Network Working Group S. Shalunov
Internet-Draft
Intended status: Standards Track M. Swany
Expires: January 14, 2011 University of Delaware
July 13, 2010

Reporting IP Performance Metrics to Users
draft-ietf-ippm-reporting-05.txt

Abstract

The aim of this document is to define a small set of metrics that are robust, easy to understand, orthogonal, relevant, and easy to compute. The IPPM WG has defined a large number of richly parameterized metrics because network measurement has many purposes. Often, the ultimate purpose is to report a concise set of metrics describing a network's current state to an end user. It is for this purpose that the present set of metrics is defined, and the document is principally concerned with "short-term" reporting considerations (a few seconds or minutes as opposed to days, months or years.)

Status of this Memo

Copyright Notice

Table of Contents

1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Introduction

The IPPM working group has defined many richly parameterized performance metrics with a number of variants (one-way delay, one-way loss, delay variation, reordering, etc.) and a protocol for obtaining the measurement data needed to compute these metrics (OWAMP). It would be beneficial to define a standard way to report a set of performance metrics to end users. Parameterization might be acceptable in such a set, but there must still be defaults for everything. It is especially important to get these defaults right. Such a set would enable different tools to produce results that can be compared against each other.

Existing tools already report statistics about the network. This is done for varying reasons: network testing tools, such as the ping program available in UNIX-derived operating systems as well as in Microsoft Windows, report statistics with no knowledge of why the user is running the program; networked games might report statistics of the network connection to the server so users can better understand why they get the results they get (e.g., if something is slow, is this because of the network or the CPU?), so they can compare their statistics to those of others ("you're not lagged any more than I am") or perhaps so that users can decide whether they need to upgrade the connection to their home; IP telephony hardware and software might report the statistics for similar reasons. While existing tools report statistics, the particular set of metrics they choose is ad hoc; some metrics are not statistically robust, some are not relevant, and some are not easy to understand; more important than specific shortcomings, however, is the incompatibility: even if the sets of metrics were perfect, they would still be all different, and, therefore, metrics reported by different tools would not be comparable.

The set of metrics of this document is meant for human consumption. It must therefore be small. A screen full of numbers is likely to be too confusing. Restricting output to a single number would likewise not be descriptive enough. (Would you pick loss? delay? throughput? something else?) In this document we aim for a "handful" of numbers.

Each of the metrics must be statistically robust. Intuitively, this means that having a small number of bad data points and a small amount of noise must not dramatically change the metric.

Each metric in the set must have, qualitatively, an immediate intuitive meaning that has to be obvious for an advanced end user without consulting documentation (that is, it has to be clear what rough meaning, intuitively, the larger values of a given metric have).

To be small, the set has to be orthogonal: each of the metrics has to express a property not covered by other metrics (otherwise, there's redundancy).

The metrics in the set must be relevant. Partly, being easy to understand will help achieve this, but additional constraint may be placed by relevancy.

Finally, while this set will most frequently be computed for small data sets, where efficiency is not a serious consideration, it must be possible to compute for large data sets, too. In particular, it must be possible to compute the metrics in a single pass over the data using a limited amount of memory (i.e., it must be possible to take a source of measurement data with a high data rate and compute the metrics on the fly, discarding each data point after it is processed).

3. Applicability for Long-Term Measurements

The metrics in this document are most applicable to short-term network measurements (seconds or at most minutes) and are targeted for real-time display of such measurements.

One consideration that would have to be addressed to make these metrics suitable for longer-term measurements (hours and days) is that of network availability: during such long periods of time the network may be completely down for some time and it does not seem to make sense to average out the reports in such a way that the network being down for 1% of the time becomes 1% packet loss.

4. Reportable Metrics Set

The following metrics comprise the set:

1. median delay;
2. loss ratio;
3. delay spread;
4. duplication;
5. reordering.

Each of the above is represented by a single numeric quantity, computed as described below.

4.1. Median Delay

The reported median delay is the median of all delays in the sample. When a packet is lost, its delay is to be considered +infinity for the purposes of this computation; therefore, if more than half of all packets are lost, the delay is +infinity.

For more information, refer to section 5.2 (Type-P-One-way-Delay-Median) of RFC 2679 [RFC2679] (A One-way Delay Metric for IPPM), and supporting text.

4.2. Loss Ratio

Loss Ratio is the fraction, expressed as a percentile, of packets that did not arrive intact within a given number of seconds (the timeout value) after being sent. Since this set of metrics often has to be reported to a waiting human user, the default timeout value should be small. By default, 2 seconds MUST be the timeout value. Use of a different timeout value MUST be reported.

For more information, refer to Section 4.1 (Type-P-One-way-Packet-Loss-Average) of RFC 2680 [RFC2680] (A One-way Packet Loss Metric for IPPM). The Loss Ratio is 100*Type-P-One-way-Packet-Loss-Average.

4.3. Delay Spread

Delay spread is the interquartile spread of observed delays. This is a metric to represent what is commonly referred to as "jitter". Delay spread is reported as the difference of the 75th and 25th percentiles of delay. When both percentiles are +infinity, the value of delay spread is undefined. Therefore, if less than 25% of packets

are lost, it is defined and finite; if between 75% and 25% of packets are lost, it is +infinity; finally, if more than 75% of packets are lost, it is undefined.

For more information, refer to section 5.1 (Type-P-One-way-Delay-Percentile) of RFC 2679 [RFC2679] (A One-way Delay Metric for IPPM), and supporting text. The Delay Spread is the 75th Type-P-One-way-Delay-Percentile minus the 25th Type-P-One-way-Delay-Percentile.

4.4. Duplication

Duplication is the fraction of packets sent but not lost for which more than a single copy of the packet was received within the timeout period (ideally using the same timeout as in the definition of loss), expressed in percentage points.

Note: while most received packets can be ones previously seen, duplication can never exceed 100%.

For more information, see section 5.2 (Type-P-one-way-replicated-packet-rate) of RFC 5560 [RFC5560] (A One-Way Packet Duplication Metric). Duplication is Type-P-one-way-replicated-packet-rate expressed as a percentage.

4.5. Reordering

Reordering is the fraction of unique packets received for which the sequence number of any given packet is less than the highest sequence number largest sequence number of any packet previously received. For the purposes of determining the sequence number of the preceding packets in this definition, assume sequence numbers starting with 1, and an extra packet at the start of the stream of received packets, with a sequence number of 0, is considered to be present (this extra packet, of course, is not counted for the purposes of computing the fraction).

For more information, refer to section 4.1.3 (Type-P-Reordered-Ratio-Stream) of RFC 4737 [RFC4737] (Packet Reordering Metrics), and supporting text. The precise definition of a reordered packet is in section 3.3.

{Comment: As the non-normative sample code in Appendix A below shows, this is also related to the amount of 1-reordering (Section 5.3 RFC 4737 [RFC4737]). It is not, however, the degree of 1-reordering in 5.3; because 1-reordering divides by the number of all packets received, instead of the number of non-duplicate packets received.}

5. Sample Source

Section 4 describes the metrics to compute on a sample of measurements. The source of the sample in not discussed there, and, indeed, the metrics discussed (delay, loss, etc.) are simply estimators that could be applied to any sample whatsoever. For the names of the estimators to be applicable, of course, the measurements need to come from a packet delivery network.

The data in the samples for the set of metrics discussed in this document can come from the following sources: one-way active measurement, round-trip measurement, and passive measurement. There infrequently is a choice between active and passive measurement, as, typically, only one is available; consequently, no preference is given to one over the other. In cases where clocks can be expected to be synchronized, in general, one-way measurements are preferred over round-trip measurements (as one-way measurements are more informative). When one-way measurements cannot be obtained, or when clocks cannot be expected to be synchronized, round-trip measurement MAY be used.

5.1. One-Way Active Measurement

The default duration of the measurement interval is 10 seconds.

The default sending schedule is a Poisson stream.

The default sending rate is 10 packets/second on average. The default sending schedule is a Poisson stream. When randomized schedules, such as a Poisson stream, are used, the rate MUST be set with the distribution parameter(s). With a randomized schedule, the default sample size is 100 packets and the measurement window duration can vary to some extent depending on the values of the (pseudo-)random deviates.

The default packet size is the minimum necessary for the measurement.

Values other than the default ones MAY be used; if they are used, their use, and specific values used, MUST be reported.

A one-way active measurement is characterized by the source IP address, the destination IP address, the time when measurement was taken, and the type of packets (e.g., UDP with given port numbers and a given DSCP) used in the measurement. For the time, the end of the measurement interval MUST be reported.

5.2. Round-Trip Active Measurement

The same default parameters and characterization apply to round-trip measurement as to one-way measurement (Section 5.1).

5.3. Passive Measurement

Passive measurement use whatever data it is natural to use. For example, an IP telephony application or a networked game would use the data that it sends. An analysis of performance of a link might use all the packets that traversed the link in the measurement interval. An analysis of performance of an Internet service provider's network might use all the packets that traversed the network in the measurement interval. An analysis of performance of a specific service from the point of view of a given site might use an appropriate filter to select only the relevant packets. In any case, the source needs to be reported.

The same default duration applies to passive measurement as to one-way active measurement (Section 5.1).

When the passive measurement data is reported in real time, or based on user demand, a sliding window SHOULD be used as a measurement period, so that recent data become more quickly reflected. For historical reporting purposes, a fixed interval may be preferable.

6. Security Considerations

The reporting per se, not being a protocol, does not raise security considerations.

An aspect of reporting relevant to security is how the reported metrics are used and how they are collected. If it is important that the metrics satisfy certain conditions (e.g., that the ISP whose network is being measured be unable to make the metrics appear better than they are), the collection mechanism MUST ensure that this is, indeed, so. The exact mechanisms to do so are outside of scope of this document and belong with discussion of particular measurement data collection protocols.

7. Acknowledgments

We gratefully acknowledge discussion with, encouragement from, and contributions of Lawrence D. Dunn, Reza Fardid, Ruediger Geib, Matt Mathis, Al Morton, Carsten Schmoll, Henk Uijterwaal, and Matthew J. Zekauskas.

8. IANA Considerations

This document requires no action from the IANA.

9. Internationalization Considerations

The reported metrics, while they might occasionally be parsed by machine, are primarily meant for human consumption. As such, they MAY be reported in the language preferred by the user, using an encoding suitable for the purpose, such as UTF-8.

10. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2679] Almes, G., Kalidindi, S., and M. Zekauskas, "A One-way Delay Metric for IPPM", RFC 2679, September 1999.

[RFC2680] Almes, G., Kalidindi, S., and M. Zekauskas, "A One-way Packet Loss Metric for IPPM", RFC 2680, September 1999.

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

[RFC5560] Uijterwaal, H., "A One-Way Packet Duplication Metric", RFC 5560, May 2009.

Appendix A. Sample Source Code for Computing the Metrics

This appendix only serves for illustrative purposes.

```
/*
 * reporting.c -- performance metrics reporting as in Internet Draft
 *                draft-ietf-ippm-reporting.
 *
 * Written by Stanislav Shalunov, http://www.internet2.edu/~shalunov/
 *            Bernhard Lutzmann, belu@users.sf.net
 *            Federico Montesino Pouzols, fedemp@altern.org
 *
 * This file is also available, under a different (BSD-style)
 * license, as part of thrulay.
 */

/**
 * @file reporting.c
 *
 * @short metrics computation and reporting.
 **/

#include <stdlib.h>
#include <stdint.h>
#include <float.h>
#include <math.h>
#include <string.h>
#include <assert.h>

#define min(a, b)       ((a) < (b) ? (a) : (b))
#define max(a, b)       ((a) > (b) ? (a) : (b))

/*
 * Reordering.
 */
#define loop(x)         ((x) >= 0 ? (x) : (x) + (int)reordering_max)

/*
 * Duplication.
 */
static uint64_t *bitfield = NULL; /* Bit field used to check for
                                     duplicated packets. */

/*
 * Reordering.
 */
static uint64_t reordering_max;   /* We have m[j-1] == number of */
```

```
   static uint64_t *reordering_m;    /* We have m[j-1] == number of
                                        j-reordered packets. */
   static uint64_t *reordering_ring; /* Last sequence numbers seen */
   static int r = 0;                 /* Ring pointer for next write. */
   static int l = 0;                 /* Counter of sequence numbers. */

   /*
    * Quantiles
    *
    * Reference:
    *
    * [1] Manku, Rajagopalan, Lindsay: "Approximate Medians and other
    *     Quantiles in One Pass and with Limited Memory",
    * http://www-db.stanford.edu/~manku/papers/98sigmod-quantiles.pdf
    */

   #define QUANTILE_EPS    0.005

   static uint16_t quantile_max_seq; /* Maximum number of sequences */
   static int *quantile_k;         /* number of elements in buffer */

   static double **quantile_input; /* This is the buffer where the
                                      sequence of incoming packets is
                                      saved.  If we received enough
                                      packets, we will write this
                                      buffer to a quantile buffer. */
   static int *quantile_input_cnt;   /* number of elements in input
                                      * buffer */
   static int *quantile_empty_buffers; /* number of empty buffers */

   static int *quantile_b;         /* number of buffers */

   static double **quantile_buf;

   static int *quantile_alternate;     /* this is used to determine
                                          the offset in COLLAPSE (if
                                          weight is even) */

   static uint64_t *quantile_inf_cnt;  /* this counter is for the
                                          additional -inf, +inf
                                          elements we added to NEW
                                          buffer to fill it up. */

   typedef struct quantile {
       struct quantile *next;      /* pointer to next quantile
                                    * buffer */
       int weight;         /* 0 if buffer is empty, > 0 if buffer is
                            * full */
```

```
       int level;
       double *buffer;
       int pos;            /* current position in buffer; used in
                              quantile_collapse() */
   } quantile_t;

   static quantile_t **quantile_buffer_head;

   int
   reordering_init(uint64_t max)
   {
       reordering_max = max;
       reordering_m = calloc(reordering_max, sizeof(uint64_t));
       reordering_ring = calloc(reordering_max, sizeof(uint64_t));
       if (reordering_m == NULL) {
           return -1;
       } else {
           return 0;
       }
   }

   int
   reordering_checkin(uint64_t packet_sqn)
   {
       int j;

       for (j = 0; j < min(l, (int)reordering_max) &&

                packet_sqn < reordering_ring[loop(r-j-1)]; j++) {
           reordering_m[j]++;
       }
       reordering_ring[r] = packet_sqn;
       l++;
       r = (r + 1) % reordering_max;
       return 0;
   }

   double
   reordering_output(uint64_t j)
   {
       if (j >= reordering_max)
           return -1;
       else
           return (double)reordering_m[j] / l);
   }

   void
   reordering_exit(void)
```

```
   {
       free(reordering_ring);
       free(reordering_m);
   }

   int
   duplication_init(uint64_t npackets)
   {
       uint64_t bitfield_len = 0; /* Number of sectors in bitfield */

       /* Allocate memory for bit field */
       bitfield_len = ((npackets % 64)?
                       (npackets / 64 + 1) : npackets / 64);
       bitfield = calloc(bitfield_len, sizeof(uint64_t));
       if (bitfield == NULL) {
           return -1;
       } else {
           return 0;
       }
   }

   int
   duplication_check(uint64_t packet_sqn)
   {
       uint64_t bitfield_sec; /* Which sector in bitfield */
       uint64_t bitfield_bit; /* Which bit in sector */

       /* Calculate sector */
       bitfield_sec = packet_sqn >> 6;

       /* Calculate bit in sector */
       bitfield_bit = (uint64_t)1 << (packet_sqn & 63);

       if (bitfield[bitfield_sec] & bitfield_bit) {
           /* Duplicated packet */
           return 1;
       } else {
           /* Unique packet */
           bitfield[bitfield_sec] |= bitfield_bit;
           return 0;
       }
   }

   void
   duplication_exit(void)
   {
       free(bitfield);
   }
```

```
   /* Calculate binomial coefficient C(n, k). */
   int64_t
   binomial (int n, int k)
   {
       int64_t bc = 0;
       int i, m;

       /* C(n, k) = C(n, n-k) */
       k = min(k, n-k);

       if (k >= 0) {
           bc = 1;
           m = max(k, n-k);

           for (i = 1; i <= k; i++) {
               bc = (bc * (m + i)) / i;
           }
       }

       return bc;
   }

   int
   quantile_compare(const void *d1, const void *d2)
   {
       if (*(double *)d1 < *(double *)d2) {
           return -1;
       } else if (*(double *)d1 == *(double *)d2) {
           return 0;
       } else {
           assert(*(double *)d1 > *(double *)d2);
           return 1;
       }
   }

   void
   quantile_sort (double *list, int length)
   {
       qsort(list, length, sizeof(double), quantile_compare);
   }

   /**
    * Implementation of NEW operation from section 3.1 of paper [1].
    *
    * Takes as input an empty buffer. Simply populates the quantile
    * buffer with the next k elements from the input sequence, labels
    * the buffer as full, and assigns it a weight of 1.
    *
```

```
     * If there are not enough elements to fill up the buffer, we
     * alternately add -inf, +inf elements until buffer is full (-inf
     * == 0, +inf == DBL_MAX).
     *
     * NOTE: We sort the elements in the input buffer before we copy
     * them to the quantile buffer.
     *
     * @param seq Sequence index.
     *
     * @return
     * @retval 0 on success.
     * @retval -2 need an empty buffer.
     * @retval -3 bad input sequence length.
     **/
    int
    quantile_new(uint16_t seq, quantile_t *q, double *input, int k,
                 int level)
    {
        int i;

        /* Check that buffer is really empty. */
        if (q->weight != 0) {
            return -2;
        }

        /* Sanity check for length of input sequence. */
        if (k > quantile_k[seq]) {
            return -3;
        }

        /* If there are not enough elements in the input buffer, fill
         * it up with -inf, +inf elements. */
        for (i = k; i < quantile_k[seq]; i++) {
            if (i % 2) {
                input[i] = DBL_MAX;
            } else {
                input[i] = 0;
            }

            /* Increment counter that indicates how many additional
             * elements we added to fill the buffer. */
            quantile_inf_cnt[seq]++;
        }

        quantile_sort(input, quantile_k[seq]);

        memcpy(q->buffer, input, sizeof(double) * quantile_k[seq]);
```

```
        /* Mark buffer as full and set level. */
        q->weight = 1;
        q->level = level;

        /* Update number of empty quantile buffers. */
        quantile_empty_buffers[seq]--;

        return 0;
    }


    /* Implementation of COLLAPSE operation from section 3.2 of paper
     * [1].
     *
     * This is called from quantile_algorithm() if there are no empty
     * buffers.  We COLLAPSE all the full buffers, where level has
     * value `level'.  Output is written to the first buffer in linked
     * list with level set to `level'. The level of the output buffer
     * is increased by 1.  All other buffers we used in the COLLAPSE
     * are marked empty. */
    int
    quantile_collapse(uint16_t seq, int level)
    {
        quantile_t *qp = NULL, *qp_out = NULL;
        int num_buffers = 0;        /* number of buffers with level
                                     * `level' */
        int weight = 0;             /* weight of the output buffer */
        int offset;
        int i, j;
        double min_dbl;
        long next_pos;
        long merge_pos = 0;

        /* Check that there are at least two full buffers with given
         * level.  Also calculate weight of output buffer. */
        for (qp = quantile_buffer_head[seq]; qp != NULL; qp = qp->next) {
            if ((qp->weight != 0) && (qp->level == level)) {
                num_buffers++;
                weight += qp->weight;
                qp->pos = 0;
            } else {
                /* We mark the buffers that are not used in this
                 * COLLAPSE. */
                qp->pos = -1;
            }
        }
        if (num_buffers < 2) {
            return -4;
```

```
       }

       /* NOTE: The elements in full buffers are sorted. So we don't
        * have to do that again.
        */
       /* Search for first full buffer with matching level. This is
        * the buffer where we save the output. */
       for (qp_out = quantile_buffer_head[seq]; qp_out != NULL;
            qp_out = qp_out->next) {
           if (qp_out->pos != -1) {
               break;
           }
       }

       /* Calculate offset */
       if (weight % 2) {
           /* odd */
           offset = (weight + 1) / 2;
       } else {
           /* even - we alternate between two choices in each
            * COLLAPSE */
           if (quantile_alternate[seq] % 2) {
               offset = weight / 2;
           } else {
               offset = (weight + 2)/ 2;
           }
           quantile_alternate[seq] = (quantile_alternate[seq] + 1) % 2;
       }

       /* Initialize next position of element to save. Because first
        * position is at 0, we have to decrement offset by 1. */
       next_pos = offset - 1;

       for (i = 0; i < quantile_k[seq]; ) {

           /* Search for current minimal element in all buffers.
            * Because buffers are all sorted, we just have to check
            * the element at current position. */
           min_dbl = DBL_MAX;
           for (qp = quantile_buffer_head[seq]; qp != NULL;
                qp = qp->next) {
               /* Skip wrong buffers. */
               if (qp->pos == -1) {
                   continue;
               }

               /* Check that we are not at the end of buffer. */
               if (qp->pos >= quantile_k[seq]) {
```

```
                    continue;
                }

                /* Update minimum element. */
                min_dbl = min(min_dbl, qp->buffer[qp->pos]);
            }

            /* Now process this minimal element in all buffers. */
            for (qp = quantile_buffer_head[seq]; qp != NULL;
                 qp = qp->next) {
                /* Skip wrong buffers. */
                if (qp->pos == -1) {
                    continue;
                }

                /* Now process minimal element in this buffer. */
                for (; (qp->buffer[qp->pos] == min_dbl) &&
                        (qp->pos < quantile_k[seq]);
                     qp->pos++) {

                    /* We run this loop `qp->weight' times.
                     * We check there if we are in a position
                     * so we have to save this element in our
                     * output buffer. */
                    for (j = 0; j < qp->weight; j++) {

                        if (next_pos == merge_pos) {
                            quantile_buf[seq][i] = min_dbl;
                            i++;

                            if (i == quantile_k[seq]) {
                                /* We have written
                                 * all elements to
                                 * output buffer, so
                                 * exit global loop. */
                                goto out;
                            }

                            /* Update next position. */
                            next_pos += weight;
                        }

                        merge_pos++;
                    } /* for(j = 0; j < qp->weight; j++) */
                } /* for (; (qp->buffer[qp->pos] == min_dbl) &&
                     (qp->pos < quantile_k[seq]); qp->pos++) */
            } /* for (qp = quantile_buffer_head[seq]; qp!=NULL;
                 qp = qp->next) */
```

```
       } /* for (i = 0; i < quantile_k[seq]; ) */

    out:
       memcpy(qp_out->buffer, quantile_buf[seq],
              sizeof(double) * quantile_k[seq]);

       /* Update weight of output buffer. */
       qp_out->weight = weight;
       qp_out->level = level+1;

       /* Update list of empty buffers. */
       for (qp = quantile_buffer_head[seq]; qp != NULL; qp = qp->next) {
           if ((qp->pos != -1) && (qp != qp_out)) {
               qp->weight = 0;
               qp->level = 0;
           }
       }
       quantile_empty_buffers[seq] += num_buffers - 1;
       return 0;
   }

   /**
    * Implementation of COLLAPSE policies from section 3.4 of paper
    * [1].
    *
    * There are three different algorithms noted in the paper. We use
    * the "New Algorithm".
    *
    * @param seq Sequence index.
    *
    * @return
    * @retval 0 on success.
    * @retval -1 quantiles not initialized.
    * @retval -2 need an empty buffer for new operation.
    * @retval -3 bad input sequence length in new operation.
    * @retval -4 not enough buffers for collapse operation.
    **/
   int
   quantile_algorithm (uint16_t seq, double *input, int k)
   {
       int rc;
       quantile_t *qp = NULL, *qp2 = NULL;
       int min_level = -1;

       /* This should always be true. */
       if (quantile_buffer_head[seq] != NULL) {
           min_level = quantile_buffer_head[seq]->level;
       } else {
```

```
            return -1;
        }

        /* Get minimum level of all currently full buffers. */
        for (qp = quantile_buffer_head[seq]; qp != NULL; qp = qp->next) {
            if (qp->weight != 0) {
                /* Full buffer. */
                min_level = min(min_level, qp->level);
            }
        }

        if (quantile_empty_buffers[seq] == 0) {
            /* There are no empty buffers. Invoke COLLAPSE on the set
             * of buffers with minimum level. */

            rc = quantile_collapse(seq, min_level);
            if (rc < 0)
                return rc;
        } else if (quantile_empty_buffers[seq] == 1) {
            /* We have exactly one empty buffer. Invoke NEW and assign
             * it level `min_level'. */

            /* Search the empty buffer. */
            for (qp = quantile_buffer_head[seq]; qp != NULL;
                 qp = qp->next) {
                if (qp->weight == 0) {
                    /* Found empty buffer. */
                    break;
                }
            }

            rc = quantile_new(seq, qp, input, k, min_level);
            if (rc < 0)
                return rc;
        } else {
            /* There are at least two empty buffers. Invoke NEW on each
             * and assign level `0' to each. */

            /* Search for two empty buffers. */
            for (qp = quantile_buffer_head[seq]; qp != NULL;
                 qp = qp->next) {
                if (qp->weight == 0) {
                    /* Found first empty buffer. */
                    break;
                }
            }
            for (qp2 = qp->next; qp2 != NULL; qp2 = qp2->next) {
                if (qp2->weight == 0) {
```

```
                  /* Found second empty buffer. */
                  break;
              }
          }

          if (k <= quantile_k[seq]) {
              /* This could happen if we call this after we
               * received all packets but don't have enough to
               * fill up two buffers. */

              rc = quantile_new(seq, qp, input, k, 0);
              if (rc < 0)
                  return rc;
          } else {
              /* We have enough input data for two buffers. */
              rc = quantile_new(seq, qp, input, quantile_k[seq], 0);
              if (rc < 0)
                  return rc;
              rc = quantile_new(seq, qp2, input + quantile_k[seq],
                                k - quantile_k[seq], 0);
              if (rc < 0)
                  return rc;
          }
      }
      return 0;
  }

  int
  quantile_init_seq(uint16_t seq)
  {
      quantile_t *qp = NULL;
      int i;

      if ( seq >= quantile_max_seq)
          return -5;

      /* Allocate memory for quantile buffers. Buffers are linked
       * lists with a pointer to next buffer.  We need `quantile_b'
       * buffers, where each buffer has space for `quantile_k'
       * elements. */
      for (i = 0; i < quantile_b[seq]; i++) {
          if (i == 0) {
              /* Initialize first buffer. */
              qp = malloc(sizeof(quantile_t));
              if (qp == NULL) {
                  return -1;
              }
              quantile_buffer_head[seq] = qp;
```

```
           } else {
               qp->next = malloc(sizeof(quantile_t));
               if (qp->next == NULL) {
                   return -1;
               }
               qp = qp->next;
           }

           /* `qp' points to buffer that should be initialized. */
           qp->next = NULL;
           qp->weight = 0; /* empty buffers have weight of 0 */
           qp->level = 0;
           qp->buffer = malloc(sizeof(double) * quantile_k[seq]);
           if (qp->buffer == NULL) {
               return -1;
           }
       }
       /* Update number of empty quantile buffers. */
       quantile_empty_buffers[seq] = quantile_b[seq];

       return 0;
   }

   int
   quantile_init (uint16_t max_seq, double eps, uint64_t N)
   {
       int b, b_tmp = 0;
       int k, k_tmp = 0;
       int h, h_max = 0;
       int seq, rc;

       quantile_max_seq = max_seq;
       /* Allocate array for the requested number of sequences. */
       quantile_k = calloc(max_seq, sizeof(int));
       quantile_input = calloc(max_seq, sizeof(double*));
       quantile_input_cnt = calloc(max_seq, sizeof(int));
       quantile_empty_buffers = calloc(max_seq, sizeof(int));
       quantile_b = calloc(max_seq, sizeof(int));
       quantile_buf = calloc(max_seq, sizeof(double*));
       quantile_alternate = calloc(max_seq, sizeof(int));
       quantile_inf_cnt = calloc(max_seq, sizeof(uint64_t));
       quantile_buffer_head = calloc(max_seq, sizeof(quantile_t*));

       /* "In practice, optimal values for b and k can be computed by
        * trying out different values of b in the range 1 and 30." */
       for (b = 2; b <= 30; b++) {
           /* For each b, compute the largest integral h that
            * satisfies:
```

```
             *   (h-2) * C(b+h-2, h-1) - C(b+h-3, h-3) +
             *                         C(b+h-3, h-2) <= 2 * eps * N
             */
            for (h = 0; ; h++) {
                if (((h-2) * binomial(b+h-2, h-1) -
                     binomial(b+h-3, h-3) +
                     binomial(b+h-3, h-2)) >
                    (2 * eps * N)) {
                    /* This h does not satisfy the inequality from
                     * above. */
                    break;
                }
                h_max = h;
            }

            /* Now compute the smallest integral k that satisfies:
             *   k * C(b+h-2, h-1) => N. */
            k = ceil(N / (double)binomial(b+h_max-2, h_max-1));

            /* Identify that b that minimizes b*k. */
            if ((b_tmp == 0) && (k_tmp == 0)) {
                /* Initialize values */
                b_tmp = b;
                k_tmp = k;
            }

            if ((b * k) < (b_tmp * k_tmp)) {
                /* Found b and k for which the product is smaller than
                 * for the ones before. Because we want to minimize
                 * b*k (required memory), we save them. */
                b_tmp = b;
                k_tmp = k;
            }
        }

        /* Set global quantile values. For now, all sequences share
           the same k and b values.*/
        for (seq = 0; seq < max_seq; seq++ ) {
            quantile_b[seq] = b_tmp;
            quantile_k[seq] = k_tmp;
        }

        /* Allocate memory for input buffer.  We allocate enough space
         * to save up to `2 * quantile_k' elements.  This space is
         * needed in the COLLAPSE policy if there are more than two
         * empty buffers. Because then we have to invoke NEW on two
         * buffers and thus need an input buffer with `2 * quantile_k'
         * elements. */
```

```
       for (seq = 0; seq < quantile_max_seq; seq++) {
           quantile_input[seq] = malloc(sizeof(double) * 2 *
                                        quantile_k[seq]);
           if (quantile_input[seq] == NULL) {
               return -1;
           }
           quantile_input_cnt[seq] = 0;
       }

       /* Allocate memory for output buffer.  This buffer is used in
        * COLLAPSE to store temporary output buffer before it gets
        * copied to one of the buffers used in COLLAPSE. */
       for (seq = 0; seq < quantile_max_seq; seq++ ) {
           quantile_buf[seq] = malloc(sizeof(double) * quantile_k[seq]);
           if (quantile_buf[seq] == NULL) {
               return -1;
           }
       }

       for (seq = 0; seq < max_seq; seq++) {
           rc = quantile_init_seq(seq);
           if (rc < 0)
               return rc;
       }

       return 0;
   }

   int
   quantile_value_checkin(uint16_t seq, double value)
   {
       int rc = 0;

       if ( seq >= quantile_max_seq)
           return -5;

       quantile_input[seq][quantile_input_cnt[seq]++] = value;

       /* If we have at least two empty buffers,
        * we need input for two buffers, to twice
        * the value of `quantile_k'. */
       if (quantile_empty_buffers[seq] >= 2) {
           if (quantile_input_cnt[seq] ==
               (2 * quantile_k[seq])) {
               rc = quantile_algorithm(seq, quantile_input[seq],
                                       quantile_input_cnt[seq]);
               /* Reset counter. */
               quantile_input_cnt[seq] = 0;
```

```
            }
        } else {
            /* There are 0 or 1 empty buffers */
            if (quantile_input_cnt[seq] == quantile_k[seq]) {
                rc = quantile_algorithm(seq, quantile_input[seq],
                                        quantile_input_cnt[seq]);
                /* Reset counter. */
                quantile_input_cnt[seq] = 0;
            }
        }
        return rc;
    }

    int
    quantile_finish(uint16_t seq)
    {
        int rc = 0;

        if ( seq >= quantile_max_seq)
            return -5;

        if (quantile_input_cnt[seq] > 0) {
            rc = quantile_algorithm(seq, quantile_input[seq],
                                    quantile_input_cnt[seq]);
        }
        return rc;
    }

    void
    quantile_reset(uint16_t seq)
    {
        quantile_input_cnt[seq] = 0;
        quantile_empty_buffers[seq] = quantile_b[seq];
        memset(quantile_buf[seq],0,sizeof(double) * quantile_k[seq]);
        memset(quantile_input[seq],0,sizeof(double) * quantile_k[seq]);
    }

    /**
     * Deinitialize one quantile sequence.
     **/
    void
    quantile_exit_seq(uint16_t seq)
    {
        quantile_t *qp = NULL, *next;

        if (seq >= quantile_max_seq)
            return;
```

```
       qp = quantile_buffer_head[seq];
       while (qp != NULL) {
           /* Save pointer to next buffer. */
           next = qp->next;

           /* Free buffer and list entry. */
           free(qp->buffer);
           free(qp);

           /* Set current buffer to next one. */
           qp = next;
       }

       quantile_buffer_head[seq] = NULL;
       quantile_input_cnt[seq] = 0;
       quantile_empty_buffers[seq] = quantile_b[seq];
   }

   void
   quantile_exit(void)
   {
       int seq;

       /* Free per sequence structures */
       for (seq = 0; seq < quantile_max_seq; seq++) {
           quantile_exit_seq(seq);

           /* Free output buffer. */
           free(quantile_buf[seq]);

           /* Free input buffer. */
           free(quantile_input[seq]);
       }

       free(quantile_buffer_head);
       free(quantile_inf_cnt);
       free(quantile_alternate);
       free(quantile_buf);
       free(quantile_b);
       free(quantile_empty_buffers);
       free(quantile_input_cnt);
       free(quantile_input);
       free(quantile_k);
   }

   int
   quantile_output (uint16_t seq, uint64_t npackets, double phi,
                    double *result)
```

```
   {
       quantile_t *qp = NULL;
       int num_buffers = 0;
       int weight = 0;
       int j;
       long next_pos = 0;
       long merge_pos = 0;
       double min_dbl;
       double beta;
       double phi2;                /* this is phi' */

       if ( seq >= quantile_max_seq)
           return -5;

       /* Check that there are at least two full buffers with given
        * level. */
       for (qp = quantile_buffer_head[seq]; qp != NULL; qp = qp->next) {
           if (qp->weight != 0) {
               num_buffers++;
               weight += qp->weight;
               qp->pos = 0;
           } else {
               qp->pos = -1;
           }
       }
       if (num_buffers < 2) {
           /* XXX: In section 3.3 "OUTPUT operation" of paper [1] is
            * says that OUTPUT takes c => 2 full input buffers. But
            * what if we just have one full input buffer?
            *
            * For example this happens if you run a UDP test with a
            * block size of 100k and a test duration of 3 seconds: $
            * ./thrulay -u 100k -t 3 localhost
            */

           if (num_buffers != 1) {
               return -1;
           }
       }

       /* Calculate beta and phi' */
       beta = 1 + quantile_inf_cnt[seq] / (double)npackets;
       assert(beta >= 1.0);

       assert(phi >= 0.0 && phi <= 1.0);
       phi2 = (2 * phi + beta - 1) / (2 * beta);

       next_pos = ceil(phi2 * quantile_k[seq] * weight);
```

```
       /* XXX: If the client just sends a few packets, it is possible
        * that next_pos is too large. If this is the case, decrease
        * it. */
       if (next_pos >= (num_buffers * quantile_k[seq])) {
           next_pos --;
       }

       while (1) {

           /* Search for current minimal element in all buffers.
            * Because buffers are all sorted, we just have to check
            * the element at current position. */
           min_dbl = DBL_MAX;
           for (qp = quantile_buffer_head[seq]; qp != NULL;
                qp = qp->next) {
               /* Skip wrong buffers. */
               if (qp->pos == -1) {
                   continue;
               }

               /* Check that we are not at the end of buffer. */
               if (qp->pos >= quantile_k[seq]) {
                   continue;
               }

               /* Update minimum element. */
               min_dbl = min(min_dbl, qp->buffer[qp->pos]);
           }

           /* Now process this minimal element in all buffers. */
           for (qp = quantile_buffer_head[seq]; qp != NULL;
                qp = qp->next) {
               /* Skip wrong buffers. */
               if (qp->pos == -1) {
                   continue;
               }

               /* Now process minimal element in this buffer. */
               for (; (qp->buffer[qp->pos] == min_dbl) &&
                        (qp->pos < quantile_k[seq]);
                    qp->pos++) {

                   /* Increment merge position `qp->weight'
                    * times. If we pass the position we seek,
                    * return current minimal element. */
                   for (j = 0; j < qp->weight; j++) {
                       if (next_pos == merge_pos) {
                           *result = min_dbl;
```

```
                           return 0;
                       }
                       merge_pos++;
                   }
               }
           }
       }

       /* NOTREACHED */
   }

   #ifdef THRULAY_REPORTING_SAMPLE_LOOP

   #include <stdio.h>
   #include <strings.h>

   #ifndef NAN
   #define _ISOC99_SOURCE
   #include <math.h>
   #endif

   #define ERR_FATAL       0
   #define ERR_WARNING     1

   void __attribute__((noreturn))
       quantile_alg_error(int rc)
   {
       switch (rc) {
       case -1:
           fprintf(stderr, "Error: quantiles not initialized.");
           break;
       case -2:
           fprintf(stderr, "Error: NEW needs an empty buffer.");
           break;
       case -3:
           fprintf(stderr, "Error: Bad input sequence length.");
           break;
       case -4:
           fprintf(stderr, "Error: Not enough buffers for COLLAPSE.");
           break;
       default:
           fprintf(stderr, "Error: Unknown quantile_algorithm error.");
       }
       exit(1);
   }

   /**
    * Will read a sample data file (first and only parameter) whose
```

```
     * lines give two values per line (per received packet): measured
     * packet delay and packet sequence number (in "%lf %lu"
     * format). As an exception, the first line specifies the number
     * of packets actually sent.
     * NOTE: The code as written assume there is no newline on the last
     *       line.  FIXME.
     * Example:
     * ----
     10
     0.101 1
     0.109 2
     0.12 2
     0.10 4
     0.14 5
     0.15 6
     0.13 3
     0.09 7
     0.1 9
     0.091 8
     * ----
     *
     * To compile this sample reporting main():
     *
     * gcc -std=c99 -DTHRULAY_REPORTING_SAMPLE_LOOP reporting.c -lm
     *
     **/
    int
    main(int argc, char *argv[])
    {
        FILE *sf;
        /* 'Measured data' */
        const int max_packets = 65535;
        /* 'Received' packets*/
        int npackets = 0;
        uint64_t packet_sqn[max_packets]; /* Fill in with sample data */
        double packet_delay[max_packets]; /* Fill in with sample data */
        uint64_t packets_sent = 0;        /* Fill in with sample data */
        /* reordering */
        const uint64_t reordering_max = 100;
        char buffer_reord[reordering_max * 80];
        size_t r = 0;
        uint64_t j = 0;
        /* Stats */
        uint64_t unique_packets = 0, packets_dup = 0;
        double quantile_25, quantile_50, quantile_75;
        double delay, jitter;
        double packet_loss;
        char report_buffer[1000];
```

```
       /* Auxiliary variables */
       int i, rc, rc2, rc3;

       memset(packet_sqn,0,sizeof(uint64_t)*max_packets);
       memset(packet_delay,0,sizeof(double)*max_packets);

       /* Inititalize duplication */
       rc = duplication_init(max_packets);
       if (-1 == rc) {
           perror("calloc");
           exit(1);
       }

       /* Initialize quantiles */
       rc = quantile_init(1, QUANTILE_EPS, max_packets);
       if (-1 == rc) {
           perror("malloc");
           exit(1);
       }

       /* Initialize reordering */
       rc = reordering_init(reordering_max);
       if (-1 == rc) {
           perror("calloc");
           exit(1);
       }

       /* Open sample file */
       if (2 == argc) {
           sf = fopen(argv[1],"r");
       } else {
           fprintf(stderr, "no input file\n");
           exit(1);
       }

       /* Process sample input file. */

       /* The sender somehow tells the receiver how many packets were
          actually sent. */
       fscanf(sf,"%lu",&packets_sent);

       for (i = 0; i < max_packets && !feof(sf); i++) {

           fscanf(sf,"%lf %lu",&packet_delay[i],&packet_sqn[i]);
           /* Take care of common issue of ending the file with a
               newline; feof would not have been set but there is
               no more data.  Assume delay of 0.0 means we're done.
           */
```

```
            if (packet_delay[i] == 0.0) break;
            npackets++;

            /*
             * Duplication
             */
            if (duplication_check(packet_sqn[i])) {
                /* Duplicated packet */
                packets_dup++;
                continue;
            } else {
                /* Unique packet */
                unique_packets++;
            }

            /*
             * Delay quantiles.
             */
            rc = quantile_value_checkin(0, packet_delay[i]);
            if (rc < 0)
                quantile_alg_error(rc);

            /*
             * Reordering
             */
            reordering_checkin(packet_sqn[i]);
        }

        /*
         * Perform last algorithm operation with a possibly not full
         * input buffer.
         */
        rc = quantile_finish(0);
        if (rc < 0)
            quantile_alg_error(rc);

        rc = quantile_output(0, unique_packets, 0.25, &quantile_25);
        rc2 = quantile_output(0, unique_packets, 0.50, &quantile_50);
        rc3 = quantile_output(0, unique_packets, 0.75, &quantile_75);
        if (-1 == rc || -1 == rc2 || -1 == rc3) {
            fprintf(stderr,"An error occurred while computing delay "
                    "quantiles. %d %d %d\n",rc, rc2, rc3);
            exit(1);
        }

        /* Delay and jitter computation */
        packet_loss = packets_sent > unique_packets?
            (100.0*(packets_sent - unique_packets))/packets_sent: 0;
```

```
       delay = (packet_loss > 50.0)? INFINITY : quantile_50;
       if (packet_loss < 25.0 ) {
           jitter = quantile_75 - quantile_25;
       } else if (packet_loss > 75.0) {
           jitter = NAN;
       } else {
           jitter = INFINITY;
       }

       /* Format final report */
       snprintf(report_buffer, sizeof(report_buffer),
                "Delay: %3.3fms\n"
                "Loss: %3.3f%%\n"
                "Jitter: %3.3fms\n"
                "Duplication: %3.3f%%\n"
                "Reordering: %3.3f%%\n",
                1000.0 * delay,
                packet_loss,
                1000.0 * jitter,
                100 * (double)packets_dup/npackets,
                100.0 * reordering_output(0));

       printf("%s", report_buffer);

       /* Deallocate resources for statistics. */
       reordering_exit();
       quantile_exit();
       duplication_exit();

       fclose(sf);

       exit(0);
   }

   #endif /* THRULAY_REPORTING_SAMPLE_LOOP */
```

Appendix B. Example Report

This appendix only serves for illustrative purposes.

This report is produced by running the sample program in Appendix A on the sample input embedded in a comment in its source code:

```
Delay: 109.000ms
Loss: 10.000%
Jitter: 40.000ms
Duplication: 10.000%
Reordering: 22.222%
```

Authors' Addresses

Stanislav Shalunov

Email: shalunov@shlang.com
URI: http://shlang.com/

Martin Swany
University of Delaware
Department of Computer and Information Sciences
Newark, DE 19716
US

Email: swany@cis.udel.edu
URI: http://www.cis.udel.edu/~swany/