST GroupBy          field        (ipsizes
                                        |ipproto|tos|df|mf|frag|ttl
                                        |icmp
                                        |tcptype)                  #REQUIRED
                           sizeclasses  (per200|per400)            #IMPLIED
                           tosspec      (precedence|dscp|tosbyte)  #IMPLIED
                           fragspec     (fragnumber|fragoffset)    #IMPLIED
                           icmpspec     (icmpclass|icmptype
                                        |icmptypecode)             #IMPLIED
                           tcptypespec  (byflags|byflagsandretran
                                        |S_D_A_E|S_D_A_E_R
                                        |S_F_R_A_E|S_F_R_A_E_R)    "S_D_A_E"
>
            <!--   ipsizes:  keep granularity per 'sizeclasses' of IP pkt   -->
            <!--   ipproto:  keep granularity per UDP|TCP|Other IP protocol -->
            <!--   tos:      keep granularity as per IP TypeOfSvc 'tosspec' -->
            <!--   df:       keep granularity per IP Don't Fragment flag    -->
            <!--   mf:       keep granularity per IP More Fragment flag     -->
            <!--   frag:     keep granularity as per IP Fragment 'fragspec' -->
            <!--   ttl:      keep granularity per IP Time-to-Live value     -->
            <!--   icmp:     keep granularity of ICMP pkts as per 'icmpspec'-->
            <!--   tcptype:  keep granularity as per 'tcptypespec' grouping -->
            <!--                                                           -->
            <!-- sizeclasses: groups IP packet sizes in buckets            -->
            <!--   per400:   4 buckets: boundaries 400, 800, 1200          -->
            <!--   per200:   8 buckets:     200,400,600,800,1000,1200,1400 -->
```

```
            <!-- BUT, for datagram cummulated IP sizes, sizes >= 1600       -->
            <!-- are grouped by thousands: 1600, 2000, 3000 ... 7000, 8000  -->
            <!--                                                            -->
            <!-- tosspec:                                                   -->
            <!--   precedence: per value of the three ToS precedence bits   -->
            <!--   dscp:       per value of the six DSCP bits               -->
            <!--   tosbyte:    per distinct values of the complete ToS byte -->
            <!--                                                            -->
            <!-- fragspec:                                                  -->
            <!--   fragnumber: per ordinal number of the fragment           -->
            <!--   fragoffset: per value of the fragment offset             -->
            <!--                                                            -->
            <!-- icmpspec:                                                  -->
            <!--   icmpclass:    group as per Echo | Error | Info | Other   -->
            <!--   icmptype:     per value of the ICMP Type byte            -->
            <!--   icmptypecode: per value of the ICMP Type and Code bytes  -->
            <!--                                                            -->
            <!-- tcptypespec: distinguish                                   -->
            <!--   byflags:      per distinct values of the TCP flags byte  -->
            <!--   byflagsandretran: idem, but also distinguish between     -->
            <!--                 first and subsequent transmissions         -->
            <!--                 of a not empty data segment                -->
            <!--  *S_D_A_E:      Start (syn/syn-ack),                       -->
            <!--                 Data (not empty payload)                   -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack/reset)                    -->
            <!--   S_D_A_E_R:    Start (syn/syn-ack),                       -->
            <!--                 Data (not empty payload)                   -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack)                          -->
            <!--                 RESET (rst flag)                           -->
            <!--   S_F_R_A_E:    Start (syn/syn-ack),                       -->
            <!--                 FIRST transmission of data segment         -->
            <!--                 RETRANSMISSION of data segment             -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack/reset)                    -->
            <!--   S_F_R_A_E_R:  Start (syn/syn-ack),                       -->
            <!--                 FIRST transmission of data segment         -->
            <!--                 RETRANSMISSION of data segment             -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack)                          -->
            <!--                 RESET (rst flag)                           -->

<!ELEMENT FlowClass       (Measure, FlowGrain?, Filter+, Condition?)>
<!ATTLIST FlowClass       id          NMTOKEN                  #REQUIRED
                          name        ID                       #REQUIRED
                          descr       CDATA                    #IMPLIED
>

<!ELEMENT Measure         (Delay?, Stats?) >
<!ATTLIST Measure         interval    (each|10s|20s|30s
                                       |1min|10min|20min|30min
                                       |1h)                    #REQUIRED
>
            <!--   Maximum 1 <Delay>, maximum 1 <Stats>, but not empty      -->

<!ELEMENT Delay           ((OneWayDelay|RoundTripDelay|InterPacket),Histogram?)>
```

```
<!ATTLIST Delay           for          (firstFragment|allFragments
                                        |datagram)               #REQUIRED
                          granularity  (individual|collectorAggregated
                                        |probeAggregated)        #REQUIRED
>
<!-- How delay measurements are processed:                                  -->
<!-- ====================================                                  -->
<!--   Either individual measurements are loaded in the database, or these  -->
<!--   are pre-aggregated into histogram slices, either by the collector or -->
<!--   where applicable, by the probe itself, before transmission.          -->
<!--                                                                        -->
<!--             granularity: Where individual measurements are             -->
<!--                          aggregated:                                   -->
<!--               individual    Individual measured values are delivered   -->
<!--                             by the probe to the collector and from the -->
<!--                             collector to the database.                 -->
<!--                             So actual aggregation occurs inside the    -->
<!--                             database itself.                           -->
<!--               collectorAggregated                                      -->
<!--                             Individual measured values are delivered   -->
<!--                             by the probe to the collector and these are-->
<!--                             aggregated inside the collector which      -->
<!--                             supplies the database with short duration  -->
<!--                             histogram slices                           -->
<!--               probeAggregated                                          -->
<!--                             Individual measured values are aggregated  -->
<!--                             inside the probe which transmits resulting -->
<!--                             short duration histogram slices to the     -->
<!--                             collector, in turn supplying them to the   -->
<!--                             database                                   -->
<!--                ==> probeAggregated NOT VALID for Delay type="oneway"   -->
<!--                ==> probeAggregated IGNORED for Measure interval="each" -->
<!--                                                                        -->
<!--             interval:    Length of the histogram slice aggregating     -->
<!--                          the measurements                              -->
<!--               each          Used when data are not aggregated          -->
<!--                             (granularity=='individual'), meaning that  -->
<!--                             any new value has to be transmitted as soon-->
<!--                             as computed and individually supplied by   -->
<!--                             the collector to the database.             -->
<!--               (duration)    Over which duration measurements have to be-->
<!--                             pre-aggregated (in the probe or collector) -->

<!-- Measuring One-Way Latencies:                                           -->
<!-- ===========================                                           -->
<!--   Means that probes sends individual timestamps for data units to the  -->
<!--   central collector(s)                                                 -->
<!--                                                                        -->
<!--     ==> Potentially high volume of observations need to be centralised -->
<!--         in the collector                                               -->
<!--     ==> Implies granularity=individual or collectorAggregated          -->
<!--                                                                        -->
<!--             for:       Which data unit to measure?                     -->
<!--               firstFragment One capture timestamp for single           -->
<!--                             or first IP fragment of a datagram/segment -->
<!--                             ==> IP reassembly is not required for this -->
<!--                                 (second and subsequent fragments are   -->
```

```
<!--                                                          ignored for this) -->
<!--               allFragments  One capture timestamp for every fragment   -->
<!--                             ==> only meaningful when NO fragmentation  -->
<!--                                       between concerned probing points -->
<!--                             ==> IP reassembly is required for          -->
<!--                                      FlowClass membeship determination -->
<!--                 + APPLIES ALSO   to subseq. fragments not individually -->
<!--                                  matching Filter                       -->
<!--                 + DOES NOT APPLY to subseq. fragments explicitly       -->
<!--                                  rejected by StatePred Condition       -->
<!--                                                                        -->
<!--               datagram      One capture timestamp for unfragmented pkt -->
<!--                             But two capture timestamps for chains of   -->
<!--                             datagram fragments: {first seen, last seen}-->
<!--                             ==> IP reassembly is required for this     -->
<!--                             ==> Latency                                -->
<!--                                   = lastTS(dst side) -firstTS(src side)-->
<!--                                                                        -->
<!ELEMENT OneWayDelay      (Hop+, Sign?) >

<!ATTLIST OneWayDelay      from          NMTOKEN                  #IMPLIED
                           to            NMTOKEN                  #IMPLIED
                           lost          (each|count)             "each"
>
            <!-- Producing One-Way Latency:                                 -->
            <!-- ==========================                                -->
            <!-- When Delay granularity=collectorAggregated                 -->
            <!-- produce the delay between the 'from' Hop and the 'to' Hop  -->
            <!--                                                           -->
            <!-- When Delay granularity=individual                          -->
            <!-- 'from' and 'to' are disregarded                            -->
            <!-- Individual HOP timestamps records are produced             -->
            <!--                                                           -->
            <!-- Note that Delay granularity=probeAggregated IS NOT VALID   -->
            <!--                                                           -->
            <!-- lost "each" : produce individual pkt/datagram lost records -->
            <!--      "count": produce count of lost data units per given   -->
            <!--               Measure interval unit                       -->
<!ELEMENT Hop EMPTY >
<!ATTLIST Hop              name          NMTOKEN                  #REQUIRED
                           descr         CDATA                    #IMPLIED
>
            <!-- Measuring One-Way Latencies:   hopName per timestamp       -->
            <!-- ============================   ====================       -->
            <!-- NOTE: <Hop> inside a <OneWayDelay> tag are ordered!!       -->
            <!-- The order represents the chronology of each timestamp in   -->
            <!-- the sequence, whatever its type (capture/IP/NTP)           -->
            <!--                                                           -->
            <!-- This tag must be in one or more copies in order to         -->
            <!-- specify the sequence of hop timestamps expected for        -->
            <!-- completing a valid record.                                 -->
            <!-- More than one probe (interface) can provide the same time  -->
            <!-- in the case of redundant transmission via more than one    -->
            <!-- link. But in any way, identical timestamp IDs are          -->
            <!-- expected to identify the same logical hop along the path.  -->
            <!-- If a probing point is labeled 'dmz', two different probes  -->
            <!-- reporting times for 'dmz' are supposed to be located in    -->
```

```
            <!-- alternate instances of the corporate DMZ.                    -->
            <!-- Every Hop name is assigned inside one or more <Filter>:      -->
            <!--  + either as the capture timestamp at the interface          -->
            <!--  + or as an IP option timestamp of a given sequence number   -->
            <!--  + or as an NTP timsetamp of a given type from either the    -->
            <!--     NTP Request or NTP Response                              -->

<!ELEMENT Sign     (Mask*, Chunk*) >
<!ELEMENT Mask EMPTY >
<!ATTLIST Mask      field              (srcAddr|dstAddr|srcPort|dstPort
                                       |ipFragData|ipTOS|ipID|ipOpts
                                       |transportCkSum)          #REQUIRED
>
            <!-- There can be multiple fields being masked.                   -->
            <!--                                                              -->
            <!-- field:       which part(s), if any, of the packet to         -->
            <!--              zeroize before signature hash computation.      -->
            <!--                                                              -->
            <!--              Masking any of srcAddr, dstAddr, srcPort,       -->
            <!--              dstPort also implies masking transportCkSum     -->
            <!--                                                              -->
            <!-- Mask directive can be combined with Chunk directives:        -->
            <!--              Specified fields laying into selected Chunks    -->
            <!--              do not participate to the signature hashing     -->
            <!-- Absence of Mask directive leads to masking only the          -->
            <!-- systematically (TTL, cksum) - or supposedly (ipOpts) -       -->
            <!-- varying fields of the IP header                              -->

<!ELEMENT Chunk EMPTY >
<!ATTLIST Chunk     start              NMTOKEN                    "0"
                    relTo              (ipHeader|ipPayload|tcpHeader|tcpPayload
                                       |udpHeader|udpPayload)     "ipHeader"
                    length             NMTOKEN                    "0"
>
            <!-- There can be one or multiple disjoint chunks of bytes        -->
            <!-- being concatenated then hashed to produce the signature      -->
            <!--                                                              -->
            <!-- By default (absence of Chunk directive), the entire packet   -->
            <!-- is subject to hashing, starting start the IP header          -->
            <!-- (masking at least systematically varying fields of IP hdr)   -->
            <!--                                                              -->
            <!-- When <Delay type="oneway" for="datagram"                     -->
            <!--                       or for="firstFragment"> is specified   -->
            <!--      a fragmented datagram is reassembled, and the IP        -->
            <!--      header, only with common stable fields, being appended  -->
            <!--      the reassembled IP payload is subject to signature,     -->
            <!--      according to specified Mask/Chunk directives.           -->
            <!--      ==> length==0 means to the end of reassembled payload   -->
            <!--                                                              -->
            <!-- start: offset form the given relTo base.                     -->
            <!--      ==0 by default                                          -->
            <!--      valid values [0..70000]  (max reass. dgram = 65535)     -->
            <!--                                                              -->
            <!-- relTo: when start==0 (or absent):                            -->
            <!--         ipHeader:   Starts at first byte of IP header        -->
            <!--                                                              -->
            <!--         ipPayload|tcpHeader|udpHeader:                       -->
```

```
            <!--                      Starts at first byte after IP header   -->
            <!--                      and IP options                         -->
            <!--                    VALID FOR ANY PROTOCOL ABOVE IP          -->
            <!--                    VALID FOR ANY IP FRAGMENT INDIV. MEASURED-->
            <!--                                                             -->
            <!--         tcpPayload: Starts at first byte after TCP header   -->
            <!--                      and TCP options                        -->
            <!--                    IGNORED FOR NON-TCP PACKETS              -->
            <!--                    IGNORED FOR NON-FIRST IP FRAGMENTS       -->
            <!--                                       INDIVIDUALLY MEASURED -->
            <!--                                                             -->
            <!--         udpPayload: Starts at first byte after TCP header   -->
            <!--                    IGNORED FOR NON-UDP PACKETS              -->
            <!--                    IGNORED FOR NON-FIRST IP FRAGMENTS       -->
            <!--                                       INDIVIDUALLY MEASURED -->
            <!--                                                             -->
            <!-- length:      how many bytes, since the specified start of   -->
            <!--              the chunk, do participate to the pkt signature-->
            <!--              hash computation                               -->
            <!--      valid values [0..70000]  (max reass. dgram = 65535)    -->
            <!--              0 MEANS TO END OF CAPTURED DATA                -->
            <!--                 OPTIONAL, default to entire pkt             -->
            <!--                can be overriden on a per flow basis         -->
            <!--                                                             -->
            <!-- IMPORTANT NOTE:                                             -->
            <!-- Chunks will be concatenated in the order specified.         -->
            <!-- Overlapping chunks are concatenanted independently,         -->
            <!-- possibly leading to redundant data segments.                -->
            <!-- Nevertheless, this will give same signature in different    -->
            <!--  probes.                                                    -->

<!ELEMENT RoundTripDelay  EMPTY>
<!ATTLIST RoundTripDelay  protocol     (icmpEcho|udpNTP|udpSNMP|udpDNS
                                       |tcpSynAck|tcpOptRTTM
                                       |tcpDataAck)              #REQUIRED
                          with         (initiator|responder
                                       |both|unspecified)       "unspecified"
>
<!-- Measuring Round-trip Delays for specific protocol exchanges:           -->
<!-- ========================================================            -->
<!--   The probe itself can match corresponding pairs of data units,       -->
<!--   one per direction, and therefore compute the round-trip time        -->
<!--   between the probing point and one of the communication side.        -->
<!--                                                                       -->
<!--     ==> Measurements can be individual (from probe to collector to    -->
<!--         database), or aggregated in the probe, or individually        -->
<!--         forwarded and aggregated by the collector                     -->
<!--         As specified by <Delay> directive                             -->
<!--                                                                       -->
<!--             target:    Which data unit to measure?                    -->
<!--                          (Default "firstFragment")                    -->
<!--               firstFragment One capture timestamp for single          -->
<!--                             or first IP fragment of a datagram/segment -->
<!--                             ==> IP reassembly is not required for this -->
<!--                                 (second and subsequent fragments are   -->
<!--                                                   ignored for this) -->
<!--               allFragments  Meaningless in case of round-trip          -->
```

```
<!--               datagram      One capture timestamp for unfragmented pkt -->
<!--                             But two capture timestamps for chains of   -->
<!--                             datagram fragments: {first seen, last seen}-->
<!--                             ==> IP reassembly is required for this     -->
<!--                             ==> Round-Trip                             -->
<!--                                   = firstTS(ingress) -lastTS(egress)   -->
<!--             protocol:  -What upper layer protocol is covered by the    -->
<!--                         <FlowClass> definition (<Filter> MUST BE       -->
<!--                         PROPERLY EXPRESSED TO MATCH THIS PROTOCOL)     -->
<!--                        -Hence specifies which upper layer protocol     -->
<!--                         analysis has to be applied on <FlowClass>      -->
<!--                         matching packets.                              -->
<!--               TCP connections:                                         -->
<!--                         Identifies which further peer packets matching -->
<!--                         is required.                                   -->
<!--                  Syn/Syn+Ack: 'with' is implied, its given value is not-->
<!--                         considered.                                    -->
<!--               udpSNMP or                                               -->
<!--               udpDNS:   Round-trip between the probing point and the   -->
<!--                         SNMP Agent or DNS Server                       -->
<!--               udpNTP:  -Round-trip between the probing point and the   -->
<!--                         NTP Server, but query/reply Transmit Time      -->
<!--                         permit to know latency between probe and each  -->
<!--                         end (when these have there time precisely      -->
<!--                         synchronised); hence:                          -->
<!--                        -ALSO deduced Round-Trip between NTP client and -->
<!--                         NTP server:                                    -->
<!--                             ==> Round-Trip(client, server)             -->
<!--                                   = RTT(probe, server)                 -->
<!--                                             + 2 * Delay(client, probe)-->

<!ELEMENT InterPacket     EMPTY >

<!-- Measuring Inter-Packet (inter-data-unit) Delays:                       -->
<!-- ================================================                       -->
<!--   Means that the probe itself can compute the delay between capture    -->
<!--   timestamps of individual packets or reassembled datagram units       -->
<!--                                                                        -->
<!--     ==> Measurements can be individual (from probe to collector to     -->
<!--         database), or aggregated in the probe, or individually         -->
<!--         forwarded and aggregated by the collector                      -->
<!--         As specified by <Delay> directive                              -->
<!--                                                                        -->
<!--             target:    Which successive data unit to measure?          -->
<!--               firstFragment Compare TS of single                       -->
<!--                             or first IP fragment of a datagram/segment -->
<!--                             ==> IP reassembly is not required for this -->
<!--                                 (second and subsequent fragments are   -->
<!--                                                   ignored for this) -->
<!--               allFragments  Compare every fragment with its successor  -->
<!--               datagram      One capture timestamp for unfragmented pkt -->
<!--                             But two capture timestamps for chains of   -->
<!--                             datagram fragments: {first seen, last seen}-->
<!--                             ==> IP reassembly is required for this     -->
<!--                             ==> InterDatagram delay                    -->
<!--                                    = firstTS(next) - lastTS(previous) -->
```

```
<!ELEMENT Histogram        EMPTY >
<!ATTLIST Histogram        lowBound     NMTOKEN                    #REQUIRED
                           highBound    NMTOKEN                    #REQUIRED
                           sliceCount   NMTOKEN                    "12"
>

<!-- For any three types of Delay, EITHER "collectorAggregated" or          -->
<!-- "probeAggregated" Histogram MAY (optional) be specified to preserve    -->
<!-- several ranges of observed delay values:                               -->
<!--                                                                        -->
<!-- When Histogram is not specified, all values are aggregated over time   -->
<!-- period into a single unbound bucket (no histogram slices)              -->
<!--                                                                        -->
<!--               lowBound:  First slice covers all values <  lowBound     -->
<!--              highBound:  Last  slice covers all values >= highBound    -->
<!--             sliceCount:  In between first and last slices, there are   -->
<!--                          sliceCount-2 intervals of values, of equal    -->
<!--                          length                                        -->
<!--       Currently only used for Delay metrics, the values are            -->
<!--       signed integers coded in 32 bit. Specifying any bound            -->
<!--       as either INT32_MIN (0xffff or -2147483648)                      -->
<!--              or INT32_MAX (0x7fff or  2147483647)                      -->
<!--       means "unbound".                                                 -->

<!ELEMENT Stats            (PacketCounters?,TCPConnections?,FileTransfers?) >
<!ATTLIST Stats            verifChksum  (none|bestEffort|fullReassembly)
                                                                  #REQUIRED
>
<!ELEMENT PacketCounters   EMPTY >
<!ATTLIST PacketCounters   for          (firstFragment|allFragments
                                         |datagram)               #REQUIRED
>

<!ELEMENT TCPConnections   EMPTY >
<!ATTLIST TCPConnections   granularity  (each|groupedByCollector
                                         |groupedByProbe)         "each"
>

<!ELEMENT FileTransfers    EMPTY >
<!ATTLIST FileTransfers    protocol     (FTP|HTTP)                 #REQUIRED
                           granularity  (each|groupedByCollector
                                         |groupedByProbe)         "each"
                           ftpdata      (start-stop|full)          #IMPLIED
>
            <!-- Measuring FTP File Transfers                              -->
            <!-- ++++++++++++++++++++++++++++                              -->
            <!-- protocol="FTP" ==> ftpdata field must be specified        -->
            <!--                    to permit matching the associated FTP  -->
            <!--                    data connections that would not        -->
            <!--                    otherwise be analysed by other Flow    -->
            <!--                    Classes Filters.                       -->
            <!-- ftpdata: start-stop:                                      -->
            <!--                    Only look at SYN (or first seen) packet -->
            <!--                    and at FIN or RST packet of any TCP    -->
            <!--                    connection between the pair of IP      -->
            <!--                    addresses of an FTP Control connection -->
            <!-- ftpdata: full:                                            -->
```

```
            <!--                    Look at every packet  of any TCP        -->
            <!--                    connection between the pair of IP       -->
            <!--                    addresses of an FTP Control connection  -->
            <!-- RESTRICTION : Only works when client IP (PORT) or server   -->
            <!--               IP (PASV) stays the same between Control and -->
            <!--               Data connections.                            -->


<!ELEMENT FlowGrain        EMPTY >
<!ATTLIST FlowGrain        ref          IDREF                    #REQUIRED
>

<!ELEMENT Filter           (On+, CaptureTimeStamp?,
                                (IpOptTimestamp | NtpTimestamp)*, NatPat*,
                                                  PacketExpr, FlowGrain?)>

<!ELEMENT On               EMPTY >
<!ATTLIST On               probe        IDREF                    #REQUIRED
                           if           NMTOKEN                  #REQUIRED
>
            <!-- probe:        a reference to the name of the probe         -->
            <!--               to which the Packet Filter Expression applies-->
            <!-- if:           a reference to the probe interface name      -->
            <!--               to which the Packet Filter Expression applies-->

<!ELEMENT CaptureTimeStamp EMPTY >
<!ATTLIST CaptureTimeStamp
                           hopName      NMTOKEN                  #REQUIRED
>
<!ELEMENT IpOptTimestamp   EMPTY >
<!ATTLIST IpOptTimestamp   ipTSNum      NMTOKEN                  #REQUIRED
                           hopName      NMTOKEN                  #REQUIRED
>

<!ELEMENT NtpTimestamp     EMPTY >
<!ATTLIST NtpTimestamp     ntpTime      (originreq|recvdreq
                                        |xmitreq|originrsp
                                        |recvdrsp|xmitrsp)       #REQUIRED
                           hopName      NMTOKEN                  #REQUIRED
>
            <!-- Capture Timestamps on packet/datagram                     -->
            <!-- ==================================================       -->
            <!--   name of the hop timestamp at a probing point          -->
            <!--                                                           -->
            <!-- Optional Additional Timestamps on packet/datagram         -->
            <!-- ==================================================       -->
            <!--                                                           -->
            <!-- + based on IP OPTION TIMESTAMP (milliseconds since midnight-->
            <!--     The nth (ipTSNum) IP timestamp in the flow packets is  -->
            <!--     mapped to the given hopName                           -->
            <!--     1 <= ipTSNum <= 9                                     -->
            <!--                                                           -->
            <!-- + based on UDP NTP Embedded Timestamps                    -->
            <!--     Designates, respectively for a request or response,   -->
            <!--     which of the three relevant NTP Timestamps to report  -->
            <!--     as hopName                                            -->
            <!--     This requires firstFragment|datagram                  -->
```

```
<!ELEMENT NatPat           EMPTY>
<!ATTLIST NatPat           translate    (src|dst|addr
                                         |sport|dport|port)        #REQUIRED
                           from         CDATA                      #IMPLIED
                           into         CDATA                      #REQUIRED
>
            <!-- Address and/or Port Translation for Probing Point(s)       -->
            <!-- ===================================================       -->
            <!--                                                           -->
            <!-- Granular flows corresponding to this FlowClass and        -->
            <!--  discovered at the given probing point(s) (as per <On>    -->
            <!--  clause(s)> must translate their identification information-->
            <!--  transmitted to the collector according to the given      -->
            <!--  <NatPat> directive(s) for permitting common network-wide  -->
            <!--  determination of same flows independent on their local   -->
            <!--  translation on IP address(es) and UDP/TCP port number(s)  -->
            <!--                                                           -->
            <!-- translate:                                                -->
            <!--  src or dst: specifies that the source or destination     -->
            <!--              locally seen IP address is subject to        -->
            <!--              translation before being mentioned to the    -->
            <!--              collector.                                   -->
            <!--  addr:       specifies that both the source and destination-->
            <!--              locally seen IP addresses are subject to     -->
            <!--              translation before being mentioned to the    -->
            <!--              collector.                                   -->
            <!--              ==> only meaningful with explicit 'from' value-->
            <!--  sport or dport:                                          -->
            <!--              specifies that the source port number or     -->
            <!--              destination port number locally seen value has-->
            <!--              to be translated before being mentioned to the-->
            <!--              collector.                                   -->
            <!--  port:       specifies that both the source and destination-->
            <!--              locally seen port number values are subject to-->
            <!--              translation before being mentioned to the    -->
            <!--              collector.                                   -->
            <!--              ==> only meaningful with explicit 'from' value-->
            <!-- from   is OPTIONAL                                        -->
            <!--   absent:   replace the value of the field indicated by   -->
            <!--             'translate' whatever it is, with the value    -->
            <!--             provided by 'into'                            -->
            <!--             ==> only 'src', 'dst, 'sport' or dport' are   -->
            <!--                 meaningful here),                         -->
            <!--   present:  gives the value of the field(s) indicated by  -->
            <!--             'translate' which would be subject to be      -->
            <!--             replaced by that given by 'into'              -->
            <!-- into:  the replacing value for field(s) designated by     -->
            <!--        'translate'                                        -->


<!ELEMENT PacketExpr       (Predicate|AND|NotAND)>
<!ELEMENT AND              (Predicate|OR|NotOR)+>
<!ELEMENT NotAND           (Predicate|OR|NotOR)+>
<!ELEMENT OR               (Predicate|SubAND|SubNotAND)+>
<!ELEMENT NotOR            (Predicate|SubAND|SubNotAND)+>
```

```
<!ELEMENT SubAND          EMPTY>
<!ELEMENT SubNotAND       EMPTY>
            <!-- Packet Filter Expression for <FlowClass> match             -->
            <!-- ============================================             -->
            <!-- This expression has up to three layers of sub-expressions. -->
            <!-- Either Single-predicate:                                  -->
            <!--     <PacketExpr>                                          -->
            <!--         <Predicate field=... op=... value=.../>           -->
            <!--     </PacketExpr>                                         -->
            <!-- Or predicate AND predicate AND predicate ...              -->
            <!--      <PacketExpr>                                         -->
            <!--         <AND>                                             -->
            <!--            <Predicate field=... op=... value=.../>        -->
            <!--            <Predicate field=... op=... value=.../>        -->
            <!--            ...                                            -->
            <!--         </AND>                                            -->
            <!--     </PacketExpr>                                         -->
            <!-- Or NOT (predicate AND predicate AND predicate ...)        -->
            <!--      <PacketExpr>                                         -->
            <!--         <NotAND>                                          -->
            <!--            <Predicate field=... op=... value=.../>        -->
            <!--            <Predicate field=... op=... value=.../>        -->
            <!--            ...                                            -->
            <!--         </NotAND>                                         -->
            <!--     </PacketExpr>                                         -->
            <!--                                                           -->
            <!-- The list inside <AND> ... </AND> can also contain 1 or more-->
            <!-- alternatives of predicates, introducing a second level    -->
            <!-- with the <OR> ... </OR> or its contrary <NotOR> .. </NotOR>-->
            <!--                                                           -->
            <!-- For sake of completeness, elements joined by an <OR> or   -->
            <!-- <NotOR> connectives can be a mix of predicates and of     -->
            <!-- sub-expressions consisting only in predicates, assembled by-->
            <!-- logical AND connective (<SubAND> tag) or their contrary   -->
            <!-- (<SubNorAND> tag)                                         -->

<!ELEMENT Predicate       EMPTY>
<!ATTLIST Predicate       field       (src|dst|addr
                                       |ipsize|ttl|tosprec|tosdscp|tosbyte
                                       |df|mf|fragofst
                                       |proto
                                       |icmpTypeCode
                                       |sport|dport|port
                                       |syn|fin|ack|psh|urg)     #REQUIRED
                          op          (eq|ne|lt|le|gt|ge|betw|mask) #REQUIRED
                          value       CDATA                     #REQUIRED
                          value2      CDATA                     #IMPLIED
>
            <!-- src or dst: Predicate applies respectively to the source  -->
            <!--             or destination IP address field of the captured-->
            <!--             packet                                        -->
            <!-- addr:       Predicate applies any of source or destination -->
            <!--             IP address fields of the captured packet      -->
            <!--      The 'mask' operator permits to test subnet membership -->
            <!-- ipsize:     Length of the IP fragment, in bytes, including -->
            <!--             the 20 bytes of IPv4 header and the potential  -->
            <!--             IPv4 options data.                            -->
```

```
            <!-- ttl:        The IP Time-to-Leave byte value                 -->
            <!-- tosprec:    The numeric prority (precedence) given in the  -->
            <!--             three high-order bits of the Type-of-Service    -->
            <!--             byte.                                           -->
            <!-- tosdscp:    The numeric Differentiated Service Code Point   -->
            <!--             given in the six high-order bits of the IPv4    -->
            <!--             Type-of-Service byte.                           -->
            <!-- tosbyte:    The numeric value of the complete IPv4          -->
            <!--             Type-of-Service byte.                           -->
            <!--      Subject to 'mask' operator                             -->
            <!-- df:         The boolean "Don't Fragment" IPv4 flag bit      -->
            <!-- mf:         The boolean "More Fragment" IPv4 flag bit       -->
            <!-- fragofst:   The value of the IPv4 Fragment Offset,          -->
            <!--             expressed in 8-byte words                       -->
            <!--    Unfragmented pkt: (mf == 0) AND (fragofst == 0)          -->
            <!--    First fragment or unfragmented pkt: (fragofst == 0)      -->
            <!-- proto:      valid operators are "eq", "ne", "mask"          -->
            <!--             valid value are "esp", "ah", "udp", "tcp",      -->
            <!--             "icmp", "icmpEcho", "icmpFragNeed",             -->
            <!--             "icmpSrcQuench", "icmpTTLexpired",              -->
            <!--             "icmpReassemblyFailed", "icmpUnreachable",      -->
            <!--             "icmpOtherError", "icmpInfo", "icmpOther"       -->
            <!--    Note that upper layer protocols are not know at          -->
            <!--         <PacketExpr> level, applied on individual packet.   -->
            <!--         This <FlowFilter> is precisely used for determining-->
            <!--         <FlowClass> membership, whose <Measure> statements  -->
            <!--         permit to identify applicable upper layer protocol  -->
            <!--         analysis. Hence this filter permits the user to     -->
            <!--         specify the upper layer protocol, (typically based  -->
            <!--         TCP/UDP service 'port').                             -->
            <!-- icmpTypeCode is an alternative to 'proto'. It designates    -->
            <!--              the 2-byte ICMP Type and ICMP code of an ICMP  -->
            <!--              packet.                                        -->
            <!--      Subject to 'mask' operator                             -->

<!ELEMENT Condition       (StatePred|AND_list|NotAND_list|OR_list|NotOR_list)>
<!ELEMENT AND_list        (StatePred+)>
<!ELEMENT NotAND_list     (StatePred+)>
<!ELEMENT OR_list         (StatePred+)>
<!ELEMENT NotOR_list      (StatePred+)>

<!ELEMENT StatePred       EMPTY>
<!ATTLIST StatePred       state       (fragnum|fragcnt
                                       |dgramsize)              #REQUIRED
                          op          (eq|ne|lt|le|gt|ge|betw)  #REQUIRED
                          value       CDATA                     #REQUIRED
                          value2      CDATA                     #IMPLIED
>
                               <!-- Not yet implemented                    -->
                               <!--     |tcpwindow                         -->
                               <!--     |tcpretransmitted                  -->
                               <!--     |tcpdirtyopen                      -->
                               <!--     |tcpdirtyclose                     -->
                               <!--     |tcpemptyconn                      -->
                               <!--     |ftpemptysession                   -->
                               <!--     |ftpnofiletransfer                 -->
                               <!--     |ftpactivetransfer                 -->
```

```
                        <!--     |ftppassivetransfer)      #REQUIRED -->
              <!-- The <FlowClass> optional <Condition> permits to express    -->
              <!-- additional contraints on matching packets, based on result -->
              <!-- of statefull analysis (e.g. IP reassembly, or TCP          -->
              <!-- connection follow-on)                                      -->
              <!--                                                           -->
              <!-- Only a single-level of predicates can be expressed,       -->
              <!-- assembled by one of the following boolean connectives:    -->
              <!--  none:           single state predicate only              -->
              <!--  <AND_list>:     all state predicates must be true        -->
              <!--  <NotAND_list>:  at least one of the predicates is false  -->
              <!--  <OR_list>:      at least one of the predicates is true   -->
              <!--  <NotOR_list>:   all state predicates must be false       -->
              <!--                                                           -->
              <!-- fragnum:    IP Fragment number (first is 1)               -->
              <!-- fragcnt:    Number of IP fragment in datagram/segment     -->
              <!--             (==1 for single fragment)                     -->
              <!-- dgramsize:  How many bytes of UDP payload, TCP payload or -->
              <!--             otherwise IP payload, after IPv4 reassembly   -->
              <!--             (single frag. or reassembled datagram/segment) -->
              <!-- tcpwindow:  Size in bytes of the TCP window, after agreed -->
              <!--             TCP window scale option                       -->
              <!-- tcpretransmitted:                                         -->
              <!--             Whether (==true) or not (==false)             -->
              <!--             the TCP segment contains retransmitted payload -->
              <!--             data                                          -->
              <!-- tcpdirtyopen:                                             -->
              <!--             The corresponding TCP connection has been     -->
              <!--             initiated with one SYN or a SYN/SYN+ACK       -->
              <!--             but has not been continued (no complete 3-way -->
              <!--             SYN/SYN+ACK/ACK handshake seen during timeout -->
              <!--             period                                        -->
              <!-- tcpdirtyclose:                                            -->
              <!--             The corresponding TCP connection has been     -->
              <!--             closed, either, by RESET or single FIN packet -->
              <!--             without reverse FIN packet seen during timeout -->
              <!--             period                                        -->
              <!-- tcpemptyconn:                                             -->
              <!--             Not any payload data byte has been passed,    -->
              <!--             either way, over the connection               -->
              <!-- ftpemptysession:                                          -->
              <!--             Not any useful FTP command has been passed,   -->
              <!--             over the control connection                   -->
              <!-- ftpnofiletransfer:                                        -->
              <!--             The FTP control connection didn't initiate any -->
              <!--             actual file transfer (e.g. poll only)         -->
              <!-- ftpactivetransfer:                                        -->
              <!--             The FTP control connection or the resulting   -->
              <!--             FTP data connection involves active FTP mode  -->
              <!--             (i.e. PORT command)                           -->
              <!-- ftppassivetransfer:                                       -->
              <!--             The FTP control connection or the resulting   -->
              <!--             FTP data connection involves passive FTP mode -->
              <!--             (i.e. PASV command)                           -->
```

4.1.2 Example of XML Configuration File

The following XML file is used for trafMon Factory Qualification in the following test environment:

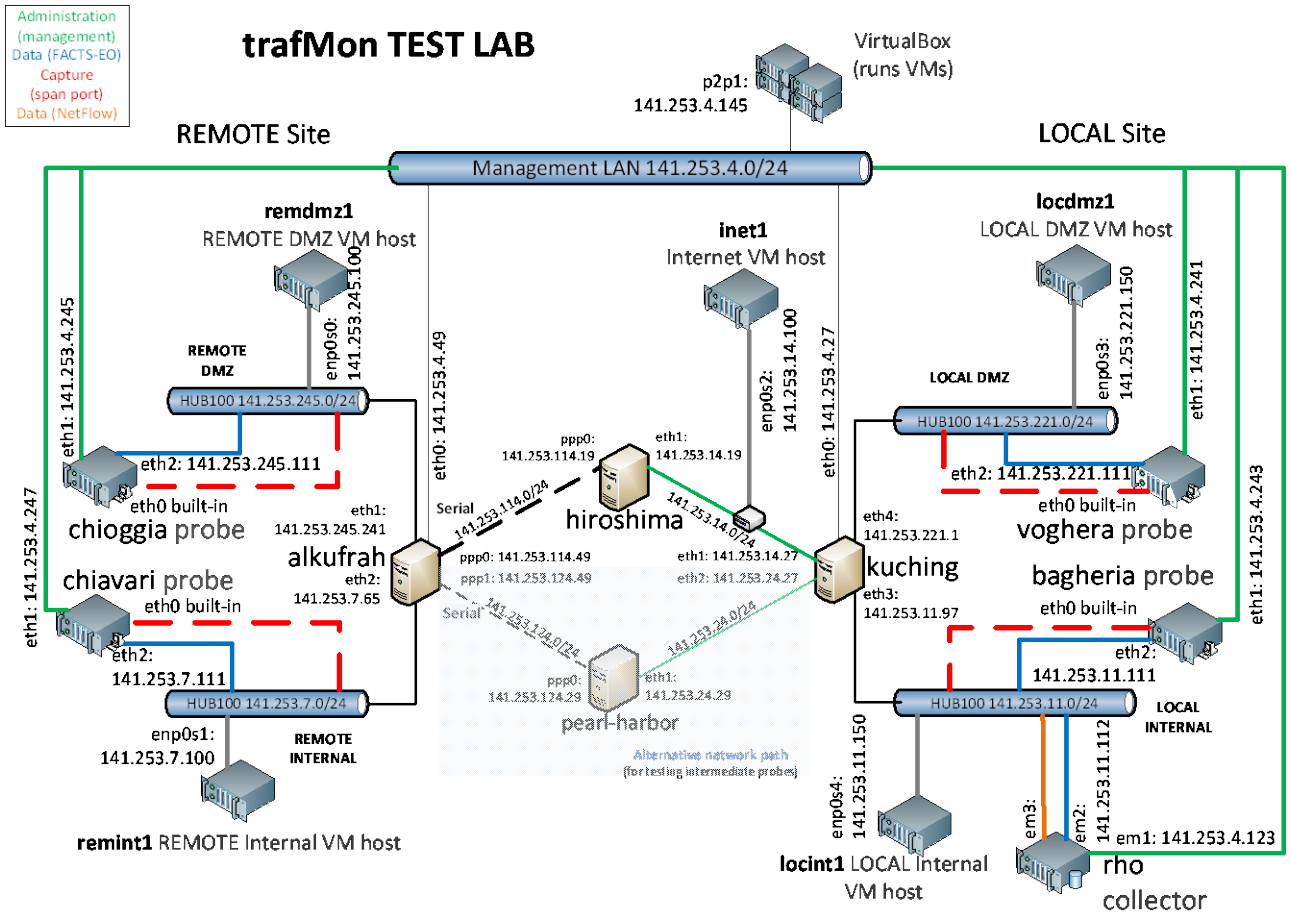


Figure 1: trafMon Factory Qualification Environment

```
<!-- trafMon configuration file for TEST LAB -->

<!DOCTYPE trafMonConfig SYSTEM "tmon.dtd" [

<!-- Data network -->
<!ENTITY data.rho   "141.253.11.112">
<!ENTITY data.bag   "141.253.11.111">
<!ENTITY data.vog   "141.253.221.111">
<!ENTITY data.chio  "141.253.245.111">
<!ENTITY data.chia  "141.253.7.111">
<!-- Traffic generators -->
<!ENTITY locdmz1    "141.253.221.150">
<!ENTITY locint1    "141.253.11.150">
<!ENTITY remdmz1    "141.253.245.100">
<!ENTITY remint1    "141.253.7.111">
<!ENTITY inet1      "141.253.14.100">
<!-- Service Port numbers -->
<!ENTITY dns    "53">
<!ENTITY ntp    "123">
```

```
<!ENTITY iperf "5001">
<!ENTITY http  "80">
<!ENTITY snmp  "161">
<!ENTITY ftp   "21">
<!ENTITY ftpdata "20">
<!-- PDU port (for sending PDUs from probes to TrafMon server) -->
<!ENTITY tmonProbe "9877">
<!ENTITY tmonColl  "9878">
<!-- Offsets used in SignBytes tags – Not used here -->
<!-- IP header offsets -->
<!ENTITY ipSrc "12">
<!ENTITY ipDst "16">
<!-- UDP header offsets -->
<!ENTITY udpSrc "0">
<!ENTITY udpDst "2">
<!-- NTP offsets -->
<!ENTITY ntpTT "40">
<!ENTITY ntpOT "24">
<!-- DNS offsets -->
<!ENTITY dnsID "0">
<!– Capture Filter fields – Not used here -->
<!ENTITY TOSBYTE "ip[1:1]">
<!ENTITY TOSPRI "ip[1:1]&amp;0xE0">
<!ENTITY TCPFLAGS "tcp[13:1]">
<!ENTITY TCPLEN "(ip[2:2] - (ip[0:1]&amp;0x0F)*4 - (tcp[12:1]&amp;0xF0)/4)">
<!– Capture Filters – Not used here -->
<!ENTITY ISTCPSYN "&TCPFLAGS; = 0x02">
<!ENTITY ISECHO "icmp and icmp[icmptype] = icmp-echo">
<!ENTITY ISECHOREPLY "icmp and icmp[icmptype] = icmp-echoreply">
<!ENTITY ISTOWEB "tcp and dst port &http;">
<!ENTITY ISFROMWEB "tcp and src port &http;">
<!ENTITY ISNTP "(udp and port &ntp;)">
<!ENTITY ISDNS "(udp and port &dns;)">
<!ENTITY ISDNSREQ "(&ISDNS; and udp[10] &amp; 0x80 = 0)">
<!ENTITY ISDNSRESP "(&ISDNS; and udp[10] &amp; 0x80 = 0x80)">
<!ENTITY ISSNMPREQ "(udp and dst port &snmp;)">
<!-- Temporary storage for CSV data produced by Central Processor -->
<!ENTITY dataPath "/var/trafMon/collector">
<!-- End of ENTITIES -->
]>

<trafMonConfig serial="2" startAt="2020-10-24 16:00:00" pktSignBytes="3"
  maxTravelTime="30000" >

  <Collector name="rho" ID="100" descr="trafMon collector on rho"
    burstRate="30">
    <Addr ip="&data.rho;" port="&tmonColl;" UDPBufferSize="20000"/>
    <Output dataFile="&dataPath;/observations.%y%m%d%H%M"
            eventFile="&dataPath;/events.%y%m%d%H%M"
            excepFile="&dataPath;/exceptions.%y%m%d%H%M"
            period="5"  />
  </Collector>

  <Probe name="voghera" ID="11" descr="trafMon probe on LOCAL DMZ">
    <Interface name="eth0" ID="111" descr="LOCAL DMZ" snapLen="210"
      bufPacketCount="1000000"
      expr= "ip"
```

```
      />
    <PDUSending probePort="&tmonProbe;">
      <SendTo collector="rho" maxPDUSize="1460" minTimeGap="0"
        maxPDUBuildTime="5" heartBeatDelay="90" timeout="20" retries="5"
        TOMult="1" TOIncr="5"  breakBorderTime="180" dropObsFinalTimeout="5" />
    </PDUSending>
  </Probe>

  <Probe name="bagheria" ID="12" descr="trafMon probe on LOCAL INTERNAL">
    <Interface name="eth0" ID="112" descr="LOCAL Internal LAN" snapLen="210"
      bufPacketCount="1000000"
      expr= "ip"
      />
    <PDUSending probePort="&tmonProbe;">
      <SendTo collector="rho" maxPDUSize="1460" minTimeGap="0"
        maxPDUBuildTime="5" heartBeatDelay="90" timeout="20" retries="5"
        TOMult="1" TOIncr="5"  breakBorderTime="180" dropObsFinalTimeout="5" />
    </PDUSending>
  </Probe>

  <Probe name="chioggia" ID="21" descr="trafMon probe on REMOTE DMZ">
    <Interface name="eth0" ID="221" descr="REMOTE DMZ" snapLen="210"
      bufPacketCount="1000000"
      expr= "ip"
      />
    <PDUSending probePort="&tmonProbe;">
      <SendTo collector="rho" maxPDUSize="500" minTimeGap="30"
        maxPDUBuildTime="5" heartBeatDelay="90" timeout="20" retries="6"
        TOMult="1" TOIncr="5"  breakBorderTime="100" dropObsFinalTimeout="5" />
    </PDUSending>
  </Probe>

  <Probe name="chiavari" ID="22" descr="trafMon probe on REMOTE INTERNAL">
    <Interface name="eth0" ID="222" descr=" REMOTE Internal LAN"
      snapLen="210"
      bufPacketCount="1000000"
      expr= "ip"
      />
    <PDUSending probePort="&tmonProbe;">
      <SendTo collector="rho" maxPDUSize="500" minTimeGap="30"
        maxPDUBuildTime="5" heartBeatDelay="90" timeout="20" retries="6"
        TOMult="1" TOIncr="5"  breakBorderTime="100" dropObsFinalTimeout="5" />
    </PDUSending>
  </Probe>

  <!-- ============================                                    -->

  <GranularFlow name="peers" > <!-- NO DistinctIf for oneWay partial obs -->
    <DistinctAddr field="addrpair" />
  </GranularFlow>

  <GranularFlow name="protoConversAtProbeIf" >
    <DistinctIf /> <!-- mandatory when Counters, to avoid double records -->
    <DistinctAddr field="addrpair" />
    <DistinctPort field="portpair" portspec="privileged" />
    <GroupBy      field="ipproto"/>
  </GranularFlow>
```

```
<GranularFlow name="uniDirAtProbeIf" >
  <DistinctIf /> <!-- mandatory when Counters, to avoid double records -->
  <DistinctAddr field="srcdst" />
  <DistinctPort field="portpair" portspec="privileged" />
  <GroupBy      field="ipproto"/>
</GranularFlow>

<GranularFlow name="perProtoServicePort" >
  <DistinctAddr field="srcdst" />
  <DistinctPort field="portpair" portspec="privileged" />
  <GroupBy      field="ipproto"/>
</GranularFlow>


<!-- FTP: TCP port 21
     ================
-->
<FlowClass id="21" name="FTP_port21" descr="TCP with port==21">
  <Measure interval="1min" >
    <Stats verifChksum="bestEffort">
      <PacketCounters for="firstFragment"/>
                        <!-- Don't ask for Dgram for TCP to avoid unnecessary
                             keeping of subsequent frags (of other flows)
                             between same IP address pair -->
      <TCPConnections granularity="each"/>
      <FileTransfers  protocol="FTP" granularity="each"
                      ftpdata="full"/>
    </Stats>
  </Measure>
  <FlowGrain ref="protoConversAtProbeIf" />
  <Filter>
    <On probe="voghera" if="eth0" />
    <On probe="bagheria" if="eth0" />
    <On probe="chioggia" if="eth0" />
    <On probe="chiavari" if="eth0" />
    <PacketExpr>
      <AND>
        <Predicate field="proto" op="eq" value="tcp"/>
        <Predicate field="port"  op="eq" value="21"/>
      </AND>
    </PacketExpr>
  </Filter>
</FlowClass>

<!-- HTTP: TCP port 80
     =================
-->
<FlowClass id="80" name="HTTP" descr="TCP with port==80">
  <Measure interval="1min" >
    <Stats verifChksum="bestEffort">
      <PacketCounters for="firstFragment"/>
                        <!-- Don't ask for Dgram for TCP to avoid unnecessary
                             keeping of subsequent frags (of other flows)
                             between same IP address pair -->
      <TCPConnections granularity="each"/>
    </Stats>
```

```
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="voghera" if="eth0" />
      <On probe="bagheria" if="eth0" />
      <On probe="chioggia" if="eth0" />
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="tcp"/>
          <Predicate field="port"  op="eq" value="80"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>

  <!-- ALL Unidirectional packets (for volumes counting)
       ================================================
  -->
  <FlowClass id="200" name="ALL_packets"
                                      descr="ALL Unidirectional IP Fragments">
    <Measure interval="1min" >
      <Stats verifChksum="bestEffort">
        <PacketCounters for="allFragments"/>
      </Stats>
    </Measure>
    <FlowGrain ref="uniDirAtProbeIf" />
    <Filter>
      <On probe="voghera" if="eth0" />
      <On probe="bagheria" if="eth0" />
      <On probe="chioggia" if="eth0" />
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
            <Predicate field="src" op="betw"
                                   value="0.0.0.1" value2="255.255.255.254" />
            <Predicate field="dst" op="betw"
                                   value="0.0.0.1" value2="255.255.255.254" />
                          </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>

  <!-- Bulk UDP Bidirectional (Fragmented) Datagrams
       ============================================
  -->
  <FlowClass id="210" name="two-way_Datagrams" descr="2-way UDP De-fragmented">
    <Measure interval="1min" >
      <Stats verifChksum="fullReassembly">
        <PacketCounters for="datagram"/>
      </Stats>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="voghera" if="eth0" />
      <On probe="bagheria" if="eth0" />
      <On probe="chioggia" if="eth0" />
```

```
    <On probe="chiavari" if="eth0" />
    <PacketExpr>
      <Predicate field="proto" op="eq" value="udp"/>
    </PacketExpr>
  </Filter>
</FlowClass>

<!-- Individual One-Way timestamps with IP Timestamps Option
     ======================================================
-->
<FlowClass id="12345" name="withIPTS"
  descr="UDP with port==21 and IP Timestamps">
  <Measure interval="each" >
    <Delay for="datagram" granularity="individual">
      <OneWayDelay>
        <Hop name="remint1"/>
        <Hop name="remint"/>
        <Hop name="alkuf"/>
        <Hop name="hiros"/>
        <Hop name="kuching"/>
        <Hop name="locdmz"/>
        <Sign>
          <Chunk start="0" relTo="ipPayload" length="40" />
        </Sign>
      </OneWayDelay>
    </Delay>
  </Measure>
  <FlowGrain ref="peers" />

  <Filter>
    <On probe="chiavari" if="eth0" />
    <CaptureTimeStamp           hopName="remint"/>
    <PacketExpr>
      <AND>
        <Predicate field="proto" op="eq" value="udp"/>
        <Predicate field="port"  op="eq" value="21"/>
      </AND>
    </PacketExpr>
  </Filter>

  <Filter>
    <On probe="voghera" if="eth0" />
    <CaptureTimeStamp           hopName="locdmz"/>
    <IpOptTimestamp ipTSNum="1" hopName="remint1"/>
    <IpOptTimestamp ipTSNum="2" hopName="alkuf"/>
    <IpOptTimestamp ipTSNum="3" hopName="hiros"/>
    <IpOptTimestamp ipTSNum="4" hopName="kuching"/>
    <PacketExpr>
      <AND>
        <Predicate field="proto" op="eq" value="udp"/>
        <Predicate field="port"  op="eq" value="21"/>
      </AND>
    </PacketExpr>
  </Filter>
</FlowClass>

  <!-- One-way Delay and Packet Loss on SNMP: LOCAL INT <> REMOTE DMZ
```

```
      ============================================================
      Request: SNMP from LOCAL INT to REMOTE DMZ
      ++++++++++++++++++++++++++++++++++++++++
-->
<FlowClass id="1111" name="One-Way_SNMP_Requests"
  descr="SNMP Requests one-way aggregated delay and loss/partial counters">
  <Measure interval="1min" >
    <Delay for="allFragments" granularity="collectorAggregated">
      <OneWayDelay from="locint_rq" to="remdmz_rq" lost="count" >
        <Hop name="locint_rq"/>
        <Hop name="remdmz_rq"/>
        <Sign/>
      </OneWayDelay>
      <Histogram lowBound="0" highBound="5000" sliceCount="7" />
    </Delay>
  </Measure>
  <FlowGrain ref="perProtoServicePort" />
  <Filter>
    <On probe="bagheria" if="eth0" />
    <CaptureTimeStamp           hopName="locint_rq"/>
    <PacketExpr>
      <AND>
        <Predicate field="proto" op="eq" value="udp"/>
        <Predicate field="dport" op="eq" value="161"/>
        <Predicate field="dst"   op="eq" value="&remdmz1;"/>
      </AND>
    </PacketExpr>
  </Filter>
  <Filter>
    <On probe="chioggia" if="eth0" />
    <CaptureTimeStamp           hopName="remdmz_rq"/>
    <PacketExpr>
      <AND>
        <Predicate field="proto" op="eq" value="udp"/>
        <Predicate field="dport" op="eq" value="161"/>
        <Predicate field="dst"   op="eq" value="&remdmz1;"/>
      </AND>
    </PacketExpr>
  </Filter>
</FlowClass>
<!-- Response: SNMP from REMOTE DMZ to LOCAL INT
     ++++++++++++++++++++++++++++++++++++++++
-->
<FlowClass id="2222" name="One-Way_SNMP_Responses"
  descr="SNMP Responses one-way aggregated delay and loss/partial counters">
  <Measure interval="1min" >
    <Delay for="allFragments" granularity="collectorAggregated">
      <OneWayDelay from="remdmz_rs" to="locint_rs" lost="count" >
        <Hop name="remdmz_rs"/>
        <Hop name="locint_rs"/>
        <Sign/>
      </OneWayDelay>
      <Histogram lowBound="0" highBound="5000" sliceCount="7" />
    </Delay>
  </Measure>
  <FlowGrain ref="perProtoServicePort" />
  <Filter>
```

```
      <On probe="chioggia" if="eth0" />
      <CaptureTimeStamp           hopName="remdmz_rs"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="sport" op="eq" value="161"/>
          <Predicate field="dst"   op="eq" value="&locint1;"/>
        </AND>
      </PacketExpr>
    </Filter>
    <Filter>
      <On probe="bagheria" if="eth0" />
      <CaptureTimeStamp           hopName="locint_rs"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="sport" op="eq" value="161"/>
          <Predicate field="dst"   op="eq" value="&locint1;"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>

  <!-- Round-Trip Delays
       ================
  -->
  <!--  Round trip delay measurement for ICMP
  -->
  <FlowClass id="3333" name="Echo-RoundTrip-histo"
    descr="ICMP Echo Req/Rsp aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="icmpEcho" />
        <Histogram lowBound="0" highBound="5000" sliceCount="5" />
      </Delay>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="bagheria" if="eth0" />
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="icmpEcho"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
  <!-- Round trip delay measurement for DNS
  -->
  <FlowClass id="4444" name="DNS-RoundTrip-histo"
    descr="DNS Req/Rsp aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="udpDNS" />
        <Histogram lowBound="0" highBound="800" sliceCount="4" />
      </Delay>
    </Measure>
```

```
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="53"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
  <!-- Round trip delay measurement for SNMP
  -->
  <FlowClass id="5555" name="SNMP-RoundTrip-histo"
    descr="SNMP Req/Rsp aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="udpSNMP" />
        <Histogram lowBound="0" highBound="800" sliceCount="4" />
      </Delay>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="bagheria" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="161"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
  <!-- Round trip delay measurement for NTP
  -->
  <FlowClass id="6666" name="NTP-RoundTrip-histo"
    descr="NTP Req/Rsp aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="udpNTP" with="both" />
        <Histogram lowBound="0" highBound="1000" sliceCount="5" />
      </Delay>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <!-- Real NTP from client remdmz1 to server locdmz1 -->
      <On probe="chioggia" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="addr" op="eq" value="&remdmz1;"/>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port" op="eq" value="123"/>
        </AND>
      </PacketExpr>
    </Filter>
    <Filter>
      <!-- Fake NTP from clients remint1/locint1 to server locdmz1 -->
      <On probe="bagheria" if="eth0" />
```

```
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="addr" op="eq" value="&locdmz1;"/>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port" op="eq" value="123"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
  <!-- Round trip delay measurement for TCP-SYN
  -->
  <FlowClass id="7777" name="TCP-SYN-RoundTrip-histo"
    descr="TCP SYN/SYN+ACK aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="tcpSynAck" />
        <Histogram lowBound="0" highBound="2000" sliceCount="6" />
      </Delay>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="bagheria" if="eth0" />
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="tcp"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
  <!-- Round trip delay measurement for TCP-RTTM
  -->
  <FlowClass id="8888" name="TCP-RTTM-RoundTrip-histo"
    descr="TCP RTTM Timestamps aggregated delay">
    <Measure interval="10s" >
      <Delay for="firstFragment" granularity="probeAggregated">
        <RoundTripDelay protocol="tcpOptRTTM" with="both" />
        <Histogram lowBound="0" highBound="300" sliceCount="8" />
      </Delay>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="bagheria" if="eth0" />
      <On probe="chiavari" if="eth0" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="tcp"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
</trafMonConfig>
```

4.2 TRAFMON DIAGNOSTIC LOGGING CONTROL INTERFACE

The diagnostic trace logging is provided by the common core module *tmon_diag.c*.

Each message generated from the C source code is assigned a severity level among (by decrease severity and increasing level of verbosity):

```
Config.      C code
fatal        FATAL -- the program print-out its message then aborts
error        ERR
warning      WARN
trace0       TR0
trace1       TR1
trace2       TR2
trace3       TR3
```

A configuration file permits to determine, for each software module (C source file radix name) part of the program, as well as per program:

- the maximum level of verbosity (minimal severity) will actually be logged
- the (list of) destination log pathname(s) per software module

Furthermore, this file establishes a maximum runtime level applicable to all modules (`Highest_level`).

By default, the diagnostic logging configuration is found in `/etc/trafMon/diag/`*name*`.diag`, where *name* is the name of either the probe or the collector instance (given as argument to the program):

File *tmon_probe.h*:

```
#define TM_PROBE_CONFIG_PATH   "/etc/trafMon"
#define TM_PROBE_DIAG_DIR      "diag"
#define TM_PROBE_DIAG_SUF      "diag"
```

File *tmon_collector.h*:

```
#define TM_COLL_CONFIG_PATH   "/etc/trafMon"
#define TM_COLL_DIAG_DIR      "diag"
#define TM_COLL_DIAG_SUF      "diag"
```

However, using the –l program option (local), the file can alternatively be taken from the current directory, as `./`*program_name*`.diag`, where *program_name* is either `tmon_probe` or `tmon_collector`.

The format of the config file is the same for all active lines except that with global threshold:

- either

```
Highest_level  trace0
```

- or

```
program module level logname1 logname2 ...
```

as in

```
tmon_probe tmp_statistics     trace2    /var/log/trafMon/probe.log
#tmon_probe tmp_tcpconnection  warning    /var/log/trafMon/probe.log
                                                    /var/log/trafMon/tcp.log
tmon_probe tmp_tcpconnection  trace2    /var/log/trafMon/probe.log
                                                    /var/log/trafMon/tcp.log
tmon_probe tmp_transmission   trace3    /var/log/trafMon/probe.log stderr
#tmon_probe tmp_transmission   trace2    /var/log/trafMon/probe.log stderr
tmon_probe tmp_udptransaction trace3    /var/log/trafMon/probe.log
```

Note that `stderr` and `stdout` are special keywords of output log file.

In the C code, each invocation of a logging function takes the form of **printf-like message** formatting argument list (with %m being the last errno string), pre-pended with the **severity keyword**, and appended with END (or **END;** for pleasing the debugger).

```
  TR1 "FTP Data Conn: %p %s %s%s%s user=%s", tcpConnp,
            TmTcpConnStr(&(search.tcpConnKey), tcpConnp->tcpConnState),
            (listOrFile==TMP_TCPUPPER_FTPLIST)?"LIST":"",
            (dataConnp->ftpDataXferDir==TMP_FTPFILE_GET)? "GET ":
              (dataConnp->ftpDataXferDir==TMP_FTPFILE_PUT)? "PUT ":"",
            (dataConnp->ftpDataFileName[0])? dataConnp->ftpDataFileName: "",
            (dataConnp->ftpCtlUser[0])? dataConnp->ftpCtlUser: "?"
            END;
```

The resulting log message is formatted as

```
DATETIME.MS,prog[PID],SEV,src_file:src_line:src_function,formatted mesage
```

as in

```
20201012T164520.346864,tmon_probe[14970],TR1,tmp_tcpconnection.c:2750:TmpF
tpDataConnMatch,FTP Data Conn: 0x1ec2f60 141.253.221.100:40274<-
>141.253.245.248:0 PUT transferred00.tar0.tar user=?
```

Note that when several C code statements are necessary to build-up the **trace level 2 or 3** message, but are useless if that message level is disabled, it is more efficient to surround this portion of code by

```
START_TR3
  TmonTimeSub(&(ackPktp->pktPcapHdr.ts), &(synPktp->pktPcapHdr.ts), &delta);
  TR3 "pkt 0x%016"PRIx64" TCP SYN Round-Trip %s - %s DELAY %f ms",
      synPktp->pktID, TmonTimeToStr(&(synPktp->pktPcapHdr.ts)),
      TmonTimeToStr(&(ackPktp->pktPcapHdr.ts)),
      (delta.tv_sec * 1000.0) + (delta.tv_usec / 1000.0) END;
END_TR3
```

Because those files are quickly growing, it is good to regularly proceed to a `logrotate`: `/etc/logrotate.d/trafmon`

```
/var/log/trafMon/*.log {
  # When some files have been mistakenly created as root,
```

```
  # this can perturbate the automated operations, so restore ownership
  firstaction
  chown -R trafmon:trafmon /var/log/trafMon
  endscript

  lastaction
  chown -R trafmon:trafmon /var/log/trafMon
  endscript

  rotate 300000
  # on CentOS 6.x, use daily insead of not yet suppoprted hourly
  hourly
  size 200M
  compress
  delaycompress
  missingok
  notifempty
  create 0644 trafmon trafmon
}
```

4.3 PROBE CAPTURE INTERFACE

The characteristics of a probe interface are described with following tags in the trafMon tool XML configuration file.

```
<!ELEMENT Probe           ((CapFile | Interface+), PDUSending*, PDUSaving?) >
<!ATTLIST Probe           name         ID                          #REQUIRED
                          ID           NMTOKEN                     #REQUIRED
                          descr        CDATA                       #IMPLIED
>
          <!-- ID:          [0..255]                                        -->

<!ELEMENT CapFile       EMPTY >
<!ATTLIST CapFile         filename     CDATA                       #REQUIRED
                          ID           NMTOKEN                     #REQUIRED
                          expr         CDATA                       #IMPLIED
                          rate         (withDelay|fullSpeed)       "withDelay"
>
          <!-- filename:   Full pathname of packet capture file to read   -->
          <!-- id:         TrafMon-wide unique numeric ID of the probe    -->
          <!--             interface that can distinguish among granular  -->
          <!--             flow instances                                 -->
          <!--                    valid values are [1..65535]             -->
          <!-- expr:       tcpdump-like packet capture filter expression  -->
          <!--                   WHEN VLAN PARTLY TAGS PRESENT            -->
          <!--                        INVOLVE vlan at end of expr         -->
          <!--                   MATCH only IP packets                    -->
          <!--                   DON'T use netmask based criteria         -->
          <!-- rate:       fullSpeed: captured pakets are processed       -->
          <!--                        untouched (with their original capt.-->
          <!--                        time) without waiting between each  -->
          <!--             withDelay: every packet has its capture time   -->
          <!--                        artificially translated by a fixed  -->
          <!--                        amount of time so as if the exact   -->
          <!--                        same traffic behaviour would occur  -->
          <!--                        'now'. The necessary variable delay -->
```

```
            <!--                        is respected before processing each -->
            <!--                        next packet                         -->

<!ELEMENT Interface        EMPTY >
<!ATTLIST Interface        name          NMTOKEN                    #REQUIRED
                           ID            NMTOKEN                    #REQUIRED
                           descr         CDATA                      #IMPLIED
                           snapLen       NMTOKEN                    "1600"
                           bufPacketCount
                                         NMTOKEN                    "70000"
                           expr          CDATA                      #IMPLIED
>
            <!-- id:         TrafMon-wide unique numeric ID of the probe    -->
            <!--             interface that can distinguish among granular  -->
            <!--             flow instances                                 -->
            <!--                     valid values are [1..65535]            -->
            <!-- snapLen:    maximum Ethernet frame Captured portion        -->
            <!--                     valid values are [125..65535]          -->
            <!--   On Linux:                                                -->
            <!--       snapLen=125 leads to 122 Eth capture => IP len=108   -->
            <!--                                                            -->
            <!--   NOTE:                                                    -->
            <!--       Even on Ethernet (max MTU = 1500 bytes IP),          -->
            <!--       larger packets can be actually captured due to       -->
            <!--       reassembly being offloaded in the NIC Card           -->
            <!--       Linux Ethtool -k:  LRO - Large Receive Offload or    -->
            <!--                          or GRO - Generic Receive Offload  -->
            <!--       ATTEMPT IS MADE TO DEACTIVATE THIS and the Reception -->
            <!--       Checksum processing offload upon initiatisation of   -->
            <!--       the probe capture interfaces                         -->
            <!--                                                            -->
            <!-- bufPacketCount:                                            -->
            <!--             How many packets (of ~ snapLen) could be       -->
            <!--             buffered upon traffic burst                    -->
            <!--                     valid values are [1000..1000000]       -->
            <!--                                                            -->
            <!-- expr:       tcpdump-like packet capture filter expression  -->
            <!--                    WHEN VLAN PARTLY TAGS PRESENT           -->
            <!--                         INVOLVE vlan at end of expr        -->
            <!--                    MATCH only IP packets                   -->
            <!--                    DON'T use netmask based criteria        -->
            <!--     LIMITATION: care must been taken with VLAN packets:    -->
            <!--                 using vlan in expression let the software  -->
            <!--                 use a four bytes offset                    -->
            <!--                 BUT for mixed tagged and untagged traffic  -->
            <!--                 the expression "vlan or udp or tcp" could  -->
            <!--                 fail to work on some systems/NICs          -->
```

Packet capture is achieved through the use of the public domain portable `libpcap` software module. Its API is generic and independent of the specifics of the particular operating system.

In particular, the configured snap length (maximum length of the captured frame that is actually buffered and passed to the application) is apparently aligned to a set of discrete values.

In Linux, libpcap exploits the system capability of reserving and initialising a potentially quite large ring memory buffer, resident in the kernel itself, and storing the successive captured packets that match the capture filter (also kernel resident).

Libpcap implementation does permit to specify the requested memory size of this per-interface memory reservation. But the API does not give access to the stored packet sequence. A callback routine of the application software is repetitively invoked with only one packet at a time. When this routine returns, the corresponding slot of the ring buffer is released for reused by a future capture.

For working at wire speed, the application should avoid to copy, in user space, the content of most of the analysed packets.

These limitations and performance constraints are the technical reasons why the selective permanent preservation of a chunk of packets, after they have been captured and analysed, upon request by the collector has not been implemented.

Note that the requested size for the ring buffer isn't necessarily allocated. First the size is rounded to a number of slots of maximum packet size capacity. Then the kernel reservation ioctl(SOL_PACKET, PACKET_RX_RING) is repeated, asking for only half the previous size at each retry, until it succeeds to reserve a reasonable amount of the available memory resource.

To test if the reservation does actually correspond to the configured `bufPacketCount`, one must launch the probe under strace (system call tracing), verifying that the first ioctl(PACKET_RX_RING) succeeds for each configured interface. Otherwise, the `bufPacketCount` values must be decreased and the strace attempt retried.

See Table 1 for some observed sizing values.

Modern kernels and modern NIC cards work in tandem: several functions, like checksum production/verification and IP reassembly, are by defaults done in the NIC card. And packet capture does occur in between the kernel protocol stack and the NIC.

Therefore, **the trafMon probe deactivates IP reassembly of incoming packets in every NIC port of its probing interfaces.**

Also the trafMon probe checksum verification may be fooled by the fact that the kernel is not assigning the UDP or TCP checksum value, relying in the NIC to the jobs (only after capture of locally generated packets).

For specific testing and diagnostics purposes, the probe can run by replaing a previously captured sequence of packets (commonly named a pcap file), produced by tcmpdump or Wireshark. There, the packets are handled as if they were occurring now. either they a replayed by respecting the original delay between successive packets, or they are processed at full speed, to measure the maximum analysis performance of the probe father and child processes on a given hardware configuration.

4.4 PROBE PDU TO COLLECTOR PROTOCOL

4.4.1 General Mechanism

See section 1.3.5 above.

4.4.2 Common PDU Header

```
/*
 * trafMon Absolute Time Reference as a Unix Timestamp
 */
#define TMON_PDU_ABSTIME_REF 1300000000L  /* Sun Mar 13 07:06:40 2011 UTC */
/*
 * Common PDU TrafMon PDU Header
 * ================================
 */
typedef struct tmon_pdu_hdr {
  uint16_t  pduLen;
  uint8_t   pduProbeID;
  uint8_t   pduCfgVers;
  uint32_t  pduTimeRef; /* Unix Abs. Timestamp - TMON_PDU_ABSTIME_REF */
  uint32_t  pduID;
  uint8_t   pduType;
  uint8_t   pduRecCnt;
  uint8_t   pduCRC[2];
} tmon_pdu_hdr_t;
```

4.4.3 Heart Beat PDU

This consists only in a PDU header with no data (pduRecCnt == 0)

pduType == TMON_PDU_TYPE_HRTBEAT 10

4.4.4 Flow Instance Description Records PDU

pduType == TMON_PDU_TYPE_FLDESCR 1

```
/*
 * I. FLOW INSTANCES DESCRIPTION PDU
 * ================================= */

/*
 * Possible values for flowGrainTcpSpec
 * ------------------------------------
 */
#define TMFGTCP_FLAGS  1 /* distinct per TCP Flags byte value */
#define TMFGTCP_FLRETR 2 /* dist. per TCP Flags byte and retransm. or not */
#define TMFGTCP_SDAE   3 /* distinct per TCP Start/Data/emptyAck/End */
#define TMFGTCP_SDAER  4 /* distinct per TCP Start/Data/emptyAck/End/RST */
#define TMFGTCP_SFRAE  5 /* distinct per TCP Start/First/Retr/emptyAck/End */
#define TMFGTCP_SFRAER 6 /* distinct per TCP Start/First/Retr/Ack/End/RST */
```

```
/*
 * Flow Instance description record for PDU
 * ----------------------------------------
 *
 *  Sorry: a coding trick has been used to code the five bits of potential
 *         network Mask length for ipAddr1/ipAddr2 by using unused
 *         nibble of the size bucket index and 1 unused bit of ICMP type/code
 *         (real-life ICMP non-experimental ICMP types are below 127)
 *
 *         This uggly coding is necessary to preserve the total
 *         sizeof(tmon_flow_descr_t) being a multiple of 8 bytes for
 *         alignment purpose, without gap, in the PDU: successive records
 *         form a clean array of tmon_flow_descr_t
 *
 *         Collector and output convention:
 *         --------------------------------
 *         The collector arranges for ensuring that 'ipAddr1 <= ipAddr2'
 *         for whatever direction is reflected: uni- or bi-directional.
 *         Where required it exchanges the description provided by a probe
 *         (swapping addr/port 1 and 2) and adapts the flow direction
 *         in useSpec.direct accordingly.
 *         This way, the address ordering invariant persists in the database
 *         and simplifies the grouping of site-to-site flows for a given
 *         pair of sites.
 */
typedef struct tmon_flow_descr {
  uint64_t  uniqId;
  uint32_t  ipAddr1;    /* 0 means no addressing */
  uint32_t  ipAddr2;    /* 0 means with any peer host */
  uint16_t  ifID;       /* interf. ID: 0 means don't care (NULL) */
  uint16_t  frag;       /* fragmentation info: see below */
  uint8_t   ipTTL;      /* 0 when not assigned */
  uint8_t   ipTOS;      /* either IP ToS byte, or IP Precedence or IP DSCP */
  uint16_t  tpPort1;    /* if Addr1: is UDP/TCP port on Addr1 host, 0 = none */
  uint16_t  tpPort2;    /* 0 means with any peer port */
  uint16_t  icmp;       /* low-order bits (0x7f)== ICMP type, type/code, class*/
                        /* high-order bit is high bit of net Mask */
  uint8_t   mask_lowSz;/* High nibble (0xf0): == low nibble of net Mask */
                        /* Low nibble (0x0f):  == IP sizes bucket index */
                        /*           lower bound as multiple of 200|400 */
  uint8_t   tcpType;
  uint16_t  useSpec;    /* Specification of used info fields: see below */

} tmon_flow_descr_t;

/*
 * Fragmentation Info:
 * -------------------
 *
 * On Big Endian host (network byte order), the uint16_t frag
 * is structured as follows:
 *
 *  uint16_t  ipDF:1;     * IP Don't Fragment flag
 *  uint16_t  ipMF:1;     * IP More Fragment flag
 *  uint16_t  frgNum:1;   * true: fragVal == fragNumber, else == fragOffset
 *  uint16_t  fragVal:13; * either fragNumber or fragOffset
 */
```

```
/* Network Byte Order Fragmentation Masks */
/*- - - - - - - - - - - - - - - - - - - - - - - - */
#define TMPDU_FLOW_FRGNUM 0x8000
#define TMPDU_FLOW_IPDF   0x4000
#define TMPDU_FLOW_IPMF   0x2000
#define TMPDU_FLOW_FRGVAL 0x1FFF
/*
 * Specification of used info fields: useSpec
 * ------------------------------------------
 *
 * On Big Endian host (network byte order), the uint16_t useSpec
 * is structured as follows:
 *
 *  uint8_t   direct:2;  * 1: from Addr1/Port1, 2: to Addr1/Port1, 3: both *
 *  uint8_t   ipProto:2; * 0: any,  1: UDP,         2 TCP,          3: other *
 *  uint8_t   tosSpec:2; * 0: none, 1: ToS byte,    2: precedence, 3: DSCP *
 *  uint8_t   icmpSpec:2;* 0: none, 1: type class, 2: ICMP Type, 3: Type+Code*
 *  uint8_t   useDF:1;   * is ipDF bit actually assigned ? *
 *  uint8_t   useMF:1;   * is ipMF bit actually assigned ? *
 *  uint8_t   useFrag:1; * are frgNum bit and fragVal actually assigned ? *
 *  uint8_t   tcpSpec:3; * 0: no tcpType, else: TMFGTCP_xxx value *
 *  uint8_t   szSpec:2;  * 0: none, 1: lowSize as multiple of 200, 2: of 400 *
 *                       * NOTE FOR DGRAM CUMMULATED IP SIZES:                *
 *                       *       when sizes < 1600, bucket size is 200|400    *
 *                       *       when sizes >= 1600 buckets end at thousands *
 *                       *       last bucket is >= 8000                       *
 */
/* Network Byte Order Use Masks */
/*- - - - - - - - - - - - - - - - */
#define TMPDU_FLOW_DIRECTN 0xC000
#define TMPDU_FLOW_FROM_1   0x4000
#define TMPDU_FLOW_TO_1     0x8000
#define TMPDU_FLOW_BIDIR    0xC000
#define TMPDU_FLOW_IPPROTO 0x3000
#define TMPDU_FLOW_UDP      0x1000
#define TMPDU_FLOW_TCP      0x2000
#define TMPDU_FLOW_OTHER    0x3000
#define TMPDU_FLOW_TOSSPEC 0x0C00
#define TMPDU_FLOW_TOSBYTE  0x0400
#define TMPDU_FLOW_TOSPREC  0x0800
#define TMPDU_FLOW_TOSDSCP  0x0C00
#define TMPDU_FLOW_ICMPSPC 0x0300
#define TMPDU_FLOW_ICMPCLS  0x0100 /* per ICMP class of type/code */
#define TMPDU_FLOW_ICMPTYP  0x0200 /* per ICMP Type byte */
#define TMPDU_FLOW_ICTYPCD  0x0300 /* per ICMP Type & Code */
#define TMPDU_FLOW_USEDF    0x0080
#define TMPDU_FLOW_USEMF    0x0040
#define TMPDU_FLOW_USEFRAG 0x0020
#define TMPDU_FLOW_TCPSPEC 0x001C  /* shifted TMFGTCP_xxx <<2 */
#define TMPDU_FLOW_SIZSPEC 0x0003
#define TMPDU_FLOW_SIZ200   0x0001
#define TMPDU_FLOW_SIZ400   0x0002

/*
 * Masks for ICMP value, for bucket size index and for network mask
 *  Uggly coding trick
 */
#define TMPDU_FLOW_ICMPMASK 0x7f/* value is tmon_flow_descr_t.icmp & 0x7f */
```

```
#define TMPDU_FLOW_SIZEMASK 0x0f/* val is tmon_flow_descr_t.mask_lowSz & 0x0f*/

#define TMPDU_FLOW_LOWNMASK 0xf0/* netmsk low nibble:( .mask_lowSz & 0xf0)>>4*/
#define TMPDU_FLOW_HIGNMASK 0x80/* netmsk high bit:  ( .icmp & 0x80) >>3 */
/*Resulting netmask value is (( .icmp & 0x80) >>3)|(( .mask_lowSz & 0xf0)>>4)*/
```

4.4.5 Flow Instance Protocol Counters Records PDU

pduType == TMON_PDU_TYPE_FLCNTRS 2

Protocol stack dissection and stateful analysis leads to the update of numerous counters of the several types of protocols.

The per-Flow protocol counters are:

```
typedef struct tmon_ipv4_counters {
  uint64_t reassemblyTimeout;
  uint64_t fragmentOverlap;
  uint64_t icmp;
  uint64_t udp;
  uint64_t tcp;
  uint64_t others;
} tmon_ipv4_counters_t;

typedef struct tmon_icmpv4_counters {
  uint64_t icmpCksumFailed; /* not counted as malformed */
  uint64_t icmpCksumSkipped;/* checksum not verified (!UDP/TCP or trunc.) */
  uint64_t echoRequest;
  uint64_t echoReply;
  uint64_t fragNeeded;
  uint64_t srcQuench;
  uint64_t ttlExpired;
  uint64_t reassemblyTimeout;
  uint64_t unReach;          /* except fragNeeded */
  uint64_t redirect;
  uint64_t otherErrMsg;
  uint64_t otherInfo;
} tmon_icmpv4_counters_t;

typedef struct tmon_udpv4_counters {
  uint64_t udpCksumFailed; /* not counted as malformed */
  uint64_t udpCksumSkipped;/* no checksum or not verified (frag. or trunc.) */
  uint64_t udpEmpty;
  uint64_t snmp;
  uint64_t dns;
  uint64_t ntp;
  uint64_t other;
} tmon_udpv4_counters_t;

typedef struct tmon_tcpv4_counters {
  uint64_t tcpCksumFailed; /* not counted as malformed */
  uint64_t tcpCksumSkipped;/* checksum not verified: frag. or trunccated */
  uint64_t tcpRestransmit;
  uint32_t tcpLatePkt;
  uint32_t tcpStartConns;
  uint32_t tcpCleanClose;
```

```
  uint32_t tcpDirtyClose;
  uint32_t ftpCtlConns;
  uint32_t ftpFileXfers;
  uint32_t httpFileXfers;
  uint32_t otherTcpConns;
} tmon_tcpv4_counters_t;

typedef struct tmon_ftp_counters {
  uint32_t ftpStartSessions;/* # started CTL sessions */
  uint32_t ftpCleanClose;   /* # of QUIT commands seen */
  uint32_t ftpDirtyClose;   /* # of FTP ctl session closed without QUIT */
  uint32_t ftpEncrypted;    /* # of FTP ctl session encrypted */
  uint32_t ftpNoLogin;      /* # of FTP ctl session without succ. login */
  uint32_t ftpNoCommand;    /* # of FTP ctl session without logged-in command*/
  uint32_t ftpNoFileXfer;   /* # of FTP ctl session without logged-in xfer */
  uint32_t ftpWithFileXfer; /* # of FTP ctl session with logged-in file xfer */
  uint32_t ftpActives;      /* # of PORT/EPRT/LPRT based data connections,
                                                 including Directory Listing */
  uint32_t ftpPassives;     /* # of PASV/EPSV/LPSV based data connections,
                                                 including Directory Listing */
  uint32_t ftpDirList;      /* # directory listing */
  uint32_t ftpGets;         /* # succeeded RETR of files */
  uint32_t ftpPuts;         /* # succeeded STOR/STOU/APPE of files */
  uint32_t ftpFailedGets;   /* # failed to complete RETR of files */
  uint32_t ftpFailedPuts;   /* # failed to complete STOR/STOU/APPE of files */
  uint32_t ftpRestarts;     /* # of REST commands seen */
  uint32_t ftpDataAborts;   /* # of ABOR commands seen */
  uint32_t ftpLoginFailed;  /* # failed USER/PASS exchanges */
  uint32_t ftpCipherFailed; /* # of AUTH command failures */
  uint32_t ftpCmdFailed;    /* # of Negative Replies to commands */
} tmon_ftp_counters_t;

typedef struct tmon_http_counters {
  /* TBD */
} tmon_http_counters_t;
```

These counters are complemented with the IPv4 size distribution of the packets (or reassembled datagrams).

- In case of packet sizes (maximum 1500 bytes), the size histogram consists in 7 slices covering each a fixed range of 200 bytes, terminated by an open-ended range starting at 1400 bytes.
- In case of reassembled datagram sizes, the size histogram consists in the same 200 bytes-wide 7 initial ranges, followed by an 8th range of 200 bytes ([1400 .. 1600[), followed by 7 slices of fixed-size range of 400 bytes, terminated by an open-ended range starting at 8000 bytes.

```
#define TMON_SIZE_BUCKETS     16 /* Index MUST fit in a nibble for coding */
                                 /* tmon_flow_descr_t struct in PDU */
#define TMON_SIZE_IPV4BUCKLEN 200
typedef struct tmon_sizes {
  uint64_t           sumBytes; /* cummul. volume of IPv4 pkts (hdr+payload) */
  tmon_size_bucket_t szDistrib[TMON_SIZE_BUCKETS]; /* size distribution: */
                                /* bucket  0:    0 <= size <  200 */
                                /* bucket  1:  200 <= size <  400 */
                                /* bucket  2:  400 <= size <  600 */
```

```
                                /* bucket  3:  600 <= size <  800 */
                                /* bucket  4:  800 <= size < 1000 */
                                /* bucket  5: 1000 <= size < 1200 */
                                /* bucket  6: 1200 <= size < 1400 */
                                /* bucket  7: 1400 <= size        */
  /* OR, opt. for datagram/segment bucket  7: 1400 <= size < 1600 */
                        /* optional bucket  8: 1600 <= size < 2000 */
                        /* optional bucket  9: 2000 <= size < 3000 */
                        /* optional bucket 10: 3000 <= size < 4000 */
                        /* optional bucket 11: 4000 <= size < 5000 */
                        /* optional bucket 12: 5000 <= size < 6000 */
                        /* optional bucket 13: 6000 <= size < 7000 */
                        /* optional bucket 14: 7000 <= size < 8000 */
                        /* optional bucket 15: 8000 <= size        */
  uint8_t             szBucketsCnt; /* # of szDistrib buckets actually used */

} tmon_sizes_t;
```

Each slice consists in a set of aggregate statistical values (permitting further aggregating):

```
typedef struct tmon_size_bucket {
    uint16_t  sz_lower;  /* bucket includes this lower boundary */
    uint16_t  sz_upper;  /* value just above the bucket boundary
                            ==0 when not upward bounded */
    uint16_t  sz_min;    /* actual minimal size within this bucket population*/
    uint16_t  sz_max;    /* actual maximal size within this bucket population*/
    uint16_t  sz_avg;    /* average size within this bucket population */
    uint64_t  sz_pop;    /* actual population covered by this size bucket */
    uint64_t  sz_sum;    /* sum of all size of members of this bucket */
    uint64_t  sz_sumsq;  /* sum of the square of all sizes within this bucket*/
} tmon_size_bucket_t;
```

For each Flow Instance whose Flow Class(es) request the regular reporting of <Stats><PacketCounters>, the entire set of protocol counters are regularly sampled and reported to the collector(s) and locally reset to zero for starting a new measurement time period. The reporting period for a Flow Instance is the smallest (divider) of the periods specified by its concerned Flow Classes:

```
<!ELEMENT FlowClass        (Measure, FlowGrain?, Filter+, Condition?)>
<!ATTLIST FlowClass        id           NMTOKEN                    #REQUIRED
                           name         NMTOKEN                    #REQUIRED
                           descr        CDATA                      #IMPLIED
>

<!ELEMENT Measure          (Delay?, Stats?) >
<!ATTLIST Measure          interval     (each|10s|20s|30s
                                        |1min|10min|20min|30min
                                        |1h)                       #REQUIRED
>
<!ELEMENT Stats            (PacketCounters?,TCPConnections?,FileTransfers?) >
<!ATTLIST Stats            verifChksum  (none|bestEffort|fullReassembly)
                                                                   #REQUIRED
>
<!ELEMENT PacketCounters   EMPTY >
<!ATTLIST PacketCounters   for          (firstFragment|allFragments
```

```
                                        |datagram)                #REQUIRED
>
```

For each Flow Instance, the protocol counters record consists in a fixed part and a variable part.

The fixed part refers to the probe-assigned Flow ID, the duration of the reported time interval (period), whether the measurements apply per fragment (per individual packet, either un-fragmented datagram/segment or individual fragment counted separately) or per datagram (either un-fragmented or re-assembled), and the size of the variable part:

```
typedef struct tmon_flow_cntrs {
  uint64_t  prbFlowId; /* Probe assigned ID to flow instance */
  uint16_t  timeIntrvl;/* period is timeIntrvl seconds before PDU ref. time */
  uint16_t  encodgLen; /* length of the variable data[] portion */
  uint8_t   granularity;/* TMPDU_FLCNTR_PERFRAG or TMPDU_FLCNTR_PERDGRM */
  uint8_t   data[1];   /* Byte array of Length/Type/Value entries */
} tmon_flow_cntrs_t;
```

The variable part does report all counters (and size) values resulting from the reported period as integer values encoded as variable length Length-Type-Value.

- Length is the number of bytes occupied by the encoded Value (not counting the 1-byte Type); Length may be zero, meaning that the counter Value is zero and occupies no byte.
- Type is a 1-byte numeric identifier (see below)
- Value is the normal encoding of an integer in network byte-order, but containing only the significant low order bytes (high order bytes that are zero aren't copied in the encoding).

In *tmon_PDU.h*:

```
/*
 * Flow Stats Counter Types
 * ------------------------
 */
#define TMPDU_FLCNTR_PERFRAG    1    /* per packet (fragment) counter */
#define TMPDU_FLCNTR_PERDGRM    2    /* per datagram (reassembled) counter */

#define TMPDU_FLCNTR_TOTBYTES   3    /* ipSizes.sumBytes                   */
#define TMPDU_FLCNTR_SZBUCKTS   4    /* ipSizes.szBucketsCnt               */
/*    For each bucket of size distribution :                              */
#define TMPDU_FLCNTR_SZBKIDX    5    /* index in ipSizes.szDistrib[]       */
#define TMPDU_FLCNTR_SZBKLOW    6    /* ipSizes.szDistrib.sz_lower         */
#define TMPDU_FLCNTR_SZBKUP     7    /* ipSizes.szDistrib.sz_upper         */
#define TMPDU_FLCNTR_SZBKMIN    8    /* ipSizes.szDistrib.sz_min           */
#define TMPDU_FLCNTR_SZBKMAX    9    /* ipSizes.szDistrib.sz_max           */
#define TMPDU_FLCNTR_SZBKAVG   10    /* ipSizes.szDistrib.sz_avg           */
#define TMPDU_FLCNTR_SZBKPOP   11    /* ipSizes.szDistrib.sz_pop           */
#define TMPDU_FLCNTR_SZBKSUM   12    /* ipSizes.szDistrib.sz_sum           */
#define TMPDU_FLCNTR_SZBKSQSM  13    /* ipSizes.szDistrib.sz_sumsq         */
```

In *tmon_statistics.h*:

```
/*
 * Numeric Identifiers of the Statistics Counters
 *
 * These fit in one unsigned byte
 * These codes are also used as counter identifiers in the encoded
 * Flow Instance Counters PDU: starting at 20 in order not to clash
 * with other TMPDU_FLCNTR_XXXX in tmon_PDU.h
 */
#define TMON_STATS_IP_REASTIMOUT        20
#define TMON_STATS_IP_FRAGDUPL          21 /* unused */
#define TMON_STATS_IP_FRAGOVRL          22
#define TMON_STATS_IP_FRAGSEQU          23 /* unused */
#define TMON_STATS_IP_ICMP              24
#define TMON_STATS_IP_UDP               25
#define TMON_STATS_IP_TCP               26
#define TMON_STATS_IP_OTHER             27

#define TMON_STATS_ICMP_CKSUMFAIL       30
#define TMON_STATS_ICMP_CKSUMSKIP       31
#define TMON_STATS_ICMP_ECHOREQ         32
#define TMON_STATS_ICMP_ECHORSP         33
#define TMON_STATS_ICMP_FRAGNEED        34
#define TMON_STATS_ICMP_SRCQHCH         35
#define TMON_STATS_ICMP_TTLEXPRD        36
#define TMON_STATS_ICMP_REASTIMOUT      37
#define TMON_STATS_ICMP_UNREACHD        38
#define TMON_STATS_ICMP_REDIRECT        39
#define TMON_STATS_ICMP_OTHERROR        40
#define TMON_STATS_ICMP_OTHRINFO        41

#define TMON_STATS_UDP_CKSUMFAIL        50
#define TMON_STATS_UDP_CKSUMSKIP        51
#define TMON_STATS_UDP_EMPTY            52
#define TMON_STATS_UDP_SNMP             53
#define TMON_STATS_UDP_DNS              54
#define TMON_STATS_UDP_NTP              55
#define TMON_STATS_UDP_OTHER            56

#define TMON_STATS_TCP_CKSUMFAIL        60
#define TMON_STATS_TCP_CKSUMSKIP        61
#define TMON_STATS_TCP_RETRANSMIT       62
#define TMON_STATS_TCP_LATEPKT          63
#define TMON_STATS_TCP_STARTCONNS       64
#define TMON_STATS_TCP_CLEANCLOSE       65
#define TMON_STATS_TCP_DIRTYCLOSE       66
#define TMON_STATS_TCP_FTPCTLCONNS      67
#define TMON_STATS_TCP_FTPFILEXFERS     68
#define TMON_STATS_TCP_HTTPFILEXFERS    69
#define TMON_STATS_TCP_OTHERCONNS       70

#define TMON_STATS_FTP_STARTSESSIONS    80
#define TMON_STATS_FTP_CLEANCLOSE       81
#define TMON_STATS_FTP_DIRTYCLOSE       82
#define TMON_STATS_FTP_ENCRYPTED        83
#define TMON_STATS_FTP_NOLOGIN          84
#define TMON_STATS_FTP_NOCOMMAND        85
#define TMON_STATS_FTP_NOFILEXFER       86
```

```
#define TMON_STATS_FTP_FILEXFER         87
#define TMON_STATS_FTP_ACTIVES          88
#define TMON_STATS_FTP_PASSIVES         89
#define TMON_STATS_FTP_DIRLIST          90
#define TMON_STATS_FTP_GETS             91
#define TMON_STATS_FTP_PUTS             92
#define TMON_STATS_FTP_FAILEDGETS       93
#define TMON_STATS_FTP_FAILEDPUTS       94
#define TMON_STATS_FTP_RESTARTS         95
#define TMON_STATS_FTP_DATAABORTS       96
#define TMON_STATS_FTP_LOGINFAIL        97
#define TMON_STATS_FTP_CIPHERFAIL       98
#define TMON_STATS_FTP_CMDFAIL          99

#define TMON_STATS_HTTP_XXX            110 /* Not yet */
```

4.4.6 Compact per-Packet/Datagram One Way Observations PDU

pduType == TMON_PDU_TYPE_PKTOBS 3

This contains, for individual packet or reassembled datagram, the probe-assigned ID of the Flow Instance, the IPv4 size and a series of timestamps.

In the XML configuration file, a Flow Class requiring measuring OneWay Delay specifies an ordered list of network Hops the tool must provide the packet timestamps of.

```
<!ELEMENT OneWayDelay     (Hop+, Sign?) >
<!ELEMENT Hop EMPTY >
<!ATTLIST Hop       name            NMTOKEN  #REQUIRED
                    descr           CDATA    #IMPLIED
>
            <!-- Measuring One-Way Latencies:   hopName per timestamp       -->
            <!-- ============================   ====================       -->
            <!-- NOTE: <Hop> inside a <OneWayDelay> tag are ordered!!       -->
            <!-- The order represents the chronology of each timestamp in   -->
            <!-- the sequence, whatever its type (capture/IP/NTP)           -->
            <!--                                                            -->
            <!-- This tag must be in one or more copies in order to         -->
            <!-- specify the sequence of hop timestamps expected for        -->
            <!-- completing a valid record.                                 -->
            <!-- More than one probe (interface) can provide the same time  -->
            <!-- in the case of redundant transmission via more than one    -->
            <!-- link. But in any way, identical timestamp IDs are          -->
            <!-- expected to identify the same logical hop along the path.  -->
            <!-- If a probing point is labeled 'dmz', two different probes  -->
            <!-- reporting times for 'dmz' are supposed to be located in    -->
            <!-- alternate instances of the corporate DMZ.                  -->
            <!-- Every Hop name is assigned inside one or more <Filter>:    -->
            <!--  + either as the capture timestamp at the interface        -->
            <!--  + or as an IP option timestamp of a given sequence number -->
            <!--  + or as an NTP timsetamp of a given type from either the  -->
            <!--     NTP Request or NTP Response                            -->

...
```

```
<!ELEMENT Filter           (On+, CaptureTimeStamp?,
                            (IpOptTimestamp | NtpTimestamp)*, NatPat*,
                                          PacketExpr, FlowGrain?)>

<!ELEMENT On               EMPTY >
<!ATTLIST On               probe        NMTOKEN                    #REQUIRED
                           if           NMTOKEN                    #REQUIRED
>
            <!-- probe:        a reference to the name of the probe         -->
            <!--               to which the Packet Filter Expression applies-->
            <!-- if:           a reference to the probe interface name      -->
            <!--               to which the Packet Filter Expression applies-->

<!ELEMENT CaptureTimeStamp EMPTY >
<!ATTLIST CaptureTimeStamp
                           hopName      NMTOKEN                    #REQUIRED
>
<!ELEMENT IpOptTimestamp   EMPTY >
<!ATTLIST IpOptTimestamp   ipTSNum      NMTOKEN                    #REQUIRED
                           hopName      NMTOKEN                    #REQUIRED
>

<!ELEMENT NtpTimestamp     EMPTY >
<!ATTLIST NtpTimestamp     ntpTime      (originreq|recvdreq
                                        |xmitreq|originrsp
                                        |recvdrsp|xmitrsp)         #REQUIRED
                           hopName      NMTOKEN                    #REQUIRED
>
            <!-- Capture Timestamps on packet/datagram                    -->
            <!-- ======================================================   -->
            <!--   name of the hop timestamp at a probing point          -->
            <!--                                                          -->
            <!-- Optional Additional Timestamps on packet/datagram        -->
            <!-- ======================================================   -->
            <!--                                                          -->
            <!-- + based on IP OPTION TIMESTAMP (milliseconds since midnight-->
            <!--     The nth (ipTSNum) IP timestamp in the flow packets is  -->
            <!--     mapped to the given hopName                          -->
            <!--     1 <= ipTSNum <= 9                                    -->
            <!--                                                          -->
            <!-- + based on UDP NTP Embedded Timestamps                   -->
            <!--     Designates, respectively for a request or response,  -->
            <!--     which of the three relevant NTP Timestamps to report -->
            <!--     as hopName                                           -->
            <!--     This requires firstFragment|datagram                 -->
```

Example:

```
  <FlowClass id="12345" name="withIPTS"
    descr="UDP with port==21 and IP Timestamps">
    <Measure interval="each" >
      <Delay for="datagram" granularity="individual">
        <OneWayDelay>
          <Hop name="remint1"/>
          <Hop name="remint"/>
          <Hop name="alkuf"/>
```

```
          <Hop name="hiros"/>
          <Hop name="kuching"/>
          <Hop name="locdmz"/>
          <Sign>
            <Chunk start="0" relTo="ipPayload" length="40" />
          </Sign>
        </OneWayDelay>
      </Delay>
    </Measure>
    <FlowGrain ref="peers" />

    <Filter>
      <On probe="chieti" if="eth2" />
      <CaptureTimeStamp           hopName="remint"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="21"/>
        </AND>
      </PacketExpr>
    </Filter>

    <Filter>
      <On probe="hutch" if="eth1" />
      <CaptureTimeStamp           hopName="locdmz"/>
      <IpOptTimestamp ipTSNum="1" hopName="remint1"/>
      <IpOptTimestamp ipTSNum="2" hopName="alkuf"/>
      <IpOptTimestamp ipTSNum="3" hopName="hiros"/>
      <IpOptTimestamp ipTSNum="4" hopName="kuching"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="21"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
```

Each Hop identifies

- Either a packet capture time at one among an equivalent set of probe interfaces (e.g. in *Figure 1 above*, if a probe interface was capturing the segment 141.253.14.0/24, and another on the segment 141.253.24.0/24, these two equivalent steps, on alternative network paths, would correspond to the same hop)
- or the n^{th} value of IPv4 Optional list of timestamp accumulated in the IPv4 header during the packet traversal of routers in its network path,
- or one of the NTP timestamps among:
    - NTP Request Origin Time,
    - NTP Request Reception Time,
    - NTP Request Transmit Time,
    - NTP Response Origin Time,

- NTP Response Reception Time,
- NTP Response Transmit Time

When reporting per datagram is requested, and the datagram has been re-assembled from its fragment, two different values are reported for a same capture interface hop:

- the (oldest) timestamp of the first seen fragment (not necessarily the one with offset == 0),
- the (youngest) timestamp of the last seen fragment (not necessarily the one without the more fragment flag)

The encoding is quite compact so as to limit the size of monitoring packets to about 1 % of the observed traffic volume.

4.4.6.1 Assumptions

A probe PDU contains packet observations made by a unique Probe Software Instance.

A One Way Observations PDU contains packet observations from one or more flow instances, grouped by flow instances. A flow instance is assigned by a probe a unique id from a very large 64-bit integer. Probe flow instance Ids are never zero.

One physical probe computer can host several logical probes (ie *n* probes on *1* chassis)

A Probe Software Instance can be identified by source IP + source port of PDU, as well as by the Logical Probe Identifier in the PDU header. Typically, a Physical Probe could be identified by source IP of PDU or source IP subnet of probe PDU.

A packet can (theoretically) match more than one Flow Instance with applicable OneWayDelay specifications. Anyway, the Flow Class definitions are expected to be configured in a consistent and meaningful way: a packet at a probe interface provides single subset of the expected set of hop timestamps among:

- the capture time at the interface,
- the ordered list of (up to 9) IP Option Timestamps
- the three NTP timestamps of either a request or a reply packet.

All expected hops, from (the applicable set of) flow class(es), form an ordered list. Each hop can therefore be designated by its sequence number (starting at 1) instead of by hopName.

Timestamps are expected to be close to each other, but a travel time as high as 10 seconds is not unrealistic, e.g. with IP option timestamps or with remote NTP timestamp.

At a given time, flow instances alive in a probe are expected to have clustered Flow Ids, but there may be some very ancient ones whose Ids are significantly distant numbers wrt the others.

A One Way Observations PDU carries one or more hop timestamps per observed data unit. When data unit is a reassembled datagram, its packet capture hop consists in two timestamps: first and last observed fragments.

A One Way Observations PDU also carries the size (in bytes) of the [optionally reassembled] data unit.

Some code tags are common to both formats of One Way Observations PDU as well as Individual Delays PDU (see section 4.4.7 below):

[0xfa]: TMP_PDU_CODE_FLOW
with 8-byte absolute ProbeFlowID reference

[0xfb]: TMP_PDU_CODE_RLFLW
with 2-byte relative ProbeFlowID reference and 2-byte unique FlowClass "id" and also, for OneWay Observations, the number and index list of timestamps,

[0xfc]: TMP_PDU_CODE_TS
with 2-byte relative ¼ second reference

[0xfd]: TMP_PDU_CODE_ADDFLW
with 8-byte additional Flow reference

Some code tags are specific to the data being encoded:

[0xfe]: TMP_PDU_CODE_FRGNUM, for One Way Observations
with 1-byte fragment number (for second to before-last frags)

[0xfe]: TMP_PDU_CODE_DELTYP, for Delay Observations
with 1-byte Delay Type

[0xff]: TMP_PDU_CODE_LSTFRG, for One Way Observations
alone, attached to a last fragment

[0xff]: TMP_PDU_CODE_INITOR, for Delay Observations with Round-Trip Delay alone, meaning the delay is round-trip between initiator and probe

[0xff]: TMP_PDU_CODE_ITRFRG, for Delay Observations with Inter-Unit Delay alone, meaning the delay is between 2 successive fragments of a fragmented datagram

4.4.6.2 Encoding Format

At any position inside the encoding stream, there is a known Reference Flow ID (and its Flow Class ID), a Reference list of network Hops and a known Reference Time expressed in ¼ second (`quartSec`).

The main component of the encoding covers a packet record whose primary capture time is close to the reference `quartSec` (before `quartSec` +1), and which belongs to the reference Flow ID and provides timestamps for the reference Hop list.

This is encoded as the positive delay (in milliseconds) from `quartSec`, followed by IP size, by the signature hash bytes and by N-1 timestamps encoded as a signed offset from the primary capture time of the packet.

When the reported packet is a second or subsequent IP fragment, it is added the tag TMP_PDU_CODE_FRGNUM or TMP_PDU_CODE_LSTFRG. For intermediate fragments, the tag is followed by a one-byte X, so that n^{th} fragment is given X=n (n==1 for first fragment).

When next record has its primary capture timestamp distant from the current `quartSec` by more the ¼ second, then a new TMP_PDU_CODE_TS reference `quartSec` is inserted.

When next record belongs to a (primary) flow different from the reference and/or provides a different Hop timestamps list, a new reference is inserted. If the new reference Flow ID is distant by more than +/- 0x7FFF (+/- 32767) from the previous reference, then a new TMP_PDU_CODE_FLOW tag is inserted with its new 8-byte absolute Flow ID reference. A new TMP_PDU_CODE_RLFLW tag is inserted with is signed relative delta from the reference Flow ID, its Flow Class ID, the number of Hops and the list of positions (first is 1) of Hops in the Flow Class XML definition.

GRAMMAR (where * means 0 or more, + means 1 or more, xN means N times)

PDU_PAYLD: { CODE DATA }+

CODE: 1 byte
DATA: *n* bytes according to the value of CODE
Possible combinations for (CODE DATA) are:

Specific short codes:

[0xfa] FLOW
FLOW: Current Flow instance complete identifier - 8 bytes

[0xfb] RELFLW FLOWCLS N {TINDEX}xN
RELFLW: Current Flow instance relative identifier - 2 bytes of signed delta wrt last mentioned FLOW
FLOWCLS: Corresponding applicable XML <FlowClass> “id” attribute
2 bytes
N: Number of timestamps to be reported for packet
In upper nibble (1/2 byte)
TINDEX: Respective sequence indexes of TS, then of ADDTS, for the flow (first == 1)
One nibble (1/2 byte) each

[0xfc] TS
TS: 2-byte timestamp in quarts of a second as a signed delta wrt REFTS

[0xfd] M {FLOW}xM

M: Number of following additional Flow instances for packet (1-byte)
FLOW: Additional Flow instance complete identifier for packet - 8 bytes

[0xfe] X
X: This is (X+2)th fragment
[0xff]
This is the last fragment
First fragment has neither of [0xfe] or [0xff]
Last fragment has only [0xff]
From second to before-last fragment: [0xfe] code

Packet obs. main data format:

[0x00-0xf9] SIZE HASH {ADDTS}xN-1
[0x00-0xf9]: First timestamp of the packet at this probe/interface expressed as a time offset from the current quart of a second (i.e. from last TS) - Permitted values are 0 to 249 msec.
SIZE: Size in bytes of the observed data unit (2 bytes, Maximum payload datagram is 65535 bytes + size of IP Header, but assumed that it always fit in a 2 bytes unsigned).
HASH: The hash of the packet observed (2..5 bytes of MD5).
For reassembled datagram, this is the IPv4 size of first fragment plus that of the cumulated IPv4 payload of subsequent fragments
ADDTS: [-127..-1,+0..+127] | [0xff] RELTS
Additional timestamp for the packet: as per the sequence of TINDEX defined by the [0xfb] directive.
Expressed as a signed number of milliseconds relative to the first timestamp of the packet ([0x00-0xf9])
Either encoded as a 1-byte one's complement (without -0). Or [0xff] (= -0 in one's complement) followed by signed 16-bit.

4.4.7 Individual Delays PDU

pduType == `TMON_PDU_TYPE_DELOBS` `4`

When a probe is requested to individually (not `probeAggregated`) measure either Two-Way delays (`RoundTripDelay`) of specific transactions (peer datagram in each direction) or Inter-Unit delays (`InterPacket` not implemented) between successive captured datagrams and between successive captured fragments of a same datagram, it must report each pair of observed timestamps to the collector(s).

The encoding mechanism is here quite similar (though simpler) to the one described in section 4.4.6.2 above. But only two timestamps are provided per observation record:

- primary timestamp is that of the "request" (one way data unit),
- the second timestamp is that of the "response" (corresponding return peer data unit)

Delay Observations PDU format is a simplified variant of One Way Observations encoding: N==2 timestamps per observation: start then end time of the delay, in that order (no TINDEX sequence number is needed)

In case a (typically response) datagram is fragmented (only possible with SNMP, for round-trip): when RT delay is per datagram, corresponding datagram peer time is that of the youngest fragment, otherwise it is that of the oldest fragment.

A same Delay Observations PDU type is used for both Round-Trip delay and inter-data unit delay.

Some code tags are common to both formats of One Way Observations PDU (see section 4.4.6.2 above) as well as Individual Delays PDU:

[0xfa]: TMP_PDU_CODE_FLOW
with 8-byte absolute ProbeFlowID reference

[0xfb]: TMP_PDU_CODE_RLFLW
with 2-byte relative ProbeFlowID reference and 2-byte unique FlowClass “id” and also, for OneWay Observations, the number and index list of timestamps,

[0xfc]: TMP_PDU_CODE_TS
with 2-byte relative ¼ second reference

[0xfd]: TMP_PDU_CODE_ADDFLW
with -byte additional Flow reference

Some code tags are specific to the data being encoded:

[0xfe]: TMP_PDU_CODE_FRGNUM, for One Way Observations
with 1-byte fragment number (for second to before-last frags)

[0xfe]: TMP_PDU_CODE_DELTYP, for Delay Observations
with 1-byte Delay Type

[0xff]: TMP_PDU_CODE_LSTFRG, for One Way Observations
alone, attached to a last fragment

[0xff]: TMP_PDU_CODE_INITOR, for Delay Observations with Round-Trip Delay alone, meaning the delay is round-trip between initiator and probe

[0xff]: TMP_PDU_CODE_ITRFRG, for Delay Observations with Inter-Unit Delay alone, meaning the delay is between 2 successive fragments of a fragmented datagram

4.4.7.1 Encoding Format

At any position inside the encoding stream, there is a known Reference Flow ID (and its Flow Class ID), a Reference list of network Hops and a known Reference Time expressed in ¼ second (`quartSec`).

The main component of the encoding covers a packet record whose primary capture time is close to the reference `quartSec` (before `quartSec` +1), and which belongs to the reference Flow ID and provides timestamps for the reference Hop list.

This is encoded as the positive delay (in milliseconds) from `quartSec`, followed by IP size, by the signature hash bytes and by N-1 timestamps encoded as a signed offset from the primary capture time of the packet.

When the reported packet is a second or subsequent IP fragment, it is added the tag TMP_PDU_CODE_FRGNUM or TMP_PDU_CODE_LSTFRG. For intermediate fragments, the tag is followed by a one-byte X, so that n^{th} fragment is given X=n (n==1 for first fragment).

When next record has its primary capture timestamp distant from the current `quartSec` by more the ¼ second, then a new TMP_PDU_CODE_TS reference `quartSec` is inserted.

When next record belongs to a (primary) flow different from the reference and/or provides a different Hop timestamps list, a new reference is inserted. If the new reference Flow ID is distant by more than +/- 0x7FFF (+/- 32767) from the previous reference, then a new TMP_PDU_CODE_FLOW tag is inserted with its new 8-byte absolute Flow ID reference. A new TMP_PDU_CODE_RLFLW tag is inserted with is signed relative delta from the reference Flow ID, its Flow Class ID, the number of Hops and the list of positions (first is 1) of Hops in the Flow Class XML definition.

GRAMMAR (where * means 0 or more, + means 1 or more, xN means N times)

PDU_PAYLD: { CODE DATA }+

CODE: 1 byte
DATA: *n* bytes according to the value of CODE
Possible combinations for (CODE DATA) are:

Specific short codes:

[0xfa] FLOW
FLOW: Current Flow instance complete identifier - 8 bytes

[0xfb] RELFLW FLOWCLS
RELFLW: Current Flow instance relative identifier - 2 bytes of signed delta wrt last mentioned FLOW
FLOWCLS: Corresponding applicable XML <FlowClass> “id” attribute 2 bytes

[0xfc] TS
TS: 2-byte timestamp in quarts of a second as a signed delta wrt REFTS

[0xfd] M {FLOW}xM
M: Number of following additional Flow instances for packet (1-byte)
FLOW: Additional Flow instance complete identifier for packet - 8 bytes

[0xfe] TYPE

TYPE: Either TMP_PDU_DEL_2WAY(==1) for round-trip delay, or TMP_PDU_DEL_INTER(==2) for inter-unit delay

[0xff]
Either (TYPE == TMP_PDU_DEL_2WAY): Round-trip with Initiator (instead of with Responder)
Or (TYPE == TMP_PDU_DEL_INTER): Inter-fragment delay (instead of inter-datagram, fragmented or not)

Packet obs. main data format:

[0x00-0xf9] HASH {ADDTS}x1

[0x00-0xf9]: First timestamp of the packet at this probe/interface expressed as a time offset from the current quart of a second (i.e. from last TS) - Permitted values are 0 to 249 msec.

HASH: The hash of the packet observed (2..5 bytes of MD5).

ADDTS: [-127..-1,+0..+127] | [0xff] RELTS
Second timestamp for the delay
Expressed as a signed number of milliseconds relative to the first timestamp of the packet ([0x00-0xf9])
Either encoded as a 1-byte one's complement (without -0). Or [0xff] (= -0 in one's complement) followed by signed 16-bit.

4.4.8 Metric Single or Multi-Slice (Histogram) Aggregate Description PDU

pduType == `TMON_PDU_TYPE_HISTDSC  5`

For those metrics which are computed inside the probe and whose values are aggregated over a specified regular reporting time period, the probe publishes once to the collector(s) the description of the aggregate properties.

These properties derive from applicable ("`probeAggregated`" Measure/Delay) Flow Class <Histogram> definition:

```
<!ELEMENT Histogram        EMPTY >
<!ATTLIST Histogram        lowBound     NMTOKEN                    #REQUIRED
                           highBound    NMTOKEN                    #REQUIRED
                           sliceCount   NMTOKEN                    "12"
>

<!-- For any three types of Delay, EITHER "collectorAggregated" or          -->
<!-- "probeAggregated" Histogram MAY (optional) be specified to preserve    -->
<!-- several ranges of observed delay values:                               -->
<!--                                                                        -->
<!-- When Histogram is not specified, all values are aggregated over time   -->
<!-- period into a single unbound bucket (no histogram slices)              -->
<!--                                                                        -->
```

```
<!--                lowBound:  First slice covers all values <  lowBound     -->
<!--               highBound:  Last  slice covers all values >= highBound    -->
<!--              sliceCount:  In between first and last slices, there are   -->
<!--                           sliceCount intervals of values, of equal    -->
<!--                           length                                        -->
<!--       Currently only used for Delay metrics, the values are             -->
<!--       signed integers coded in 32 bit. Specifying any bound             -->
<!--       as either INT32_MIN (0xffff or 147483648)                        -->
<!--              or INT32_MAX (0x7fff or  2147483647)                       -->
<!--       means "unbound".                                                  -->
```

As for the Protocols Counters PDU (section 4.4.5 above), the per-histogram record consists in a fixed size part followed by variable length encoded list of fields.

Fixed part:

```
/*
 * Fixed metric instance histogram description header
 * --------------------------------------------------
 */
typedef struct tmon_hist_desc {
  uint64_t  prbFlowId; /* Probe assigned ID to flow instance */
  uint16_t  encodgLen; /* length of the variable data[] portion */
  uint8_t   metricType;/* TMPDU_MTR_XXX */
  uint8_t   slicesCnt; /* how many histogram slices for this metric instance?*/
  uint8_t   data[1];   /* Byte array of Length/Type/Value entries */
} tmon_hist_desc_t;
```

The *metricType* is one of

```
/* delays */
#define TMPDU_MTR_LATENCY (TMON_MTR_DELAY_1WAY+1)
                        /* one-way latency (between 2 interfaces of a probe) */
#define TMPDU_MTR_RTRSPDR (TMON_MTR_DELAY_2W_RSPDR+1)
                        /* round-trip time towards protoc. responder(svr )*/
#define TMPDU_MTR_RTINITR (TMON_MTR_DELAY_2W_INITR+1)
                        /* round-trip time towards protoc. initiator(cli) */
#define TMPDU_MTR_ITRDRGM (TMON_MTR_DELAY_INTERDGRM+1)
                        /* inter-datagram delay at probe interface */
                        /*      for unfragmented or reassembled datagrams */
#define TMPDU_MTR_ITRFRAG (TMON_MTR_DELAY_INTERFRAG+1)
                        /* delay inter incomplete fragments of a datagram */
/* others could come later */
```

where

```
 * A. DELAY METRICS
 * ++++++++++++++++
 * Five possible instance types of probe-aggregated delay histograms:
 *
 * ==> index in table inside a flow instance record in probe.
 */
#define TMON_MTR_DELAY_1WAY      0 /* packet latency between probing
                                                              interfaces */
#define TMON_MTR_DELAY_2W_RSPDR  1 /* round-trip time towards responder */
#define TMON_MTR_DELAY_2W_INITR  2 /* round-trip time towards initiatior */
#define TMON_MTR_DELAY_INTERDGRM 3 /* inter-datagram delay */
#define TMON_MTR_DELAY_INTERFRAG 4 /* inter-fragment (of a same datagram)
```

```
                                                                    delay */
#define TMON_MTR_DELAY_COUNT     5
```

Code Fields with variable-length encoding:

```
/*
 * Variable-length metric instance histogram description fields
 * ------------------------------------------------------------
 */
#define TMPDU_HISTO_LOW  1  /* Low  Bound of histogram */
#define TMPDU_HISTO_HIGH 2  /* High Bound of histogram */
```

4.4.9 Metric Instances Data PDU

pduType == TMON_PDU_TYPE_HISTDTA 6

Probe-aggregated metric instances (only the 2-Way round-trip delays in Phase I) consist in one time period interval of aggregated characteristics of the values distributed over one or multiple range slices of possible values (see above).

Not all defined slices have actually observed values of the metric instance over the reported time interval. Only those slices with non null population are reported. A slice is reported with its index number (first is 1), its observed population, observed min and max, the sum of observed values and the sum of the square of observed values. Note that the bounds and length of every slice is assumed already known by the collector(s) thanks to the above TMON_PDU_TYPE_HISTDSC PDU.

One metric instance data interval consists in a fixed part record followed by variable-length encodings of slice data fields.

Fixed part:

```
typedef struct tmon_hist_rec {
  uint64_t  prbFlowId; /* Probe assigned ID to flow instance */
  uint16_t  timeIntrvl;/* period is timeIntrvl seconds before PDU ref. time */
  uint16_t  encodgLen; /* length of the variable data[] portion */
  uint8_t   metricType;/* TMPDU_MTR_XXX */
  uint8_t   flags;     /* TMPDU_MTR_FLAG_XXX */
  uint8_t   slicesCnt; /* how many histogram slices for this metric instance?*/
  uint8_t   data[1];   /* Byte array of Length/Type/Value entries */
} tmon_hist_rec_t;
/*
 * probe-aggregated metrics characterising flags
 * ---------------------------------------------
 */
/* NONE DEFINED YET */
/* #define TMPDU_MTR_FLAG_XXX 0x00 */
```

Code Fields with variable-length encoding:

```
#define TMPDU_SLICE_INDEX 1 /* Slice # (index starts at 1) */
#define TMPDU_SLICE_POP   2 /*Population: # observed values within slice bnd */
#define TMPDU_SLICE_MIN   3 /*Minimum value observed within slice boundaries */
#define TMPDU_SLICE_MAX   4 /*Maximum value observed within slice boundaries */
```

```
#define TMPDU_SLICE_SUM   5 /*Sum of observed values within slice boundaries */
#define TMPDU_SLICE_SUMSQ 6 /*Sum of square of obs values within slice bounds*/
```

4.4.10 Per-TCP Connection Stateful Observation Data PDU

pduType == `TMON_PDU_TYPE_TCPCONN  7`

Being reported regularly or at end, the per-TCP connection data are reported into records with a fixed part, identifying the connection and referring to the (primary) flow instance, and a variable part, as a series of state and performance values, at the level of the connection, then for each of its directions.

Those reported TCP connections are

- those match the **Filter criteria** of a Flow Class requesting to Measure TCPConnections with a granularity="`each`" or "`groupedByCollector`" (**e.g.** matching the **control connection** of FTP sessions)
- and by extension, where the Flow Class also File Transfers measurement of protocol="`FTP`", the indirectly discovered and matched **file transfer data connections** over the same FTP sessions (*data connections for FTP file listing are not reported*)

Fixed part:

```
typedef struct tmon_tcpconn_rec {
  uint64_t  prbFlowId; /* Probe assigned ID to flow instance */
  uint32_t  tcpAddrA;
  uint32_t  tcpAddrB;
  uint16_t  tcpPortA;
  uint16_t  tcpPortB;
  uint16_t  ifID;      /* optional: 0 for NULL */
  uint8_t   connState; /* TCP Connection State Information: see below */
  uint8_t   connOpts;  /* Flags of used TCP Options: see below */
  uint16_t  encodgLen; /* Length of the variable data[] portion */
  uint8_t   data[1];   /* Byte array of Length/Type/Value entries */
} tmon_tcpconn_rec_t;
```

The fixed part of record contains the set of fields (except for the *first seen time* in the variable length part) permitting to uniquely identify it, as well as it probing point (when a same connection could be seen at more than one reporting probing interface: A side is that whose IP address is the lowest or, in case of equality (?!? same host, normally localhost generic address), the lowest port:

- IPv4 address A-side
- IPv4 address B-side
- TCP port A-side,
- TCP port B-side,
- Probing Interface global ID
- First seen time (in variable length part)

The fixed part also contains the current operational state (connection report could be produced during the connection lifetime, in case of regular reporting):

```
/*
 * Values for the connState TCP Connection State information
 *                ==========
 *
 * 2bits: global connection state
 * 2bits: which side (A or B) is initiator, if known
 * 2bits: which side (A or B) is terminator, if known
 * 1bit:  whether A sent RESET
 * 1bit:  whether B sent RESET
 */
#define TMPDU_TCPCONN_STATE 0x03 /* mask for overall connection state bits */
#define TMPDU_TCPCONN_SYN     0x00 /* one-way SYN but no SYN+ACK */
#define TMPDU_TCPCONN_DATA    0x01 /* two-way SYN: in data mode */
#define TMPDU_TCPCONN_FIN     0x02 /* one-way FIN or RST, but no rev. FIN/RST*/
#define TMPDU_TCPCONN_CLS     0x03 /* closed by both parties, via FIN or RST */

#define TMPDU_TCPCONN_INITR 0x0c /* mask for TCP Connection Initiator */
#define TMPDU_TCPCONN_INIUNK  0x00 /* Initiator not known */
#define TMPDU_TCPCONN_INITA   0x04 /* Initiator is A side */
#define TMPDU_TCPCONN_INITB   0x08 /* Initiator is B side */

#define TMPDU_TCPCONN_TERM  0x30 /* mask for TCP Connection Terminator */
#define TMPDU_TCPCONN_TRMUNK  0x00 /* Terminator not known */
#define TMPDU_TCPCONN_TERMA   0x10 /* Terminator is A side */
#define TMPDU_TCPCONN_TERMB   0x20 /* Terminator is B side */

#define TMPDU_TCPCONN_RSTA  0x40 /* Side A has sent RESET */
#define TMPDU_TCPCONN_RSTB  0x80 /* Side B has sent RESET */
```

And a flags byte also identifies the set of mutually agreed TCP Options:

```
/*
 * Values for the connOpts TCP Connection Options flags
 *                ==========
 */
#define TMPDU_TCPCONN_WINSCL 0x01 /* is the connection using Window scale? */
#define TMPDU_TCPCONN_RTTM   0x02 /* is the connection using TCP Timestamp?*/
#define TMPDU_TCPCONN_MSSA   0x04 /* has A side sent a MSS value? */
#define TMPDU_TCPCONN_MSSB   0x08 /* has B side sent a MSS value? */
#define TMPDU_TCPCONN_SACKA  0x10 /* has A side sent a SACK-permitted? */
#define TMPDU_TCPCONN_SACKB  0x20 /* has B side sent a SACK-permitted? */
```

The variable length part consists in fields encoded as Length-Type-Value, as for the above described Protocol Counters PDU (section 4.4.5). The list of TCP connection state and performance observation fields of the variable length part is (A and B identifying the two directions, respectively [AtoB] and [BtoA]):

```
/*
 * Possible TYPE of variable length TCP connection fields
 * ======================================================
 */
#define TMPDU_TCPCONN_1STTIME   1   /* tcpConnFirstTime */
                                    /* POSITIVE Delta sec vs. Hdr.pduTimeRef*/
```

```
#define TMPDU_TCPCONN_SEGBYTA   2   /* tcpConnDir[ATOB].tcpConnDirSegmBytes */
#define TMPDU_TCPCONN_PLDBYTA   3   /* tcpConnDir[ATOB].tcpConnDirPaylBytes */
#define TMPDU_TCPCONN_1STBYTA   4   /* tcpConnDir[ATOB].tcpConnDir1stPlByts */
#define TMPDU_TCPCONN_RTRBYTA   5   /* tcpConnDir[ATOB].tcpConnDirRetrPayld */
#define TMPDU_TCPCONN_SEGCNTA   6   /* tcpConnDir[ATOB].tcpConnDirSegmCount */
#define TMPDU_TCPCONN_1STCNTA   7   /* tcpConnDir[ATOB].tcpConnDirFirstCnt  */
#define TMPDU_TCPCONN_RTRCNTA   8   /* tcpConnDir[ATOB].tcpConnDirRetrSegmt */
#define TMPDU_TCPCONN_MTYACKA   9   /* tcpConnDir[ATOB].tcpConnDirEmAckCnt  */

#define TMPDU_TCPCONN_LASTTMA  10   /* tcpConnDir[ATOB].tcpConnDirLastTime  */
                                    /* POSITIVE Delta sec vs. Hdr.pduTimeRef*/

#define TMPDU_TCPCONN_WUDACKA  11   /* tcpConnDir[ATOB].tcpConnDirWouldAck  */
                                    /* in Relative Sequence from first seen */
                                    /* == actual payload volume sent & seen */
#define TMPDU_TCPCONN_1stWINA  12   /* First window size announced by B     */
#define TMPDU_TCPCONN_MAXWINA  13   /* Max. window real size announced by B */
#define TMPDU_TCPCONN_LSTWINA  14   /* Last window real size announced by B */

#define TMPDU_TCPCONN_SEGBYTB  15   /* tcpConnDir[BTOA].tcpConnDirSegmBytes */
#define TMPDU_TCPCONN_PLDBYTB  16   /* tcpConnDir[BTOA].tcpConnDirPaylBytes */
#define TMPDU_TCPCONN_1STBYTB  17   /* tcpConnDir[BTOA].tcpConnDir1stPlByts */
#define TMPDU_TCPCONN_RTRBYTB  18   /* tcpConnDir[BTOA].tcpConnDirRetrPayld */
#define TMPDU_TCPCONN_SEGCNTB  19   /* tcpConnDir[BTOA].tcpConnDirSegmCount */
#define TMPDU_TCPCONN_1STCNTB  20   /* tcpConnDir[BTOA].tcpConnDirFirstCnt  */
#define TMPDU_TCPCONN_RTRCNTB  21   /* tcpConnDir[BTOA].tcpConnDirRetrSegmt */
#define TMPDU_TCPCONN_MTYACKB  22   /* tcpConnDir[BTOA].tcpConnDirEmAckCnt  */

#define TMPDU_TCPCONN_LASTTMB  23   /* tcpConnDir[BTOA].tcpConnDirLastTime  */
                                    /* POSITIVE Delta sec vs. Hdr.pduTimeRef*/

#define TMPDU_TCPCONN_WUDACKB  24   /* tcpConnDir[BTOA].tcpConnDirWouldAck  */
                                    /* in Relative Sequence from first seen */
                                    /* == actual payload volume sent & seen */
#define TMPDU_TCPCONN_1stWINB  25   /* First window size announced by A     */
#define TMPDU_TCPCONN_MAXWINB  26   /* Max. window real size announced by A */
#define TMPDU_TCPCONN_LSTWINB  27   /* Last window real size announced by A */
```

4.4.11 Per-File Transfer Information PDU

pduType == `TMON_PDU_TYPE_FTPXFER  8`

Being reported regularly or at end, the observed file transfers (currently only FTP) are reported into records with a fixed part – with the probe Flow ID and identifying (except for their first seen time) the underlying data TCP connection and the underlying FTP session control TCP connection, as well as providing type flags and state – and a variable part, as a series of information field values.

```
typedef struct tmon_ftpxfer_rec {
  uint64_t  prbFlowId; /* Probe assigned ID to flow instance */
  uint32_t  cliAddr;
  uint32_t  svrAddr;
  uint16_t  cliDataPort;
  uint16_t  svrDataPort;
  uint16_t  cliCtlPort; /* From corresponding FTP Control session */
```

```
  uint16_t   svrCtlPort; /* From corresponding FTP Control session */
  uint16_t   ifID;       /* optional: 0 for NULL */
  uint16_t   encodgLen;  /* Length of the variable data[] portion */
  uint8_t    xferTypes;
  uint8_t    connState;  /* Same as for TCP connection: TMPDU_TCPCONN_STATE */
  uint8_t    data[1];    /* Byte array of Length/Type/Value entries */
} tmon_ftpxfer_rec_t;
```

The type flags characterises the FTP file transfers:

```
/*
 * Values for the xferTypes FTP File Transfer Characteristics
 *                 =========
 * 2bits: GET, PUT or NULL (unknown)
 *
 * 2bits: ACTIVE, PASSIVE or NULL(unknown)
 *
 * 2bits: ASCII, BINARY(Image), EBCDIC or NULL(unknown)
 *   Note: Local Byte Size is coded as NULL(unknown)
 *
 * 2bits: Stream, Block, Compressed or NULL(unknown)
 */
/*
 * FTP File Transfer Direction
 * ---------------------------
 */
#define TMPDU_FTPXFDIR_NULL 0x00 /* Unknown Get or Put operation */
#define TMPDU_FTPXFDIR_GET  0x01 /* RETR command */
#define TMPDU_FTPXFDIR_PUT  0x02 /* STOR, STOU, APPE command */
#define TMPDU_FTPXFDIR_MASK 0x03

/*
 * FTP Data Connection Mode
 * ------------------------
 */
#define TMPDU_FTPDATA_NULL 0x00/* Unknown Initiating side of Data Connection */
#define TMPDU_FTPDATA_ACTV 0x04/* ACTIVE  Data Connection (via PORT equiv.t) */
#define TMPDU_FTPDATA_PASV 0x08/* PASSIVE Data Connection (via PASV equiv.t) */
#define TMPDU_FTPDATA_MASK 0x0C

/*
 * FTP File Type
 * -------------
 */
#define TMPDU_FTPTYPE_NULL 0x00 /* Unknown Type of Data File */
#define TMPDU_FTPTYPE_LCL  TMPDU_FTPTYPE_NULL /* Local Byte Size == Unknown*/
#define TMPDU_FTPTYPE_ASCI 0x10 /* ASCII Type of Data File */
#define TMPDU_FTPTYPE_BINY 0x20 /* BINARY(Image) Type of Data File */
#define TMPDU_FTPTYPE_EBCD 0x30 /* EBCDIC Type of Data File */
#define TMPDU_FTPTYPE_MASK 0x30
/*
 * FTP File Structure
 * ------------------
 */
#define TMPDU_FTPMODE_NULL 0x00 /* Unknown File Structure */
#define TMPDU_FTPMODE_STRM 0x40 /* STREAM of bytes File Structure */
#define TMPDU_FTPMODE_BLOK 0x80 /* BLOCK File Structure */
```

```
#define TMPDU_FTPMODE_CPRS 0xC0 /* COMPRESSED BLOCK File Structure */
#define TMPDU_FTPMODE_MASK 0xC0
```

The state of the file transfer is that of its underlying data connection (see 4.4.10 above).

The information fields of the variable length part are encoded as Length-Type-Value (as described in section 4.4.5 above):

```
/*
 * Possible TYPEs of variable length FTP File Transfer fields
 *          ================================================
 */
#define TMPDU_FTPXFER_FILE 1   /* ftpDataFileName */
#define TMPDU_FTPXFER_WDIR 2   /* ftpDataFileName */
#define TMPDU_FTPXFER_SIZE 3   /* ftpDataFileSz */
#define TMPDU_FTPXFER_BYTE 4   /* Total TCP Payload transferred, incl. retr. */
             /* NOTE: for <FileTransfers ftpdata=start-stop>                  */
             /*       This transferred volume is deduced from the difference */
             /*       of TCP Seq # between FIN/RST packet and SYN packet.    */
             /*   For large files (> 4GB), sequence numbers may have  wrapped*/
             /*   added N* 0xffffffff = 4 294 967 295 bytes                  */
#define TMPDU_FTPXFER_OFST 5   /* !=0 means File xfer is RESTarted at offset */
#define TMPDU_FTPXFER_USER 6   /* Username logged in for this FTP Ctl session*/
#define TMPDU_FTPXFER_FRST 7   /* First time packet has been seen over data  */
                               /* connection: as a positive delta to be      */
                               /* subtrated from the PDU reference time      */
#define TMPDU_FTPXFER_LAST 8   /* Last time packet has been seen over data   */
                               /* connection: as a positive delta to be      */
                               /* subtrated from the PDU reference time      */
#define TMPDU_FTPXFER_CTTM 9   /* First time packet has been seen over contrl*/
                               /* connection: as a positive delta to be      */
                               /* subtrated from the PDU reference time      */
```

Note that the TMPDU_FTPXFER_FRST and TMPDU_FTPXFER_CTTM complement the pair of IP address/port number for uniquely referring, respectively, to the data and control underlying TCP connections.

4.4.12 Events PDU

pduType == TMON_PDU_TYPE_EVENTS 9

Events have a time, a type, a severity and a message string.

```
typedef struct tmon_event_rec {  uint32_t  cliAddr;
  int32_t   tmev_deltaTime; /* milliseconds delta vs. the PDU ref. time */
  uint16_t  encodgLen;      /* Length of the variable data[] portion */
  uint8_t   tmev_type;      /* see #define TMEV_XXX */
  uint8_t   tmev_severity;  /* see #define TMON_SEV_XXX */
  uint8_t   data[1];        /* Byte array of Length/Type/Value entries */
} tmon_ftpxfer_rec_t;
```

The event types and severities are as follow:

```
/*
 * Possible Event Types
 */
```

```
#define TMEVT_START             1  /* This entity cold starts */
#define TMEVT_WILLRESTART       2  /* This entity will auto-restart */
#define TMEVT_STOP              3  /* This entity stops */
                                /* maybe upon catchable terminating signal ? */
#define TMEVT_FULLSPEED         4  /* This probe reaches full capacity */
                                /* child input queue reaches 97 % full */
#define TMEVT_NOMINAL           5  /* This probe is back to nominal load */
                                /* child input queue back below 50 % full */
#define TMEVT_UP                6  /* Peer entity is (again) seen */
#define TMEVT_SILENT            7  /* Peer declared silent:
                                          Either down or diconnected from us */
#define TMEVT_OBSDROPPING       8  /* Peer silent probe has expired its
                                    configured 'dropObsFinalTimeout' delay */

#define TMEVT_IFSILENT          9  /* No packet captured at this probe I/F */
                                /* repeated while no pkt recvd in time period */
#define TMEVT_IFDROPPING       10  /* This probe I/F is dropping packets */
                                /* repeated with # drops in time period */
#define TMEVT_IFNOMINAL        11  /* This probe I/F is back to nominal */
                                /* once after prev. period IFSILENT/IFDROPPING*/

#define TMEVT_FLW_1WLOSSES     12  /* count>0 of packet loss in interval */
#define TMEVT_FLW_1WINCOMPL    13  /* count>0 of partial but non-lost 1-way
                                      obs in interval */
#define TMEVT_FLW_1WSIGNCLASH  14  /* Detected collision of 1-way pkt/dgram
                                                        signature hash */
/*
 * Event Severities
 */
#define TMON_SEV_UNKNOWN    1
#define TMON_SEV_NORMAL     2
#define TMON_SEV_MINOR      3
#define TMON_SEV_MAJOR      4
#define TMON_SEV_CRITICAL   5
#define TMON_SEV_MAX        5
#define TMON_SEV_MIN        1
```

The last 3 event types, about one-way observations, are only detected by the collector, hence never encoded in a probe PDU.

The information fields of the variable length part are encoded as Length-Type-Value (as described in section 4.4.5 above):

```
/*
 * Possible TYPEs of variable length Event fields
 *          ====================================
 */
#define TMPDU_EVENT_ENTYNM  1   /* Entity (probe) name */
#define TMPDU_EVENT_INTFNM  2   /* Probe interface name, where relevant */
#define TMPDU_EVENT_FLOWID  3   /* Flow identifier (in probe), where relevant */
#define TMPDU_EVENT_MESSAGE 4   /* Event message string */
```

4.5 PROBE LOCAL SAVING OF PDU'S (UNUSED)

Although not currently used, the probe can save all its formatted PDU payload into a local file.

The XML configuration file defines the directory where to create the local PDU log files:

```
<!ELEMENT Probe           (Interface+, PDUSending*, PDUSaving?) >
<!ATTLIST Probe           name          ID                      #REQUIRED
                          ID            NMTOKEN                 #REQUIRED
                          descr         CDATA                   #IMPLIED
>
            <!-- ID:           [0..255]                                    -->
...
<!ELEMENT PDUSaving       EMPTY >
<!ATTLIST PDUSaving       filepathname CDATA                    #REQUIRED
                          maxPDUSize   NMTOKEN                  "3000"
>
            <!-- filepathname:     Full pathname of file radix where to     -->
            <!--                   locally save the various types of probe  -->
            <!--                   observation PDU's                        -->
            <!--                      supports strftime strings (%H%M...)   -->
```

as in

```
   <Probe name="probe" ID="11" descr="tmon probe on delhi">
...
      <PDUSaving filepathname="&pduPath;/pdus.%y%m%d%H" maxPDUSize="2000"/>
   </Probe>
```

For each PDU type (except Heart Beat), a specific file is created with respective suffix:

```
.fldesc
.pktobs
.delobs
.flcntrs
.histdesc
.histdata
.tcpconns
.ftpxfers
```

These fiels encompass the exact binary dump of successive Probe protocol data units. A specific decoder must be implement to exploit them. Or the collector can be adapted to import those files (manually transferred to the central trafMon system) as an alternative to the online UDP PDU reception. But beware of the deferred time, then.

4.6 COLLECTOR OUTPUT LOG FILES

The collector program continually writes its output raw measurements to files that are typically buffered in a dedicated directory of a file system partition designed to store high volumes of archived data.

The template file pathname for those output files is given in the XML configuration as per

```
<!ELEMENT Collector        (Addr,Output) >
<!ATTLIST Collector        name       ID                              #REQUIRED
                           ID         NMTOKEN                         #REQUIRED
                           descr      CDATA                           #IMPLIED
                           burstRate  NMTOKEN                         "10"
>
           <!-- ID:           [256..]                                          -->
           <!-- burstRate:  how much PDU to expect in a burst                 -->
           <!--                       valid values are [10..65535]           -->
...
<!ELEMENT Output           EMPTY >
<!ATTLIST Output           dataFile     CDATA           "tmondata-%y%m%d%H%M%S"
                           eventFile    CDATA          "tmonevent-%y%m%d%H%M%S"
                           excepFile    CDATA          "tmonexcep-%y%m%d%H%M%S"
                           period       NMTOKEN                    #IMPLIED
>
           <!-- dataFile:         CSV-file radix name for data output      -->
           <!--                      supports strftime strings (%H%M...)   -->
           <!-- eventFile:        CSV-file radix name for event output     -->
           <!--                      supports strftime strings (%H%M...)   -->
           <!-- excepFile:        CSV-file radix name for exceptions output-->
           <!--                      supports strftime strings (%H%M...)   -->
           <!-- period:           number of minutes during which output    -->
           <!--                   accumulate in a same CVS-file            -->
           <!--                      ONLY VALID where strftime %M present  -->
           <!--                      Used as the modulo on the minute field-->
           <!--                    valid values are [1..59]                -->
```

as in

```
<!DOCTYPE TrafMonConfig SYSTEM "tmon.dtd" [
...
<!ENTITY dataPath "/var/trafMon/collector">
<!ENTITY pduPath "/data/pdu">
<!-- End of ENTITIES -->
]>

<trafMonConfig serial="100" startAt="2020-10-04 13:29:00" pktSignBytes="3"
  maxTravelTime="30000" >

  <Collector name="rho" ID="100" descr="trafMon collector on rho"
    burstRate="30">
    <Addr ip="&data.rho;" port="&tmonColl;" UDPBufferSize="20000"/>
    <Output dataFile="&dataPath;/observations.%y%m%d%H%M"
            eventFile="&dataPath;/events.%y%m%d%H%M"
            excepFile="&dataPath;/exceptions.%y%m%d%H%M"
            period="5"  />

  </Collector>
```

```
<!ELEMENT Measure          (Delay?, Stats?) >
<!ATTLIST Measure          interval     (each|10s|20s|30s
                                        |1min|10min|20min|30min
                                        |1h)                        #REQUIRED
>
```

The output file template pathname (may) contains reference to the date/time, typically at the minute. This means the every minute, a same type of output is directed to a new output log file.

One log file is created per type of output. Those files are TAB-separated and their fields directly correspond to the columns of MySQL input data tables, permitting efficient bulk load.

With a configured format template as:

```
  <Collector ... >
    ...
    <Output dataFile="/var/trafMon/observations.%y%m%d%H%M" ...  />
  </Collector>
```

the collector produces its output files, each covering records produced during the same minute, like:

`/var/trafMon/collector/observations.200141013T1413.`*SUFFIX*

The *SUFFIX* denotes the type of resulting data.

Two such types are definition items, derived from the XML configuration, which must be kept with the data they qualify: the *Metric Slices Definitions* and the *Flow Classes Hop Lists*.

All resulting data refer to the Flow Instance they pertain to (one of them when there are several). Flow Instances are dynamically discovered by the distributed probes. The Flow Instance Descriptions hold the values of those flow identifying items that are remembered according to the applicable <`GranularFlow`> specifications:

```
<!ELEMENT GranularFlow    (DistinctIf?,DistinctAddr?,DistinctPort?,GroupBy*) >
<!ATTLIST GranularFlow    name        ID                          #REQUIRED
>
<!ELEMENT DistinctIf      EMPTY >
            <!-- Packets seen at different Probe Interfaces lead to        -->
            <!-- instances of granular flow, even when they produce same   -->
            <!-- results for all other criteria.                           -->
            <!-- NOTE:                                                     -->
            <!--      This may NOT be used when matching BI-DIRECTIONAL    -->
            <!--      traffic flow on the basis of packets captured by a   -->
            <!--      PASSIVE TAP devices: each direction being seen by a  -->
            <!--      separate capture interface.                          -->
<!ELEMENT DistinctAddr    EMPTY >
<!ATTLIST DistinctAddr    field       (src|srcnet|dst|dstnet|srcdst|srcdstnet
                                       |addr|net|addrpair|netpair) #REQUIRED
                          mask        CDATA                       #IMPLIED
>
            <!-- field:  which fields to preserve in grouping measurements  -->
            <!--  a) UNI-DIRECTIONAL                                        -->
            <!--   src:      keep granularity per source IP address         -->
            <!--   srcnet:   keep granularity per src IP subnet: using mask -->
            <!--   dst:      keep granularity per destination IP address    -->
            <!--   dstnet:   keep granularity per dst IP subnet: using mask -->
            <!--   srcdst:   keep granularity per source/dest. IP addresses -->
            <!--   srcdstnet:keep granularity per src/dst IP subnets: mask  -->
```

```
            <!--  b) BI-DIRECTIONAL                                          -->
            <!--   addr:     keep granularity per IP address of 1 peer       -->
            <!--   net:      keep gran. per IP subnet of 1 peer: using mask  -->
            <!--   addrpair: keep granularity per pair of IP addresses       -->
            <!--   netpair:  keep granul. per pair of IP subnets: using mask-->
            <!--                                                             -->
            <!-- mask:   subnet mask: "xxx.xxx.xxx.xxx" or "/yy" notation    -->

<!ELEMENT DistinctPort    EMPTY >
<!ATTLIST DistinctPort    field       (sport|dport|sdport
                                       |port|portpair)           #REQUIRED
                          portspec    (alldistinct|privileged)  "alldistinct"
>
            <!-- field:  which fields to preserve in grouping measurements  -->
            <!--  a) UNI-DIRECTIONAL                                         -->
            <!--   sport:    keep granularity of source UDP/TCP port number -->
            <!--                    <=> any:port to any:any                  -->
            <!--   dport:    keep granularity of destin. UDP/TCP port number-->
            <!--                    <=> any:any to any:port                  -->
            <!--   sdport:   keep granularity of src/dst UDP/TCP prt numbers-->
            <!--                    <=> any:port1 to dst:port2               -->
            <!--    EITHER  without <DistinctAddr>                           -->
            <!--    OR ONLY with    <DistinctAddr field=(src|srcnet          -->
            <!--                                        and/or dst|dstnet)   -->
            <!--   port: EITHER  with <DistinctAddr field=(src|srcnet) >     -->
            <!--             same as sport:  <=> src:sport to any:any       -->
            <!--         OR      with <DistinctAddr field=(dst|dstnet) >    -->
            <!--             same as dport:  <=> any:any to dst:dport       -->
            <!--         OR      with <DistinctAddr field=(srcdst[net]) >   -->
            <!--             preserves smallest port number:                -->
            <!--               if sport <= dport                            -->
            <!--                  <=> src:sport to dst:any                  -->
            <!--               if sport >  dport                            -->
            <!--                  <=> src:any to dst:dport                  -->
            <!--   portpair: ONLY with    <DistinctAddr field=(src|srcnet   -->
            <!--                                        and/or dst|dstnet)   -->
            <!--             same as sdport: <=> src:sport to dst:dport     -->
            <!--  OTHER COMBINATIONS of <DistinctAddr>+<DistinctPort>       -->
            <!--                      ARE NOT ALLOWED (and meaningless)     -->
            <!--  b) BI-DIRECTIONAL                                          -->
            <!--   port: EITHER without <DistinctAddr>                       -->
            <!--             keep granul. of UDP/TCP port number of 1 peer  -->
            <!--               if sport <= dport                            -->
            <!--                  <=> any:sport to any:any                  -->
            <!--               if sport >  dport                            -->
            <!--                  <=> any:dport to any:any                  -->
            <!--         OR with <DistinctAddr field=(addr|net) >           -->
            <!--             keep granul.  addr:port  to/from   any:any     -->
            <!--                           net:port  to/from   any:any      -->
            <!--                              WHERE addr/net <= peer any    -->
            <!--         OR with <DistinctAddr field=(addrpair|netpair) >   -->
            <!--             keep granul. of address pair and smallest port:-->
            <!--               if port1 <= port2                            -->
            <!--                        addr1:port1 to/from addr2:any       -->
            <!--                         net1:port1 to/from  net2:any       -->
            <!--               if port1  > port2                            -->
            <!--                        addr1:any   to/from addr2:port2     -->
            <!--                         net1:any   to/from  net2:port2     -->
```

```
            <!--   portpair:                                                 -->
            <!--         EITHER without <DistinctAddr>                       -->
            <!--             keep granul. of both UDP/TCP port numbers       -->
            <!--                   <=> any:port1 to/from any:port2           -->
            <!--                                WHERE port1 <= port2         -->
            <!--         OR with <DistinctAddr field=(addr|net) >            -->
            <!--             keep granul.  addr:port1 to/from   any:port2    -->
            <!--                           net:port1 to/from   any:port2     -->
            <!--                              WHERE addr/net <= peer any     -->
            <!--         OR with <DistinctAddr field=(addrpair|netpair) >    -->
            <!--             keep granul. addr1:port1 to/from addr2:port2    -->
            <!--                          net1:port1 to/from  net2:port2     -->
            <!--                                                             -->
            <!-- portspec:                                                   -->
            <!--  *alldistinct:  keep all values distinct                    -->
            <!--   privileged:  distinguish all service ports<1024          -->
            <!--                BUT group all ports>=1024 (as 65535)         -->

<!ELEMENT GroupBy          EMPTY >
<!ATTLIST GroupBy          field       (ipsizes
                                       |ipproto|tos|df|mf|frag|ttl
                                       |icmp
                                       |tcptype)                  #REQUIRED
                           sizeclasses (per200|per400)            #IMPLIED
                           tosspec     (precedence|dscp|tosbyte)  #IMPLIED
                           fragspec    (fragnumber|fragoffset)    #IMPLIED
                           icmpspec    (icmpclass|icmptype
                                       |icmptypecode)             #IMPLIED
                           tcptypespec (byflags|byflagsandretran
                                       |S_D_A_E|S_D_A_E_R
                                       |S_F_R_A_E|S_F_R_A_E_R)    "S_D_A_E"
>
            <!--   ipsizes:  keep granularity per 'sizeclasses' of IP pkt   -->
            <!--   ipproto:  keep granularity per UDP|TCP|Other IP protocol -->
            <!--   tos:      keep granularity as per IP TypeOfSvc 'tosspec' -->
            <!--   df:       keep granularity per IP Don't Fragment flag    -->
            <!--   mf:       keep granularity per IP More Fragment flag     -->
            <!--   frag:     keep granularity as per IP Fragment 'fragspec' -->
            <!--   ttl:      keep granularity per IP Time-to-Live value     -->
            <!--   icmp:     keep granularity of ICMP pkts as per 'icmpspec'-->
            <!--   tcptype:  keep granularity as per 'tcptypespec' grouping -->
            <!--                                                             -->
            <!-- sizeclasses: groups IP packet sizes in buckets              -->
            <!--   per400:   4 buckets: boundaries 400, 800, 1200            -->
            <!--   per200:   8 buckets:     200,400,600,800,1000,1200,1400   -->
            <!-- BUT, for datagram cummulated IP sizes, sizes >= 1600        -->
            <!-- are grouped by thousands: 1600, 2000, 3000 ... 7000, 8000   -->
            <!--                                                             -->
            <!-- tosspec:                                                    -->
            <!--   precedence: per value of the three ToS precedence bits   -->
            <!--   dscp:       per value of the six DSCP bits               -->
            <!--   tosbyte:    per distinct values of the complete ToS byte -->
            <!--                                                             -->
            <!-- fragspec:                                                   -->
            <!--   fragnumber: per ordinal number of the fragment            -->
            <!--   fragoffset: per value of the fragment offset              -->
            <!--                                                             -->
            <!-- icmpspec:                                                   -->
```

```
            <!--   icmpclass:    group as per Echo | Error | Info | Other   -->
            <!--   icmptype:     per value of the ICMP Type byte            -->
            <!--   icmptypecode: per value of the ICMP Type and Code bytes  -->
            <!--                                                            -->
            <!-- tcptypespec: distinguish                                   -->
            <!--   byflags:      per distinct values of the TCP flags byte  -->
            <!--   byflagsandretran: idem, but also distinguish between     -->
            <!--                 first and subsequent transmissions         -->
            <!--                 of a not empty data segment                -->
            <!--  *S_D_A_E:      Start (syn/syn-ack),                       -->
            <!--                 Data (not empty payload)                   -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack/reset)                    -->
            <!--   S_D_A_E_R:    Start (syn/syn-ack),                       -->
            <!--                 Data (not empty payload)                   -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack)                          -->
            <!--                 RESET (rst flag)                           -->
            <!--   S_F_R_A_E:    Start (syn/syn-ack),                       -->
            <!--                 FIRST transmission of data segment         -->
            <!--                 RETRANSMISSION of data segment             -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack/reset)                    -->
            <!--   S_F_R_A_E_R:  Start (syn/syn-ack),                       -->
            <!--                 FIRST transmission of data segment         -->
            <!--                 RETRANSMISSION of data segment             -->
            <!--                 Ack (ack flag, but no payload)             -->
            <!--                 End (fin/fin-ack)                          -->
            <!--                 RESET (rst flag)                           -->
```

The mapping between measurement data and the GranularFlow comes from the configured <FlowGrain> reference at the level of the Flow Class or of its <Filter> specifications:

```
<!ELEMENT FlowClass       (Measure, FlowGrain?, Filter+, Condition?)>
<!ATTLIST FlowClass       id            NMTOKEN                  #REQUIRED
                          name          NMTOKEN                  #REQUIRED
                          descr         CDATA                    #IMPLIED
>
<!ELEMENT Filter          (On+, CaptureTimeStamp?,
                                (IpOptTimestamp | NtpTimestamp)*, NatPat*,
                                                  PacketExpr, FlowGrain?)>
<!ELEMENT On              EMPTY >
<!ATTLIST On              probe         NMTOKEN                  #REQUIRED
                          if            NMTOKEN                  #REQUIRED
>
<!ELEMENT FlowGrain       EMPTY >
<!ATTLIST FlowGrain       ref           IDREF                    #REQUIRED
>
```

When the <GranularFlow> contains the <DistinctIf/> tag, the resulting Flow Instannces will be specific per probing point (instance of probe interface). In such case, observations of a same traffic flow (same packets) at different probing points (e.g. at peer end sides) won't be abusively merged.

But for <OneWayDelay> measurement at **granularity**="individual", the applicable <GranularFlow> specification must be such that packets seen at source and at destination (and at intermediate hops over the network travel path) at assigned to the same Flow Instance (hence avoidance of <DistinctIf/> tag for this case).

Examples:

```
  <GranularFlow name="protoConversAtProbeIf" >
    <DistinctIf /> <!-- mandatory when Counters, to avoid double records -->
    <DistinctAddr field="addrpair" />
    <DistinctPort field="portpair" portspec="privileged" />
    <GroupBy      field="ipproto"/>
  </GranularFlow>

...


  <!-- FTP: TCP port 21
  -->
  <FlowClass id="21" name="FTP_port21" descr="TCP with port==21">
    <Measure interval="1min" >
      <Stats verifChksum="bestEffort">
        <PacketCounters for="firstFragment"/>
                          <!-- Don't ask for Dgram for TCP to avoid unnecessary
                               keeping of subsequent frags (of other flows)
                               between same IP address pair -->
        <TCPConnections granularity="each"/>
        <FileTransfers  protocol="FTP" granularity="each"
                        ftpdata="start-stop"/>
      </Stats>
    </Measure>
    <FlowGrain ref="protoConversAtProbeIf" />
    <Filter>
      <On probe="hutch" if="eth1" />
      <On probe="hutch" if="eth2" />
      <On probe="chieti" if="eth1" />
      <On probe="chieti" if="eth2" />
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="tcp"/>
          <Predicate field="port"  op="eq" value="21"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
```

or

```
  <GranularFlow name="peers" > <!-- NO DistinctIf for oneWay partial obs -->
    <DistinctAddr field="addrpair" />
  </GranularFlow>

...

  <FlowClass id="12345" name="withIPTS"
    descr="UDP with port==21 and IP Timestamps">
```

```
    <Measure interval="each" >
      <Delay for="datagram" granularity="individual">
        <OneWayDelay>
          <Hop name="remint1"/>
          <Hop name="remint"/>
          <Hop name="alkuf"/>
          <Hop name="hiros"/>
          <Hop name="kuching"/>
          <Hop name="locdmz"/>
          <Sign>
            <Chunk start="0" relTo="ipPayload" length="40" />
          </Sign>
        </OneWayDelay>
      </Delay>
    </Measure>
    <FlowGrain ref="peers" />

    <Filter>
      <On probe="chieti" if="eth2" />
      <CaptureTimeStamp           hopName="remint"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="21"/>
        </AND>
      </PacketExpr>
    </Filter>

    <Filter>
      <On probe="hutch" if="eth1" />
      <CaptureTimeStamp           hopName="locdmz"/>
      <IpOptTimestamp ipTSNum="1" hopName="remint1"/>
      <IpOptTimestamp ipTSNum="2" hopName="alkuf"/>
      <IpOptTimestamp ipTSNum="3" hopName="hiros"/>
      <IpOptTimestamp ipTSNum="4" hopName="kuching"/>
      <PacketExpr>
        <AND>
          <Predicate field="proto" op="eq" value="udp"/>
          <Predicate field="port"  op="eq" value="21"/>
        </AND>
      </PacketExpr>
    </Filter>
  </FlowClass>
```

4.6.1 Flow Description Log

SUFFIX: `".flow"`

CONTENT:

```
flowID | address1 | address2 | port1 | port2 | protocol | direction | size |
ToSType | ToSValue | TimeToLive | DontFragment | MoreFragment | fragmentNumber |
fragmentOffset | icmpType | tcpType | probeInterface | comment
```

Example:

```
...
14131945711825 | '141.253.245.241' | '141.253.245.248' | 123 | 123 | 'udp' | '='
| \N | \N | \N | \N | \N | \N | \N | \N | \N | 'chieti:eth1' | 'REMOTE DMZ'
...
```

Explanations:

The notation “\N” means NULL (no value given for this field). In this context of Flow Instance, it means that the flow covers packets that have any value for corresponding field.

Direction is either

- ‘=’: encompasses both directions
- ‘>’: covers corresponding packets from `address1` to `address2`
- ‘<’: covers corresponding packets from `address2` to `address1`

Address pairs are ordered: `address1` <= `address2`, by comparing the numeric long word (32bit value of IPv4 address)

4.6.2 Flow IP Counters Log

SUFFIX: `".ipct"`

CONTENT:

```
flowID | timestamp | interval | perDatagram | totalBytes | sizeBucketCount |
probeReassemblyTimeout | probeFragmentOverlap | icmpCount | udpCount | tcpCount
| otherProtocolCount
```

Example:

```
14131941405299 | '2020-10-13 10:04:16' | 60 | 'f' | 3174 | 8 | 0 | 0 | 0 | 0 |
45 | 0
14131941155130 | '2020-10-13 10:04:16' | 60 | 'f' | 24782 | 8 | 0 | 0 | 0 | 0 |
359 | 0
14131941405292 | '2020-10-13 10:04:16' | 60 | 'f' | 17868 | 8 | 0 | 0 | 0 | 55 |
0 | 0
14131942367510 | '2020-10-13 10:04:09' | 60 | 'f' | 316 | 8 | 0 | 0 | 2 | 2 | 0
| 0
14131942367507 | '2020-10-13 10:04:09' | 60 | 't' | 316 | 16 | 0 | 0 | 2 | 2 | 0
| 0
```

Explanations:

All these samples cover 60s (1 minute) of IP packet counting. Although the time period is configurable on a per Flow Class measurement, in order to manage the time aggregates in a systematic way inside the database, it has been necessary to fix a conventional period of 1 minute for all packet counters. Each sample refers to its flow instance record (see 4.6.1 above).

The three first cover the range 2020-10-13 [10:04:16 .. 10:04:17[, the last two cover the range [10:04:09 .. 10:04:10[.

The first five records provides IP counters at the level of individually captured IPv4 packets, while the last counts only per complete datagram (either un-fragmented IPv4 packets or reassembled datagram units inside the probe). Note that, in the case of EO network, fragmentation has only been observed for some SNMP response packets; TCP is never fragmented (MSS < MTU over the path); most of UDP packets are small requests/replies of network services (DNS, NTP, most of SNMP); ICMP packets are, by definition, short packets and are never subject to fragmentation.

4.6.3 Flow IP Sizes Distribution Log

SUFFIX: `".ipsz"`

CONTENT:

```
flowID | timestamp | interval | perDatagram | lower | upper | minimum | maximum
| average | population | sum | sumOfSquares
```

Example:

```
...
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 0 | 200 | 0 | 0 | 0 | 0 | 0
| 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 200 | 400 | 0 | 0 | 0 | 0 |
0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 400 | 600 | 0 | 0 | 0 | 0 |
0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 600 | 800 | 0 | 0 | 0 | 0 |
0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 800 | 1000 | 0 | 0 | 0 | 0 |
0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 1000 | 1200 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 1200 | 1400 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 1400 | 1600 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 1600 | 2000 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 2000 | 3000 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 3000 | 4000 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 4000 | 5000 | 4128 | 4128 |
4128 | 149432 | 616855296 | 2546378661888
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 5000 | 6000 | 0 | 0 | 0 | 0
| 0 | 0
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 6000 | 7000 | 6240 | 6999 |
6383 | 1157 | 7385935 | 47206122891
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 7000 | 8000 | 7002 | 7059 |
7028 | 29 | 203840 | 1432792512
14131941405294 | '2020-10-13 22:45:56' | 60 | 't' | 8000 | 65535 | 29472 | 53249
| 49801 | 2375 | 118279137 | 18446740757931516007
...
```

Explanations:

This file provides the slices of histogram distribution of observed packet of datagram IP sizes for several flows.

Here above are all slices of the size distribution **reassembled IPv4 datagram units**. The step over the histogram slices is voluntarily not constant: 200 bytes up to the largest Ethernet MTU (1500), then 400 bytes between 1600 and 2000, then 1000 bytes for larger datagram units.

When only per IP packet measurement is reported, the first part of the histogram is provided, with a constant step=200 bytes up to 1400. And the topmost slice is not bounded: 1400 bytes-long and above.

NOTE: currently all slices are reported, even those without observed packet. In the future, those empty slices would be skipped to preserve database memory space.

As for the counters, the period is conventionally fixed as 1 minute.

The **average is sum/population**. It could be re-computed inside the database. It is given as an integer, although a fractional value should be kept in the database.

In order to permit computation of the standard deviation (sigma) for estimating the 95 percentile (avg +/- 2sigma), the **sum of squares** is given. Over the 1 minute period, this number still fits in a big integer (64 bit), although in the database it must be stored as a double fractional value preserving the most significant digits and would drop those below meaningful granularity when the value becomes too high.

$$\sigma = \sqrt{\frac{(population \cdot sum\_sq) - (sum \cdot sum)}{population \cdot (population - 1)}}$$

4.6.4 Flow ICMP Counters Log

SUFFIX: `".icmpct"`

CONTENT:

```
flowID | timestamp | interval | probeChecksumFailures | probeChecksumSkipped |
echoRequests | echoReplies | fragmentationNeeded | sourceQuench |
timeToLiveExpired | reassemblyTimeout | unReached | redirect | otherIcmpErrors |
otherIcmpInfoPackets
```

Example:

```
...
14131942227241 | '2020-10-13 10:01:56' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1
| 0 | 0 | 0
14131944689226 | '2020-10-13 10:01:56' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6
| 0 | 0 | 0
14131944689224 | '2020-10-13 10:01:56' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6
| 0 | 0 | 0
14131941665888 | '2020-10-13 10:01:59' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8
| 0 | 0 | 0
14131941665885 | '2020-10-13 10:01:59' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8
| 0 | 0 | 0
```

```
14131944589106 | '2020-10-13 10:01:59' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6
| 0 | 0 | 0
14131941665886 | '2020-10-13 10:01:59' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8
| 0 | 0 | 0
14131941665883 | '2020-10-13 10:01:59' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8
| 0 | 0 | 0
...
```

4.6.5 Flow UDP Counters Log

SUFFIX: `".udpct"`

CONTENT:

```
flowID | timestamp | interval | probeChecksumFailures | probeChecksumSkipped |
emptyDatagrams | snmpCount | dnsCount | ntpCount | otherServiceCount
```

Example:

```
14131941665877 | '2020-10-13 10:02:49' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 4
14131942227241 | '2020-10-13 10:02:56' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 8
14131942227239 | '2020-10-13 10:02:56' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 8
14131942367510 | '2020-10-13 10:03:09' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 10
14131942367507 | '2020-10-13 10:03:09' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 10
14131941665868 | '2020-10-13 10:03:19' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 12
14131941665854 | '2020-10-13 10:03:19' | 60 | 0 | 201 | 0 | 0 | 0 | 0 | 201
14131941405292 | '2020-10-13 10:03:16' | 60 | 0 | 21 | 0 | 0 | 0 | 0 | 59
14131941405296 | '2020-10-13 10:03:16' | 60 | 0 | 200 | 0 | 0 | 0 | 0 | 200
```

Explanations:

The **`probeChecksumFailures`** counts those datagrams whose UDP checksum has actually been verified wrong by the probe (either **`verifChksum`**=`"bestEffort"` and UDP packets are not fragmented, or **`verifChksum`**=`"fullReassembly"` and the probe verified the checksum of the reassembled UDP datagram).

The **`probeChecksumSkipped`** counts those datagrams whose UDP checksum has not been verified by the probe (**`verifChksum`**=`"bestEffort"` and UDP packets are fragmented and not reassembled by the probe).

An **`emptyDatagram`** is a UDP datagram without payload: a packet consisting only in a IPv4 header and a UDP header.

4.6.6 Flow TCP Counters Log

SUFFIX: `".tcpct"`

CONTENT:

```
flowID | timestamp | interval | probeChecksumFailures | probeChecksumSkipped |
restranmits | latePackets | connectionStartCount | connectionCleanCloseCount |
```

```
connectionDirtyCloseCount | ftpControlConnections | ftpFileTransfers |
httpFileTransfers | otherConnections
```

Example:

```
14131941755937 | '2020-10-13 09:59:06' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 3
| 0 | 0
14131941755939 | '2020-10-13 09:59:06' | 60 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 3
| 0 | 0
14131941405299 | '2020-10-13 09:59:16' | 60 | 0 | 0 | 1 | 13 | 4 | 3 | 2 | 10 |
0 | 0 | 0
14131941155130 | '2020-10-13 09:59:16' | 60 | 0 | 0 | 0 | 2 | 4 | 1 | 2 | 8 | 0
| 0 | 0
14131941665849 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 4 | 0
| 0 | 0
14131941816012 | '2020-10-13 09:59:16' | 60 | 0 | 0 | 0 | 2 | 0 | 3 | 3 | 1 | 0
| 0 | 0
14131941405302 | '2020-10-13 09:59:16' | 60 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 4 | 0
| 0 | 0
14131941665865 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 0
| 0 | 0
14131941135125 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 5 | 4 | 5 | 2 | 11 | 0
| 0 | 0
14131941665844 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 7 | 4 | 0 | 1 | 8 | 0
| 0 | 0
14131941665870 | '2020-10-13 09:59:19' | 60 | 0 | 165 | 0 | 0 | 0 | 1 | 0 | 0 |
0 | 0 | 0
14131941665863 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 2 | 0
| 0 | 0
14131941765958 | '2020-10-13 09:59:19' | 60 | 0 | 0 | 0 | 15 | 0 | 0 | 3 | 1 | 0
| 0 | 0
```

Explanations:

The **`restranmits`** counts those packets (never fragmented in practice) that transport a TCP segment which is detected by the probe (through its optional <TCPConnections> stateful analysis) as being a retransmission.

The **`latePackets`** counts those packets seen by the probe after the TCP connection started its closing stage: after 500 ms for a non-retransmitted FIN, 5 ms for a non retransmitted RST of for a DATA segment.

The **`connectionStartCount`** counts those TCP connections that are starting during the reported time interval.

The **`connectionCleanCloseCount`** counts those TCP connections that are properly closing over the reported interval: seen FIN+ACK response to FIN.

The **`connectionDirtyCloseCount`** counts those TCP connections that are not cleanly closed during the reported time interval: First RESET over the (open or closing) connection,

The **`ftpControlConnections`** counts the number of FTP control connections that are established during the reported time interval.

The **ftpFileTransfers** counts the number of succeeded or accepted-but-failed GET or PUT transfers concluded during the time interval.

The **httpFileTransfers** is currently always 0 (not implemented).

The **otherConnections** counts the number of TCP connection first seen during this time interval, and which are neither FTP control nor FTP File Transfer data connection (nor HTTP file transfer, when implemented) – but directory listing FTP data connection are counted as part of others.

4.6.7 Flow FTP Counters Log

SUFFIX: ".ftpct"

CONTENT:

```
flowID | timestamp | interval | startedSessions | cleanClosedSessions |
dirtyClosedSessions | encryptedSessions | noLoginSessions | noCmdSession |
noFileXferSessions | fileXferSessions | activeConnections | passiveConnections |
dirListCount | fileGetOK | filePutOK | fileGetFailures | filePutFailures |
xferRestarts | xferAborts | failedLogins | cipherFailures | commandFailures
```

Example:

```
14131941755937 | '2020-10-13 09:59:06' | 60 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1
| 1 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0
14131941755939 | '2020-10-13 09:59:06' | 60 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1
| 1 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0
14131941405299 | '2020-10-13 09:59:16' | 60 | 10 | 0 | 5 | 0 | 4 | 1 | 0 | 0 | 1
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14
14131941155130 | '2020-10-13 09:59:16' | 60 | 8 | 0 | 3 | 0 | 2 | 1 | 0 | 0 | 0
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 19
14131941665849 | '2020-10-13 09:59:19' | 60 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9
14131941816012 | '2020-10-13 09:59:16' | 60 | 1 | 0 | 4 | 0 | 1 | 3 | 0 | 0 | 0
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16
14131941405302 | '2020-10-13 09:59:16' | 60 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11
14131941665865 | '2020-10-13 09:59:19' | 60 | 2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
14131941135125 | '2020-10-13 09:59:19' | 60 | 11 | 0 | 5 | 0 | 4 | 1 | 0 | 0 | 1
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14
14131941665844 | '2020-10-13 09:59:19' | 60 | 8 | 0 | 3 | 0 | 2 | 1 | 0 | 0 | 0
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 19
```

Explanations:

The **cleanClosedSessions** counts those FTP control sessions that are cleanly ended by the client sending the QUIT command. Inversely **dirtyClosedSessions** counts those FTP control sessions that are terminated abruptly by the client closing the connection. This behaviour causes the FTP server to react and send back an “Oops”

message rejected by the client which responds with a RESET. Few late packets are then observed after the connection times-out and can finally be considered as closed.

The **encryptedSessions** counts those which start with an AUTH command and typically continue in a ciphered SSL/TLS connection whose state and command/response exchanges can't be continued by the probe. **For those, the actual transfer of files can't be monitored**.

The **noLoginSessions** counts those empty FTP control sessions where, after the connection is established, correct logon never happens. These could result from FTP server availability polling.

The **noCmdSession** counts those empty FTP control sessions over which, after the user has logged-on, no other real FTP command is issued. These could result from FTP server availability polling.

The **noFileXferSessions** counts those active FTP control sessions over which real FTP command(s) is(are) exchanged, but no file is actually transferred. These could result from FTP server file system polling: e.g. no new file spooled of interest by the client.

The **fileXferSessions** counts those active FTP control sessions over which at least one file transfer is attempted.

The **activeConnections** counts those FTP data connections that are established in active mode (through FTP PORT command or equivalent): **data connection from the server back to the client**. This covers both the directory listing results and the actual file transfers.

The **passiveConnections** counts those FTP data connections that are established in passive mode (through FTP PASV command or equivalent): **data connection from the client to the server**. This covers both the directory listing results and the actual file transfers.

The **dirListCount** counts those FTP data connections that are established for transferring the results of a directory listing request.

The **fileGetOK** counts those successful transfers of files from the server to the client.

The **filePutOK** counts those successful transfers of files from the client to the server.

The **fileGetFailures** counts those transfers of files from the server to the client ending in the server finally responding with a negative completion message (error code starting with '5').

The **filePutFailures** counts those transfers of files from the client to the server ending in the server finally responding with a negative completion message (error code starting with '5').

The **xferRestarts** counts the number of file transfers that are restarted (skipping the first part – supposedly already successfully transferred during a previous failed attempt – up to a given offset).

The **xferAborts** counts the number of time an FTP client has aborted a file transfer in progress.

The **failedLogins** counts the number of failed attempts of logon by a client over an FTP control session established with a server.

The **cipherFailures** counts the number of AUTH command by a client to which the server replied with an error message.

The **commandFailures** counts the number of negative responses send by FTP servers to commands requested by the clients.

4.6.8 Flow TCP Connections Log

SUFFIX: ".tcpcon"

CONTENT:

```
flowID | firstSeenTime | addressA | portA | addressB | portB | state | initiator
| terminator | reset | tcpOptions | probeInterface | interfaceDescription |
segmentsAB | ipBytesAB | payloadBytesAB | firstSegmentsAB |
firstSegmentsPayloadAB | retransmittedSegmentsAB | retransmittedPayloadBytesAB |
emptyAckAB | wouldAckNextAB | firstWindowAB | lastWindowAB | maxWindowAB |
lastSeenAB | durationAB | segmentsBA | ipBytesBA | payloadBytesBA |
firstSegmentsBA | firstSegmentsPayloadBA | retransmittedSegmentsBA |
retransmittedPayloadBytesBA | emptyAckBA | wouldAckNextBA | firstWindowBA |
lastWindowBA | maxWindowBA | lastSeenBA | durationBA
```

Example:

```
14134675449676 | '2020-101 08:50:41' | '141.253.221.100' | 20 | '141.253.11.100'
| 60090 | 'CLOSED' | 'A' | 'B' | 'no' | 'winScale+tcpRTTM+mssA>B+mssB>A' |
'hutch:eth1' | 'LOCAL DMZ' | 2 | 72 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 14480 | 14496
| 14496 | '2020-101 08:50:42' | 1 | 2 | 872 | 800 | 1 | 800 | 0 | 0 | 0 |
1048578 | 14600 | 14600 | 14600 | '2020-101 08:50:42' | 1
14134675449677 | '2020-101 08:50:41' | '141.253.221.100' | 20 | '141.253.11.100'
| 60090 | 'CLOSED' | 'A' | 'B' | 'no' | 'winScale+tcpRTTM+mssA>B+mssB>A' |
'hutch:eth2' | 'LOCAL Internal LAN' | 2 | 72 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 14480
| 14496 | 14496 | '2020-101 08:50:42' | 1 | 2 | 872 | 800 | 1 | 800 | 0 | 0 | 0
| 1048578 | 14600 | 14600 | 14600 | '2020-101 08:50:42' | 1
14134675449676 | '2020-101 08:50:42' | '141.253.221.100' | 32688 |
'141.253.11.100' | 34299 | 'CLOSED' | 'B' | 'B' | 'no' |
'winScale+tcpRTTM+mssA>B+mssB>A' | 'hutch:eth1' | 'LOCAL DMZ' | 2 | 72 | 0 | 0 |
0 | 0 | 0 | 0 | 1 | 14600 | 14624 | 14624 | '2020-101 08:50:42' | 0 | 2 | 912 |
840 | 1 | 840 | 0 | 0 | 0 | 1048578 | 14480 | 14480 | 14480 | '2020-101
08:50:42' | 0
14134675449677 | '2020-101 08:50:42' | '141.253.221.100' | 32688 |
'141.253.11.100' | 34299 | 'CLOSED' | 'B' | 'B' | 'no' |
'winScale+tcpRTTM+mssA>B+mssB>A' | 'hutch:eth2' | 'LOCAL Internal LAN' | 2 | 72
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 14600 | 14624 | 14624 | '2020-101 08:50:42' | 0 |
2 | 912 | 840 | 1 | 840 | 0 | 0 | 0 | 1048578 | 14480 | 14480 | 14480 | '2020-
101 08:50:42' | 0
```

```
14134675449676 | '2020-101 08:45:39' | '141.253.221.100' | 21 | '141.253.11.100'
| 44532 | 'CLOSED' | 'B' | 'A' | 'no' | 'winScale+tcpRTTM+mssA>B+mssB>A' |
'hutch:eth1' | 'LOCAL DMZ' | 30 | 1640 | 672 | 21 | 672 | 0 | 0 | 6 | 674 |
14600 | 14624 | 14624 | '2020-101 08:55:42' | 603 | 26 | 1094 | 254 | 17 | 254 |
0 | 0 | 7 | 256 | 14480 | 14496 | 14496 | '2020-101 08:55:42' | 603
14134675449677 | '2020-101 08:45:39' | '141.253.221.100' | 21 | '141.253.11.100'
| 44532 | 'CLOSED' | 'B' | 'A' | 'no' | 'winScale+tcpRTTM+mssA>B+mssB>A' |
'hutch:eth2' | 'LOCAL Internal LAN' | 30 | 1640 | 672 | 21 | 672 | 0 | 0 | 6 |
674 | 14600 | 14624 | 14624 | '2020-101 08:55:42' | 603 | 26 | 1094 | 254 | 17 |
254 | 0 | 0 | 7 | 256 | 14480 | 14496 | 14496 | '2020-101 08:55:42' | 603
```

In the above example, one can see three TCP connections that are simultaneously observed by two different probe interfaces, respectively `'hutch:eth1'` on `'LOCAL DMZ'` and `'hutch:eth2'` on `'LOCAL Internal LAN'`.

Explanations:

In fact the two above are FTP successive FTP data connections within the same FTP control session depicted by the TCP connection below. Data connection present of course an asymmetric profile: actual data are traversing only ine one direction. The FTP session is more balanced: made of client commands in one direction and server response in reverse direction.

The **`firstSeenTime`** is normally that of the initiating SYN packet, unless the probe (re)started thereafter.

The ordering A/B is such the IPv4 addressA < addressB or, for same addresses, portA < portB.

When the reporting is only at end (<Measure> **`interval`**=`"each"`), the **`state`** is either '**CLOSED'** or, after a timeout TMTCP_CONN_TIME_WAIT==30s **'FIN'**. Otherwise it could reflect an intermediate state among:

- 'SYN' (improbable): connection requested by one peer, but not yet agreed by the other;
- DATA (realistic): the connection is in progress, after establishment stage and before closing stage;
- FIN (possible): one side as released the connection (or reset or both) but the other side hasn't (yet) responded to this.
- CLOSED: both ends have sent their FIN and/or RST packet.

The **`initiator`** is 'A' or 'B' or, when SYN packet not observed: \N for NULL.

The **`terminator`** is 'A' or 'B' or, when FIN packet not yet observed: \N for NULL.

The **`reset`** is 'no' when no RST packet is (yet) observed on the connection. Or it indicates which side has send a RST: 'A' or 'B' or 'A+B'.

The **`tcpOptions`** is either '\N' for NULL, or a string of one or more keywords, separated by a + sign: winScale, tcpRTTM, mssA>B, mssB>A, sack (or sackAnotB or sackBtoA).

Unlike the case of FTP file transfer (below), the TCP **payloadBytesAB** (and BA) are counted by summing the TCP payload sizes of every inspected packet relative to the corresponding TCP connection and direction.

The **firstSegments** are those not detected, by the probe, as retransmission: Their TCP payload chunks are above the current **wouldAckNext**. Also counts the first SYN and FIN or RST.

On the contrary, the **retransmittedSegments** are those whose TCP payload chunks are either below the **wouldAckNext**, or inside an optional selective Acknowledge range prior sent by the destination peer. Also counts duplicated SYN and FIN or RST.

The **emptyAck** counts the number of TCP packet containing an ACK flag, but with no TCP payload data. These are typically sent as unique traffic in the reverse direction of a data transfer.

The **wouldAckNext** indicates the first byte, in consecutive stream order since the start of connection, that isn't yet acknowledged by the destination peer. In this ordering, the first SYN counts for first (pseudo) data byte: depending on direction, **wouldAckNext** is therefore the amount of contiguously received data stream bytes +1 or +2.

The **firstWindow**, **lastWindow** and **maxWindow** take into account the optional window multiplier agreed at connection start (provided this is observed by the probe). This gives an indication whether the TCP connection suffered from poor performances or if the widow size could be enlarged enough during its progress for taking party of a reliable communication link.

Note:

The following behaviour has been detected at end of some FTP connections: one end in ending a connection by a FIN. The other end does not properly end the connection from its side. After a long while a RST packet is finally sent. Although the connection remembering timeout, TMTCP_CONN_TIME_WAIT has been set to a long period of 30s of inactivity, it happens that the RST occurs after the probe has timed-out the connection and cleaned out its record after last report. This late RST is therefore seen a sign of not yet discovered TCP connection. This leads to the creation of a new record for a TCP connection that

- consists only in one (maybe a few) packet(s), only in one direction;
- as no packet is observed in the reverse direction, its last seen time and duration are NULL

4.6.9 Flow FTP File Transfers Log

SUFFIX: `".ftpxfr"`

CONTENT:

```
flowID | firstSeenTime | duration | ctlSessionTime | clientAddress |
clientDataPort | clientControlPort | serverAddress | serverDataPort |
serverControlPort | fileDirection | fileName | workDir | skippedFileOffset |
```

```
fileSize | payloadBytes | transferType | transferMode | connectionMode |
userName | connectionState | probeInterface | interfaceDescription
```

Example:

```
14131941135125 | '2020-10-13 10:23:58' | 282 | '2020-10-13 10:08:38' |
'141.253.245.248' | 57886 | 39344 | '141.253.221.100' | 20 | 21 | 'PUT' |
'transferred02.tar0.tar' | '"/testdir"' | 0 | 0 | 1048576 | 'BINARY' | \N |
'Active' | 'ftptest2' | 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941135125 | '2020-10-13 10:25:35' | 184 | '2020-10-13 10:12:25' |
'141.253.245.248' | 38653 | 39366 | '141.253.221.100' | 10065 | 21 | 'PUT' |
'transferred01.tar0.tar' | '"/testdir"' | 0 | 0 | 1048576 | 'BINARY' | \N |
'Passive' | 'ftptest2' | 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941665865 | '2020-10-13 10:28:44' | 1 | '2020-10-13 10:27:11' |
'141.253.11.100' | 52180 | 46767 | '141.253.221.100' | 20 | 21 | 'PUT' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 1048576 | 0 | 1048576 | 'BINARY' | \N |
'Active' | 'ftptest0' | 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941665863 | '2020-10-13 10:28:44' | 1 | '2020-10-13 10:27:11' |
'141.253.11.100' | 52180 | 46767 | '141.253.221.100' | 20 | 21 | 'PUT' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 1048576 | 0 | 1048576 | 'BINARY' | \N |
'Active' | 'ftptest0' | 'CLOSED' | 'hutch:eth2' | 'LOCAL Internal LAN'
14131941665865 | '2020-10-13 10:28:45' | 0 | '2020-10-13 10:27:11' |
'141.253.11.100' | 53468 | 46767 | '141.253.221.100' | 20 | 21 | 'GET' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 0 | 3145728 | 3145728 | 'BINARY' | \N |
'Active' | 'ftptest0' | 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941665863 | '2020-10-13 10:28:45' | 0 | '2020-10-13 10:27:11' |
'141.253.11.100' | 53468 | 46767 | '141.253.221.100' | 20 | 21 | 'GET' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 0 | 3145728 | 3145728 | 'BINARY' | \N |
'Active' | 'ftptest0' | 'CLOSED' | 'hutch:eth2' | 'LOCAL Internal LAN'
14131941665865 | '2020-10-13 10:28:45' | 0 | '2020-10-13 10:27:11' |
'141.253.11.100' | 58580 | 46767 | '141.253.221.100' | 54995 | 21 | 'GET' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 20 | 3145728 | 3145708 | 'BINARY' | \N |
'Passive' | 'ftptest0' | 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941135125 | '2020-10-13 10:00:55' | 81 | '2020-10-13 10:00:37' |
'141.253.245.248' | 39922 | 39255 | '141.253.221.100' | 28879 | 21 | 'PUT' |
'transferred01.tar0.tar' | '' | 0 | 0 | 1048576 | 'BINARY' | \N | 'Passive' | ''
| 'CLOSED' | 'hutch:eth1' | 'LOCAL DMZ'
14131941665863 | '2020-10-13 10:27:21' | 83 | '2020-10-13 10:27:11' |
'141.253.11.100' | 44638 | 46767 | '141.253.221.100' | 41696 | 21 | 'PUT' |
'long0long1long2long3long4long5long6long7long8long9long10long11long12long13name'
|
'tmpdir0/tmpdir1/tmpdir2/tmpdir3/tmpdir4/tmpdir5/tmpdir6/tmpdir7/tmpdir8/tmpdir9
```

```
/tmpdir10/tmpdir11/tmpdir12/tmpdir13' | 0 | 0 | 1048578 | 'ASCII' | \N |
'Passive' | 'ftptest0' | 'CLOSED' | 'hutch:eth2' | 'LOCAL Internal LAN'
```

Explanations:

The FTP file transfer record is mapped to its (primary) flow instance via **flowID**, as well as to the corresponding Data TCP Connection record (via **firstSeenTime + clientAddress + clientDataPort + serverAddress + serverDataPort**) and also to the FTP session (TCP control connection) via **ctlSessionTime + clientAddress + clientControlPort + serverAddress + serverControlPort**.

The **fileName** is expressed exactly as the client mentions it (pure basename, relative or absolute pathname)

During the FTP session, the user's navigation is tracked in such a way the trafMon probe attempts to continuously remember the position of the server-side current working directory. When the resulting **workDir** pathname is too long, the start and end of the string is preserved and a '...' is inserted.

When the transfer is a restart, the **skippedFileOffset** is non zero.

When discovered as a result of a SIZE command, or displayed in a server message, the actual **fileSize** on server file system is mentioned (otherwise zero).

The **payloadBytes** presents the number of data bytes that are transferred, counting possible retransmissions. This value can be obtained in two different ways:

- Either `<FileTransfers ftpdata="full" >`: each TCP data segment is inspected and its payload size is summed. The resulting value is the exact amount of file bytes transferred to the peer (incl. retransmissions), but at the expense of inspecting numerous packets!
- Or `<FileTransfers ftpdata="start-stop" >`: this proceeds heuristically by only inspecting the SYN and FIN packets, which significantly alleviates the work of the probe (intermediate TCP packets being reject in the probe father process, just after TCP dissection). By subtracting the acknowledge value of reverse FIN packet from that of the reverse SYN, one obtains (1+transferred **payloadBytes**) **modulo 4GiBytes** – the SYN counts for one byte of payload, the acknowledge value is truncated to 32bit, hence to 4294967295. Unfortunately, this very efficient heuristics shouldn't be used when monitoring transfer of files potentially larger than 4GBytes.

The **transferType** is one of

  - 'BINARY': the file content is transferred as-is.
  - 'ASCII': translation of end-of-line character(s) and maybe others is attempted to adapt to the diverging conventions of operating systems of client and server (note that this is quite often the source of problems).
  - Never used in practice: 'EBCDIC' and 'LOCAL'.

The **transferMode** is in practice always NULL as this is never explicitly negociated over the FTP control session. In fact there is only one default mode used: Stream (the file is a stream of bytes). Other theoretical possible values are Block or CompressedBlock.

The **connectionMode**:

- 'Active' – the default mode (although less recommended due to potential security barrier): the **data connection is established by the server** (typically using TCP port 20) to the client at a port number dynamically transmitted via the PORT command (or equivalent).
- 'Passive: the **data connection is established by the client** (as for the control connection) to the server at a port number dynamically transmitted in response to a PASV command (or equivalent): neither of both port numbers are typically pre-defined, hence those data connections cannot be statically matched by a pre-configured Flow Class filter.

The **userName** is that learned from the USER command at start of the FTP control session.

The **connectionState** is that of the underlying TCP connection. When the reporting is only at end (<Measure> **interval**="each"), the state is either '**CLOSED'** or, after a timeout TMTCP_CONN_TIME_WAIT==30s **'FIN'**. Otherwise it could reflect an intermediate state among:

- 'SYN' (improbable): connection requested by one peer, but not yet agreed by the other;
- DATA (realistic): the connection is in progress, after establishment stage and before closing stage;
- FIN (possible): one side as released the connection (or reset or both) but the other side hasn't (yet) responded to this.
- CLOSED: both ends have sent their FIN or RST packet.

4.6.10 Metric Slices Definitions Log

SUFFIX: ".metrc"

CONTENT:

```
flowID | metrictype | metricSubType | sliceNum | lower | upper
```

Example:

```
14134675419548 | 2way_delay | withInitiator | 1 | 147483648 | 0
14134675419548 | 2way_delay | withInitiator | 2 | 0 | 50
14134675419548 | 2way_delay | withInitiator | 3 | 50 | 100
14134675419548 | 2way_delay | withInitiator | 4 | 100 | 150
14134675419548 | 2way_delay | withInitiator | 5 | 150 | 200
14134675419548 | 2way_delay | withInitiator | 6 | 200 | 250
```

```
14134675419548 | 2way_delay | withInitiator | 7 | 250 | 300
14134675419548 | 2way_delay | withInitiator | 8 | 300 | 2147483647
14134675419554 | 2way_delay | withResponder | 1 | 147483648 | 0
14134675419554 | 2way_delay | withResponder | 2 | 0 | 333
14134675419554 | 2way_delay | withResponder | 3 | 333 | 666
14134675419554 | 2way_delay | withResponder | 4 | 666 | 999
14134675419554 | 2way_delay | withResponder | 5 | 999 | 1332
14134675419554 | 2way_delay | withResponder | 6 | 1332 | 1665
14134675419554 | 2way_delay | withResponder | 7 | 1665 | 1998
14134675419554 | 2way_delay | withResponder | 8 | 1998 | 2147483647
14134675419554 | 2way_delay | withInitiator | 1 | 147483648 | 0
14134675419554 | 2way_delay | withInitiator | 2 | 0 | 50
14134675419554 | 2way_delay | withInitiator | 3 | 50 | 100
14134675419554 | 2way_delay | withInitiator | 4 | 100 | 150
14134675419554 | 2way_delay | withInitiator | 5 | 150 | 200
14134675419554 | 2way_delay | withInitiator | 6 | 200 | 250
14134675419554 | 2way_delay | withInitiator | 7 | 250 | 300
14134675419554 | 2way_delay | withInitiator | 8 | 300 | 2147483647
14134675419557 | 2way_delay | withResponder | 1 | 147483648 | 0
14134675419557 | 2way_delay | withResponder | 2 | 0 | 333
14134675419557 | 2way_delay | withResponder | 3 | 333 | 666
14134675419557 | 2way_delay | withResponder | 4 | 666 | 999
14134675419557 | 2way_delay | withResponder | 5 | 999 | 2147483647
14134675419557 | 2way_delay | withInitiator | 1 | 147483648 | 0
14134675419557 | 2way_delay | withInitiator | 2 | 0 | 333
14134675419557 | 2way_delay | withInitiator | 3 | 333 | 666
14134675419557 | 2way_delay | withInitiator | 4 | 666 | 999
14134675419557 | 2way_delay | withInitiator | 5 | 999 | 2147483647
```

Explanations:

The **`metrictype`** is always **`2way_delay`** in phase I.

The **`metricSubType`** is either **`withResponder`** or **`withInitiator`** for **`2way_delay`** in phase I:

- the responder is the destination for ICMP Echo (ping) requests, is the server for DNS, NTP, SNMP, is the listener that responds to a SYN for TCP;
- the initiator is the peer that first send a transaction packet (request).

The meaning of histogram definitional parameters, **`sliceNum`**, **`lower`** and **`upper`**, is fully described in section 4.4.8 above.

4.6.11 Flow Round-Trip Delay Metrics Data Log

SUFFIX: `".2way"`

CONTENT:

```
flowID | timestamp | interval | withInitiator | sliceNum | population | minimum
| maximum | average | sum | sumOfSquares
```

Example:

```
14134675419548 | '2020-101 09:49:36' | 10 | 't' | 2 | 1 | 40 | 40 | 40 | 40 |
1600
14134719958249 | '2020-101 09:49:36' | 10 | 'f' | 8 | 1 | 6892 | 6892 | 6892 |
6892 | 47499664
14134675419548 | '2020-101 09:49:46' | 10 | 'f' | 8 | 1 | 4800 | 4800 | 4800 |
4800 | 23040000
14134719958249 | '2020-101 09:49:46' | 10 | 'f' | 6 | 1 | 1335 | 1335 | 1335 |
1335 | 1782225
14134719958249 | '2020-101 09:49:46' | 10 | 'f' | 8 | 1 | 6979 | 6979 | 6979 |
6979 | 48706441
14134719958249 | '2020-101 09:49:46' | 10 | 't' | 2 | 1 | 0 | 0 | 0 | 0 | 0
14134679871516 | '2020-101 09:49:56' | 10 | 'f' | 2 | 1 | 1 | 1 | 1 | 1 | 1
14134675419557 | '2020-101 09:49:56' | 10 | 'f' | 2 | 1 | 0 | 0 | 0 | 0 | 0
14134675419548 | '2020-101 09:49:56' | 10 | 't' | 2 | 1 | 1 | 1 | 1 | 1 | 1
14134719958249 | '2020-101 09:49:56' | 10 | 'f' | 8 | 1 | 2602 | 2602 | 2602 |
2602 | 6770404
14134719958249 | '2020-101 09:49:56' | 10 | 't' | 2 | 1 | 0 | 0 | 0 | 0 | 0
14134675419548 | '2020-101 09:50:06' | 10 | 't' | 2 | 1 | 0 | 0 | 0 | 0 | 0
14134719958249 | '2020-101 09:50:06' | 10 | 't' | 2 | 1 | 0 | 0 | 0 | 0 | 0
```

Explanations:

The **`timestamp`** marks the start of the **`interval`** whose duration is expressed in seconds.

The **`withInitiator`** is a Boolean: ‘t’ for true and ‘f’ for false.

The **`sliceNum`** is referring to the Metric Slice definition (see above) for the corresponding Flow, metric type and sub-type..

The slice aggregates contains the necessary statistical values permitting to further aggregate, between slices or over longer time periods.

The **`population`** is the number of individual measurements.

The **`average`** is sum/population.

Delays do not follow a classical Gaussian distribution, but a **Poisson law**: where the histogram peak is close to the minimum, while the right part of the histogram looks more like that of a Gaussian curve. It is therefore reasonable to consider that the histogram peak is approximated by the average and that the **95 percentile** is more or less corresponding to the range [minimum .. average+2sigma], where sigma is the (Gaussian) standard deviation:

$$\text{sigma} = \text{SQRT}\left(\frac{(\text{population} \cdot \text{sum\_sq}) - (\text{sum} \cdot \text{sum})}{\text{population} \cdot (\text{population}-1)} \right)$$

ICMP Echo round-trip delay can only be correctly measured where regular ping traffic is probed from a representative client site vis-à-vis a relevant remote site. It measures the sum of network-only latency for both directions. But it is only representative of user’s experienced performance when 1) the user’s traffic is shaped with small packets and 2) the ICMP protocol is handled the same way as the actual user’s protocol over the network path.

DNS round-trip delay can measure different things, without knowing what the several observed transactions actually consist in. When the server is efficient and has direct knowledge of the response (resolved locally, maybe cached), then the individual measurement covers only the network round-trip latency from the probe to the server and back. But when the server has to query any undetermined other remote server, this auxiliary transaction time is also counted in the individual observed longer delay. Here, careful histogram slicing can give a hint in discriminating between transaction types.

SNMP round-trip delay is typically measuring the 2-way network performance from one location (the probe at control centre) with the remote sites. But querying loaded network devices could suffer extra delay in that the responding device gives a low task priority to SNMP compared to its primary traffic switching and routing activity.

NTP round-trip delay is dual. With the responder, the probe located at client site is actually measuring its 2-way network latency with the server (expected located at a representative site of the monitored network). But NTP is actually implemented in such a way that it is possible to match the previous response packet from a given server with the next query sent by the client. Wherever the probe is actually located, this delay **with initiator** is totally dominated by the current poll time period the client has dynamically assigned to sample the server time (several seconds, up to 1024): **the longer this delay, the most reliable and stable is the NTP time distribution service quality**.

Although the **TCP SYN/ACK delay** is actually measuring the real user's traffic, it provides only one single measurement per connection and for a single side (probe vs. responder).

The TCP Data/ACK has been replaced by the more accurate monitoring of the (known quasi systematically enabled) TCP RTTM option (Round-Trip Time Measurement).

The **TCP RTTM delay** *attempts* to provide two measurements: probe with initiator, probe with responder. When the probe is close to one peer, the corresponding observed delay is meaningless. When the data traffic is asymmetric, only one round-trip delay can be measured: between the probe and the consumer of the transferred data. The RTTM intends only to dynamically measure a reasonable **upper bound** to the actual TCP round-trip time (**network latencies + buffering time**), including voluntarily delayed acknowledge, and quite volatile in case of burst sending of data packets. In this last case, it has been observed that the first sent packet is acknowledged within a delay close to the 2-way network latency; but each successive packet of the burst is ack'ed later and later (buffering of individually ack'ed TCP segments). here also, the proper segmentation of delays into histogram slices in necessary.

4.6.12 Flow Classes Hop Lists

SUFFIX: `".hops"`

CONTENT:

```
classID | className | classDesc | time | cfgVersion | cfgStart | hopCount |
hop1Name | hop2Name | hop3Name ... | Hop10Name
```

Example:

```
12345 | 'withIPTS' | 'UDP with port==21 and IP Timestamps' | '2020-10-16
13:51:41' | 100 | '2020-08-04 13:29:00' | 6 | 'remint1' | 'remint' | 'alkuf' |
'hiros' | 'kuching' | 'locdmz' | \N | \N | \N | \N
```

Explanations:

When specifying the measurement of 1-way network path latency, the Flow Class provides an ordered list of measured network hops.

Several distinct physical hops, over alternate routes, can be assigned the same hop name (where they materialise equivalent intermediate latency steps for different packets of a same flow that follow distinct routes).

A hop can be the probing point of a packet capture interface.

But a hop can also correspond to a timestamp extracted from the packet:

- one of origin, receive or transmit time from the NTP request or from the NTP response packet;
- the n^{th} timestamp stored in the IP option part of the header of specially crafted packets: corresponding to the $(n-1)^{th}$ network routing hop over its path (unlike other variants of operating system, the Linux kernel at the sender side places its own time in first position).

In the Flow Class definition, hops are expected to be given meaningful names. The ordered list of such names must be saved in the database together with the corresponding observations records. Hence, when the collector (re)starts and parses its XML configuration, it generates the hop list per Flow Class in a dedicated output file.

Due to technical constraint, it has been hardcoded that the maximum number of hops is 10.

This information permits to correctly interpret following 1-way observations log records.

4.6.13 Flow Individual 1-Way Observations Log

SUFFIX: `".1wobs"`

CONTENT:

```
flowID | timestamp | flowClass | signature | size | fragmentNumber | hop1FirstMS
ec | hop1LastMSec | hop2FirstMSec | hop2LastMSec ... | hop10FirstMSec | hop10Las
tMSec
```

Example:

```
14134675419538 | '2020-10-11 20:46:26.338' | 12345 | [37f198] | 11440 | 0 | 1413
924386338 | \N | 1413924386338 | 1413924386338 | 1413924386340 | \N | 1413924394
684 | \N | 1413924394684 | \N | 1413924394684 | 1413924394716 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:26.365' | 12345 | [6d56ba] | 11440 | 0 | 1413
924386365 | \N | 1413924386365 | 1413924386365 | 1413924386367 | \N | 1413924394
828 | \N | 1413924394828 | \N | 1413924394828 | 1413924394860 | \N | \N | \N | \
N | \N | \N | \N | \N
```

```
14134675419538 | '2020-10-11 20:46:26.392' | 12345 | [7908b0] | 11440 | 0 | 1413
924386392 | \N | 1413924386392 | 1413924386392 | 1413924386396 | \N | 1413924394
932 | \N | 1413924394932 | \N | 1413924394932 | 1413924394964 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:26.419' | 12345 | [7dbe03] | 11440 | 0 | 1413
924386419 | \N | 1413924386419 | 1413924386419 | 1413924386421 | \N | 1413924394
996 | \N | 1413924394996 | \N | 1413924394996 | 1413924395024 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:27.838' | 12345 | [d3822d] | 11440 | 0 | 1413
924387838 | \N | 1413924387838 | 1413924387838 | 1413924387840 | \N | 1413924396
764 | \N | 1413924396764 | \N | 1413924396764 | 1413924396800 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:27.865' | 12345 | [dda899] | 11440 | 0 | 1413
924387865 | \N | 1413924387865 | 1413924387865 | 1413924387867 | \N | 1413924396
884 | \N | 1413924396884 | \N | 1413924396884 | 1413924396916 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:27.892' | 12345 | [3d9ba3] | 11440 | 0 | 1413
924387892 | \N | 1413924387892 | 1413924387892 | 1413924387894 | \N | 1413924396
940 | \N | 1413924396940 | \N | 1413924396940 | 1413924396976 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:27.919' | 12345 | [c775ba] | 11440 | 0 | 1413
924387919 | \N | 1413924387919 | 1413924387919 | 1413924387921 | \N | 1413924397
028 | \N | 1413924397028 | \N | 1413924397028 | 1413924397060 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:27.946' | 12345 | [140cb4] | 11440 | 0 | 1413
924387946 | \N | 1413924387946 | 1413924387946 | 1413924387949 | \N | 1413924397
116 | \N | 1413924397116 | \N | 1413924397116 | 1413924397148 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:29.365' | 12345 | [09a435] | 11440 | 0 | 1413
924389365 | \N | 1413924389365 | 1413924389365 | 1413924389368 | \N | 1413924398
356 | \N | 1413924398356 | \N | 1413924398356 | 1413924398396 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:29.392' | 12345 | [71a787] | 11440 | 0 | 1413
924389392 | \N | 1413924389392 | 1413924389392 | 1413924389395 | \N | 1413924398
424 | \N | 1413924398424 | \N | 1413924398424 | 1413924398452 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:29.419' | 12345 | [0aca66] | 11440 | 0 | 1413
924389419 | \N | 1413924389419 | 1413924389419 | 1413924389422 | \N | 1413924398
480 | \N | 1413924398480 | \N | 1413924398480 | 1413924398516 | \N | \N | \N | \
N | \N | \N | \N | \N
14134675419538 | '2020-10-11 20:46:29.446' | 12345 | [ba177e] | 11440 | 0 | 1413
924389446 | \N | 1413924389446 | 1413924389446 | 1413924389449 | \N | 1413924398
592 | \N | 1413924398592 | \N | 1413924398592 | 1413924398628 | \N | \N | \N | \
N | \N | \N | \N | \N
```

Explanations:

As said above, there are up to ten observed timestamp per packet.

But when a probe is directed to report its packet capture times at one or several of its interfaces, it can be configured to report at reassembled datagram level `<Delay for="datagram" granularity="individual">`, and not individually for `"allFragments"` or only for `"firstFragment"`. In this `"datagram"` case it provides two timestamps for one capture interface hop: that of first seen fragment and that of last seen fragment, so each timestamp is potentially doubled. The one-way latency of datagram is the difference between youngest time at destination side minus oldest time at

origin side. Also inter-datagram time at a given probing point (hop) can be the delay between first (oldest) times of successive datagrams, but the delay from youngest time of one datagram and oldest time of its successor is also relevant.

The **timestamp** is the date/time of oldest observed hop times, at millisecond accuracy.

The **flowClass** identifier permits to link with the corresponding hop names list as depicted in the above section.

The **signature** is expected to uniquely define this packet record inside the given flow instance. It is the N bytes of MD5 hashing of the specified data chunk(s) of the packet/datagram, where N is the specified **pktSignBytes**="*N*" global parameter of the XML configuration.

The IP **size** of the packet (fragment) or datagram (reassembled as a long IP packet) is also given (expected reported the same by the several probing points).

Each **hop*X*FirstMS** (and maybe **hop*X*LastMS**) timestamp is expressed as a UNIX integer time in millisecond (inside a 64bit big integer). NULL value (\N) is given to irrelevant timestamps (only first or single fragment, over the corresponding specified **hopCount**).

In the case of **granularity**="allFragments", each record is also providing the corresponding **fragmentNumber** (first is 1). When not relevant (for "datagram" or only for "firstFragment"), **fragmentNumber**==0.

4.6.14 Flow 1-Way Latency Log

SUFFIX: ".latcy"

CONTENT:

```
flowID | timestamp | interval | perDatagram | sliceNum | population | minimum |
maximum | average | sum | sumOfSquares
```

Example:

```
145944288284407 | '2020-04-01 07:10:00' | 60 | 'f' | 7 | 104 | 20662 | 29545 |
25171 | 2617839 | 66474461081
145944289208282 | '2020-04-01 07:10:00' | 60 | 'f' | 2 | 18 | 0 | 1 | 0 | 2 | 2
145944289208269 | '2020-04-01 07:10:00' | 60 | 'f' | 2 | 18 | 0 | 0 | 0 | 0 | 0
145944288284407 | '2020-04-01 07:11:00' | 60 | 'f' | 7 | 86 | 19422 | 29075 |
24290 | 2088946 | 51738985494
145944289208282 | '2020-04-01 07:11:00' | 60 | 'f' | 2 | 16 | 0 | 7 | 0 | 7 | 49
145944289208269 | '2020-04-01 07:11:00' | 60 | 'f' | 2 | 16 | 0 | 1 | 0 | 1 | 1
145944288284407 | '2020-04-01 07:12:00' | 60 | 'f' | 7 | 28 | 23816 | 28362 |
26494 | 741853 | 19701893881
145944289208282 | '2020-04-01 07:12:00' | 60 | 'f' | 2 | 19 | 0 | 11 | 0 | 12 |
122
145944289208269 | '2020-04-01 07:12:00' | 60 | 'f' | 2 | 19 | 0 | 1 | 0 | 1 | 1
145944288284407 | '2020-04-01 07:13:00' | 60 | 'f' | 7 | 48 | 24093 | 29808 |
26616 | 1277606 | 34096160516
145944289208282 | '2020-04-01 07:13:00' | 60 | 'f' | 2 | 17 | 0 | 0 | 0 | 0 | 0
```

```
145944289208269 | '2020-04-01 07:13:00' | 60 | 'f' | 2 | 17 | 0 | 0 | 0 | 0 | 0
145944288284407 | '2020-04-01 07:14:00' | 60 | 'f' | 7 | 52 | 23686 | 29634 |
26082 | 1356278 | 35518732644
145944289208282 | '2020-04-01 07:14:00' | 60 | 'f' | 2 | 24 | 0 | 10 | 0 | 20 |
166
145944289208269 | '2020-04-01 07:14:00' | 60 | 'f' | 2 | 24 | 0 | 9 | 0 | 14 |
92
```

4.6.15 Flow 1-Way Abnormality Counters Log

For instance, all three counters in a single file:

SUFFIX: `".1wct"`
(also, separately, `".1wlost"` and, as exceptions, `".1wmiss"` and `".1wdrop"`)

CONTENT:

```
flowID | timestamp | interval | lost | partlyMissed | dropped
```

Example:

```
145944288284407 | '2020-04-01 06:30:00' | 60 | 0 | 8 | 0
145944288284407 | '2020-04-01 06:34:00' | 60 | 1 | 0 | 0
```