Measuring one-way metrics without a GPS

Augusto Ciuffoletti (augusto@di.unipi.it)
Dipartimento di Informatica - Università di Pisa

Abstract— **The forthcoming Internet will support applications that are able to tune their operation using data about the availability of network resources. A key issue in this perspective consists in the identification of significant metrics that can be measured without using specialized hardware and with low overhead. We define two such metrics, and we introduce a prototype implementation of a measurement tool.**

The forthcoming Internet will support applications that are able to tune their operation using data about the availability of network resources. Applications that demand the availability of important resources, like large storage, high bandwidth, or computing speed, will move resource consuming activities to sites where required resources are available.

The quality of the communication infrastructure is one of the most relevant resources, and one of the most difficult to monitor. We distinguish two classes of tools used for monitoring the communication infrastructure: *passive tools*, that process local application log-files in order to assess the quality of the communication infrastructure, and *active tools*, that *probe* the Internet using a benchmark traffic pattern.

Passive tools have the advantage of a minimal overhead on the monitored resource, but are not able to inform about resource availability: they evaluate current resource utilization, but cannot say whether a resource can be exploited further. An instance of such tool is the Unix command `netstat`, and its various user interfaces.

Active tools are more intrusive, since they present the Internet a certain traffic pattern, and observe how it is treated. *Probing* may entail a coordination among the sites that perform the measure, and the use of a common protocol. One of the most common tools in this class is the `ping` command, that relies on the existence of a `ping` server on the other end of the probed path, and uses the ICMP protocol [6].

As a general rule, the more intrusive the probe, the more accurate the information it provides: in the case of the `ping` command, the intrusion is extremely low, but the information returned by the probe is limited to the round-trip time of a special packet. More intrusive tools, like **pathchar** [2], suggest to saturate the measured resource for a very short time, in order to determine the saturation point, and therefore the available resources at a certain time.

Another important dichotomy of the measures provided by a monitoring tool divides *two-way* measures from *one-way* measures. The former are easier to obtain, but the image of the communication infrastructure returned by the latter have a finer resolution, and in many cases this is helpful: although many protocols use the full duplex mode offered by TCP, there are classes of protocols (e.g., data transfer protocols, like FTP) that use the two directions unevenly. The performance of the network in the two directions can be in fact different, due to the path selected in each direction, to the resource allocation policy of the administrations along each path, and to the relative performance of the server and of the client.

A common sense argument says that an accurate synchronization of the clocks on the two units is mandatory in order to obtain *one-way* measures. This conclusion is motivated by the fact that most *one-way* measures rely on the measurement of a *one-way* communication delay [1], and this operation relies on clock synchronization: as a general rule, to measure a *one-way* communication delay we need a clock accuracy which is one order better than the observed communication delay. For instance, an expected delay of 500 μsecs (that may be observed in an Ethernet network) would need a clock accuracy of 50 μsecs. Using simple arithmetics it is possible to prove that the probed links cannot be used to reach the required accuracy using a clock synchronization protocol (like NTP), and therefore the use of specialized hardware, like a GPS receiver, is needed.

Since the synchronization of the system clock to an external time reference is a complex and expensive functionality, the above common sense statement has the effect of associating a high cost to *one-way* measures.

The literature about the measurement of network resources is therefore split into two approaches: to provide *one-way* measures requiring accurate clock synchronization, or to provide *two-way* measures without requiring clock synchronization. I did not found any published paper in the land between the two extremes, although some experimental work is in progress, like the **UDPmon** prototype (see the home page at `http://www.hep.man.ac.uk/~rich/net/`).

This work tries to fill this gap, investigating two *one-way* metrics that can be measured by an *active tool*, without assuming clock synchronization and with a limited overhead: namely, *one-way bandwidth* and *jitter asymmetry*. We further introduce a prototype, that we call **netprobe**, that implements the results of the theoretical investigation.

I. One-way bandwidth

One of the key parameters that characterize a communication infrastructure between two sites, that we call *probe* and *target*, is *bandwidth*. Let $t_{rcv}(n)$ indicate the wire-time (see [5] for a definition of wire-time) when the n-th octet of a stream is received by the *target*, the *one-way bandwidth* observed from octet i to octet j is:

$$Bw(i,j) = \frac{j-i}{t_{rcv}(j) - t_{rcv}(i)} (\text{octets/sec}) \quad (1)$$

Note that *one-way bandwidth* does not correspond to *one-way throughput*. Throughput is bound to the time elapsing from the beginning of the send operation to the termination of the receive operation: if we indicate with δ the *one-way communication delay* of the first octet, the *one-way throughput* for a message of n octets is[1].

$$Tp = \frac{1}{\frac{\delta}{n} + \frac{1}{Bw(0,n)}} (\text{octets/sec}) \quad (2)$$

Since measuring the throughput entails the measure of a *one-way delay*, *one-way throughput* depends on clock synchronization. Instead, *one-way bandwidth* does not require clock synchronization, since it is computed as the difference between two local clock values.

One-way bandwidth provides a useful insight of the performance of the network between two sites, especially when large amounts of data are being transferred, and indicates how much information crosses the network during a time unit. This directly reflects the performance of each node along the path.

Although simple in principle, the measurement of *one-way bandwidth* exhibits some sources of inaccuracy, when implemented in practice. We list some of them, and evaluate their impact:

- *clock skew* - the system clock does not measure the real time, but a time that gradually drifts from real time, or that can be abruptly adjusted. The first fact induces errors of the order of a few parts per hundredth, and can be disregarded. Adjustments that occur during the measurement and induce a significant drift, or even a clock warp, make the measurement not adherent with reality.
- *process time vs. wire time* - a user process cannot read exactly the lapse between the beginning and the end of the communication: the operating system interface schedules visible events whose timing is related, but not identical, to the events for which we would like to measure the occurrence time.

As for the first point we can do nothing but recommend that the system clock is either left alone with its drift, or reasonably compensated.

The second problem can be addressed trying to minimize the interference of the operating system. The interference of the TCP protocol in this respect is hard to control, since communication is stream oriented, and buffer flushes can occur asynchronously with respect to *send* operations. A clever design of the measurement protocol could overcome this problem, but other features of the TCP protocol, like congestion control mechanisms, interfere with the measurement of a raw figure. Instead, the UDP protocol offers unbuffered communication, and this makes easier to measure relevant wire events.

[1] I am grateful to Petr Holub, Institute of Compt. Science of the Masaryk University, for having indicated me a mistake in the formula published in the PAM2002 paper. The problem has been corrected in this revision, and its scope is limited to the *one way bandwidth* formula

II. Jitter asymmetry

Jitter is a measure of the variation in communications delays: we compute jitter as the difference between a measured delay and an expected value, computed as a weighted average of past values.

Definition 1: Let $\delta(n)$ be the delay experienced by the n-th message of a series of messages, and $\overline{\delta}(n-1)$ the expected delay computed using the delays of the previous $n-1$ messages. The value of the *jitter* of the $n-th$ message is:

$$\text{jitter}(n) = \delta(n) - \overline{\delta}(n-1) \quad (3)$$

This definition is slightly different from the one used in RTP [7], and is more similar to other definitions, as in [3]. The use of this metric is traditionally related to playback applications, where the jitter is used for buffer configuration [4]. However, jitter can be used as a symptom of overloading, since it rapidly increases while the traffic tends to saturate the capacity of the network. This can be explained with the heavy use of buffers, and with the frequent route changes that occur when the network is overloaded, although not yet saturated.

Therefore, we claim that a communication infrastructure that exhibits frequent peaks of the communication delay is probably near to saturation. The information about jitter is therefore useful to monitor the performance of the network between two sites, but relies on the measurement of a *one-way communication delay*, which depends on clock synchronization.

A *jitter asymmetry* measurement *session* consists of a sequence of *rounds*, each measuring a *two-way* communication delay, also called round-trip time. Each round is identified by its index in the session, and is composed of two messages: a *forward* message, and a *backward* message. Let:

- $fwd(n)$ and $bwd(n)$ be the forward and backward messages in round n:
- $t_{snd}(m)$ and $t_{rcv}(m)$ the *wire times*, referenced to the clock on the unit where the event takes place, of the write of the first octet and of the read of the last octet of message m As we show in section IV, the application can measure an approximation (a *timestamp*) for these wire times;

We introduce two derived measurable quantities, in order to make the following equations more readable.

Definition 2:

$$\text{diff}(m) = t_{rcv}(m) - t_{snd}(m) \quad (4)$$

$$Os(n) = \frac{\text{diff}(bwd(n)) - \text{diff}(fwd(n))}{2} \quad (5)$$

The diff is the difference between the wire time, measured on the receiving unit, corresponding to the read of the last octet, and the wire time, measured on the sending unit, corresponding to the write of the first octet: it does not correspond to a communication delay, unless the clocks are perfectly synchronized. Os is a function of local wire times, it can be computed using measurable wire times: it

is a measurable approximation of the offset between the clocks of the measuring sites, as we show in appendix A.

Using the above quantities, the *jitter asymmetry* is computed as follows:

$$Ja(n) = 10 \operatorname{Log} \frac{\operatorname{diff}(fwd(n)) + \overline{Os}(n-1)}{\operatorname{diff}(bwd(n)) - \overline{Os}(n-1)} \qquad (6)$$

where $\overline{Os}(n-1)$ indicates an expected value for the metric Os, based on the $n-1$ previous measurements.

The fundamental property of this metric is stated by the following lemma:

Lemma 1:

$$(Ja(n) \geq 0) \Leftrightarrow (jitter_{fwd}(n) \geq jitter_{bwd}(n))$$

The relation stated by the lemma now binds quantities that are not observable: in fact, the jitter depends on one-way communication delays (see definition 1).

To understand the use of *jitter asymmetry*, we need to introduce an assumption supported by all experiments reported in the literature:

communication delays exhibit an average value that is near to the minimum, and exceptionally jump to values that are significantly higher than the average

Since we focus on significant variations, we conclude that they correspond to significant increments of the communication delay. Therefore the practical meaning of lemma 1 can be summarized as follows:

- $Ja \gg 0$ indicates an exceptionally long delay of the forward message;
- $Ja \ll 0$ indicates an exceptionally long delay of the backward message;
- $Ja \approx 0$ indicates stability or an infrequent balanced increase of the delay in both directions.

III. Computing expected values

In order to compute *jitter asymmetry* we need to be able to compute an expected value for Os. There are a number of techniques to solve this problem: [8] lists some of them.

We have opted for a filter that combines two well known techniques:

- a low-pass filter, that smoothes minor variations in the measured parameter;
- a peak clipper, that filters out measures that are apparently affected by exceptional events.

The two filters are embedded in a simple function that computes the next expected value knowing the new measure, and has an internal state consisting of two reals.

The *low-pass* filter consists of a recursive prediction:

$$\forall n \geq 0, \overline{x}_n = \begin{cases} x(0) & \text{if } n = 0 \\ \frac{\overline{x}_{n-1} * (k-1) + x_n}{k} & \text{otherwise} \end{cases} \qquad (7)$$

The value of k is called the *gain* of the filter.

We compute the recursive prediction of the input, and of the difference between the input and the predicted value. In summary we obtain each time a prediction for the measure, and for the variation of the measure. The *gain* of the two filters can be configured separately

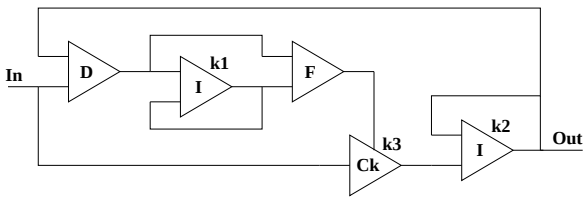

Fig. 1. Block diagram of the predictive filter

The *peak clipper* filter is implemented using the result of the application of the *low-pass* filter to the variation of the measure. The rate between the measured variation, and the predicted value is computed. The result is compared with a threshold parameter: when the variation rate exceeds the threshold, the input data will be used to feed the *low-pass* filter of the variation only; the expected value will not be affected by measures that exceed the expected variation, since they are considered a consequence of an exceptional event.

In figure 1 we show a diagram of the filter:

- **D** blocks perform a difference between the input values.
- **I** blocks implement the low-pass filter with the shown gain k; the feedback line is also shown.
- **F** blocks compute the rate between the two inputs.
- **Ck** blocks transfer the input value to the output only if the control input (on one side) is below the shown threshold k.

The overall behavior is controlled by the *gain* of the two filters ($k1$ and $k2$ in figure 1) and by the threshold ($k3$ in figure 1): their values do not appear to be critical, and are a matter of optimization.

In figure 2 we show the results of an experiment with a value of 10 for the two *gains*, and a threshold of 2 dB. The input value Os, exhibits a jump of nearly 5 msecs, due to a periodic clock adjustment. The filtered value, shown as a dotted line, amortizes but follows the change.

During the interval between 96 and 98 seconds from the beginning of the session the expected variation (not shown in figure) is about 0.01 msec. After time 98 the input value changes, and the variation exceeds the threshold: the new input affects only the predicted variation, and the output value is left unaltered. After time 99, the filter "learns" an incremented variability of the input values, and finally follows the new stabilized value. The expected variation at time 100 is more than 3 msec. We conclude the case study observing that although the filter parameters are not appropriate (the threshold should be increased, and the gain of both low-pass filters should be reduced) the behavior of the filter is still acceptable.

We use the filter to predict the value of Os, the measured clock offset, whose filtered output is used to feed the computation of *jitter asymmetry*. We assume that clocks are characterized by small periodic adjustments, whose result, in the value of *jitter asymmetry*, is shown in figure 3. When the observed variation exceeds the threshold, the round is considered anomalous, and its timing is not considered in the prediction of Os.

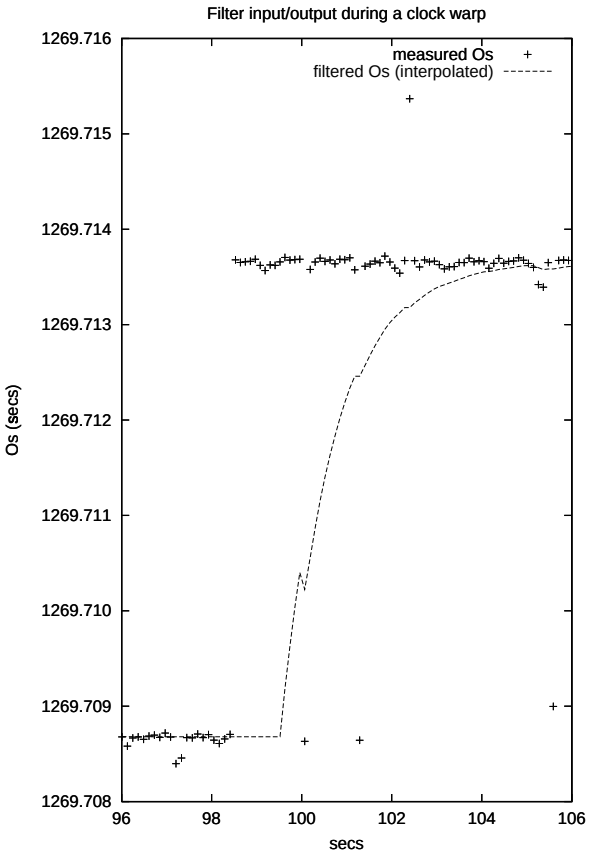

Fig. 2. Performance of the predictive filter during an experiment

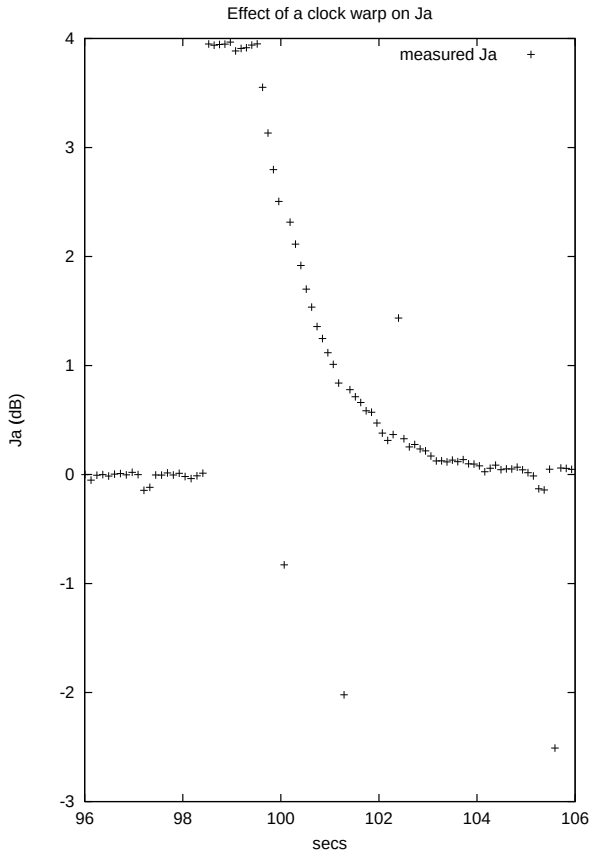

Fig. 3. Jitter asymmetry computed using predicted *Os* in figure 2

IV. Timestamping wire events

Defining a metric in terms of wire times is not sufficient to ensure that it can be measured in practice: we further have to explain how, and under which conditions, relevant wire times can be measured using a given time reference. When this is possible, we say that the wire time can be *timestamped*. In principle, *timestamping* is possible when the wire time is associated to a wire event that in its turn is related to an event that is observable from the application that wants to timestamp the wire event. Since this statement never holds in practice, we also have to bind the errors that affect a certain *timestamp*.

In a one-way message transfer we can identify four relevant wire times corresponding to the *write* and the *read* of the first and of the last octet of the message. We briefly discuss how they can be *timestamped* using the observable events related to the `sendto` and `recvfrom` system calls.

A timestamp taken immediately *before a* `sendto` *call* indicates a time reasonably near, and necessarily preceding, the time when the first octet is written to the network interface. Errors are due to the time needed to move the message to the internal buffers, and by a time-slicing introduced by time shared operating systems. The former can be regarded as negligible if the measuring unit is not overloaded with network traffic, while the latter should not occur inside the `sendto` primitive (while interrupts are normally disabled) and unfrequently occurs during the timestamping operation. Thus a timestamp taken immediately before a `sendto` operation is a good indication of the *write wire-time* of the first octet.

A timestamp taken immediately *after a* `sendto` *call* cannot be associated to a wire event: if the user process that issues the call is immediately re-scheduled, this timestamp might be taken even before the *write wire-time* of the first octet!

A timestamp taken immediately *before a* `recvfrom` *call* can be associated to the *read wire-time* of the first octet if that octet is already in the receive buffer when the call is issued: otherwise it simply marks the beginning of a wait that will terminate when the first octet is ready in the input buffer.

A timestamp taken immediately *after a* `recvfrom` *call* is a good indicator of the *read wire-time* of the last octet. As in the case of the timestamp taken immediately before a `sendto`, errors are due to the internal buffering implemented by the operating system, and to infrequent time slicing preemptions occurring during timestamping.

V. Protocol definition – The main loop

The scheme in figure 4 illustrates a *round* of a protocol that measures *one-way bandwidth* and *jitter asymmetry* using the *timestamps* marked as Ti: the unit on the left is the *probe*, and the other is the *target*, and we want to characterize the unidirectional communication from the former

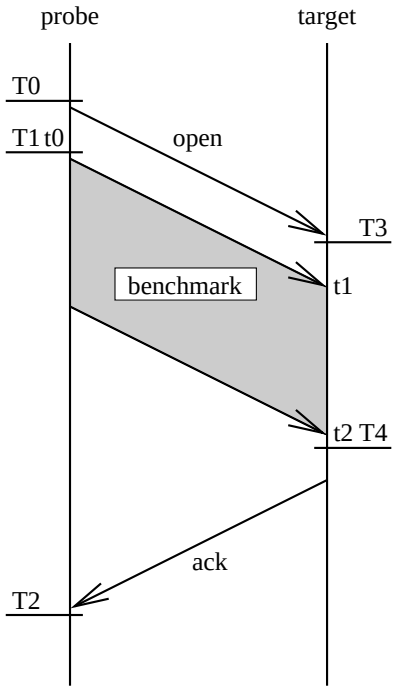

Fig. 4. Datagrams exchange during a round

to the latter.

A *round* is composed of three UDP *datagrams* sent through the same socket pair:

- a short **open** datagram informs the *target* that the round is started;
- a following **benchmark** datagram shapes the *one-way bandwidth* measurement, since the measure of this metric changes with the size of the **benchmark**;
- a final **ack** datagram is used to measure, in conjunction with the **open** datagram, the round-trip delay needed to the *jitter asymmetry* measurement, and to conclude the *round*.

Each of the three datagrams carries a 16 octets (128 bits) header:

- a two octets field (`unsigned short int`) for the type of the datagram:
  0 = OPEN, 1 = BENCHMARK, 2 = ACK;
- a two octets field (`unsigned short int`) for the size n of the **benchmark** datagram in octets. This information is included also in the **open** and **ack** datagrams;
- two four octets fields (`struct timeval`) for the timestamp of the round: one field is for the seconds, and one for the fractions of a second. This value is taken as the identifier of the *round*, and has no relevance in the measure. It is reported in each of the three datagrams.

The three datagrams are built as follows:

- the **open** datagram contains only the header;
- the **benchmark** datagram contains the header, followed by a sequence of randomly generated octets, up to the desired size;
- the **ack** datagram contains the header, followed by the timestamps T_3 and T_4. The timestamps are represented as double precision floats, and are marshalled (packed in perl's terminology) by the *target*, and unmarshalled (unpacked) by the *probe*.

The units take a number of timestamps. The *probe* reads the local clock immediately before the `sendto` of the **open** message (T_0), immediately before the `sendto` of the **benchmark** message (T_1) and immediately after the `recvfrom` of the **ack** message (T_2). The *target* reads the local clock immediately after the `recvfrom` of the **open** message (T_3), and immediately after the `recvfrom` of the **benchmark** message (T_4).

The timestamps are used to estimate the delay between the read of the first octet and the read of the last octet of the *benchmark* packet, and the round-trip delay. A problem arises for the measurement of wire time $t_{rcv}(benchmark)$, which is needed to compute the *one-way bandwidth*. The solution we propose is based on the claim that an appropriate design, that we discuss in the next section, may ensure that the timestamp T_3 as an approximation of t_0: to ensure this, the protocol should enforce the following properties of the timing of wire events:

(no shuffling) datagrams are received in the same order they are sent, and

(target ready) T_3 approximates the wire time of the read of the first octet of the second datagram in the current round.

The timestamps collected during a round are used to compute the observed *clock offset*, the *one-way bandwidth* and *jitter asymmetry*:

$$Os = T_3 - T_0 - \frac{T_2 - T_0 - (T_4 - T_3)}{2} \text{ (secs)} \quad (8)$$

$$Bw = \frac{n}{1000(T_4 - T_3)} \text{ (kbytes/sec)} \quad (9)$$

$$Ja = 10\, log_{10} \frac{(T_3 - T_0) - \overline{Os}}{(T_2 - T_4) + \overline{Os}} \text{ (dB)} \quad (10)$$

where n stands for the size of the benchmark packet, $\overline{Os}$ is the predicted value of the offset, and Ts are the timestamps shown in figure 4.

In figures 5 and 6 we show the state chart of the main loop of the two processes: each box represents a state, and transition between the states is regulated by the occurrence of an event, and entails the execution of an action, represented as follows:

$$\frac{\text{controlling event}}{\text{action}}$$

When an event or an action is timestamped, the name of the timestamp is indicated between parentheses (for instance, **send open(T0)** records the timestamp T0 before the `sendto` call, and **receive open(T3)** records the timestamp T3 after the `recvfrom` call).

A round terminates when the probe receives the **ack**, or the timeout of the **receive ack()** expires.

The next round can be executed immediately, or rounds can be scheduled periodically . In the former case, the behavior is extremely intrusive, and similar to an ftp session: the effect is to saturate the available bandwidth for

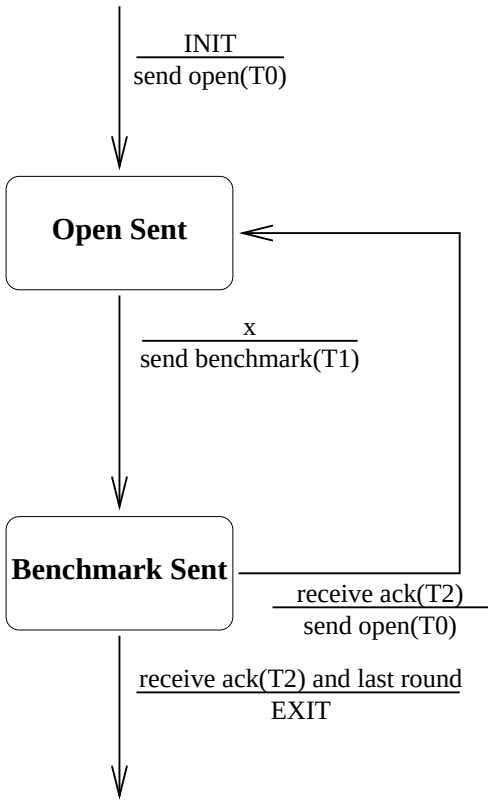

Fig. 5. The *probe* process state chart

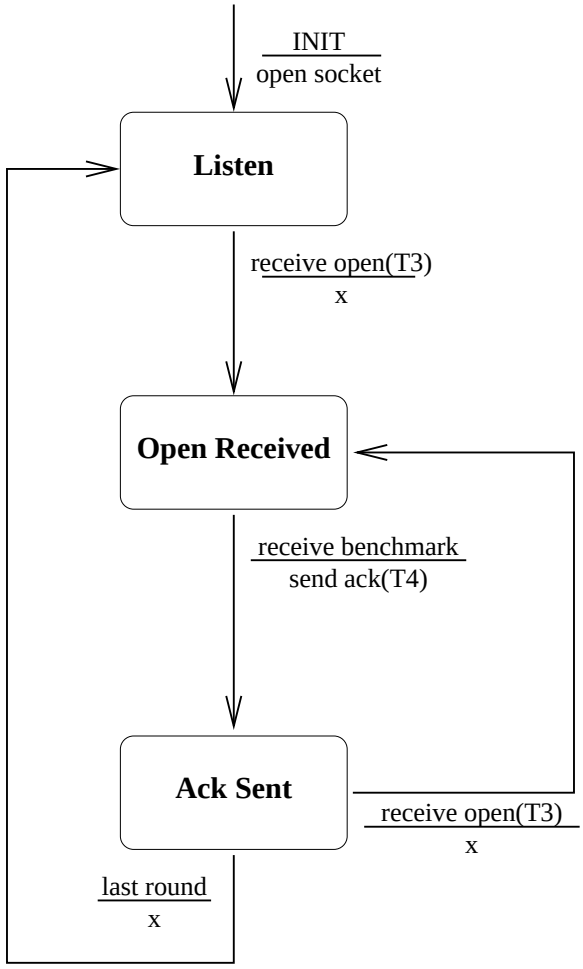

Fig. 6. The *target* process state chart

a certain time, and to measure *one-way bandwidth* and *jitter asymmetry* under such conditions. In the latter case the measurement is far less intrusive, and is adequate as a routine probe. In either case each round produces a value for *one-way bandwidth* and for the *jitter asymmetry*. This differentiates our tool from **pathchar** and **udpmon**, that are based on a burst of UDP packets.

VI. Protocol definition – Exceptions handling

To ensure the *no shuffling* requirement, we tag each message with a progressive identifier: when the *target* detects a shuffling, it aborts the round sending a negative acknowledgement.

To ensure the *target ready* requirement, the *probe* sends the **open** and the *benchmark* datagrams consecutively, and checks that the two timestamps T_0 and T_1 are sufficiently close. In case a relevant interval between the timestamps is detected, the measure is considered not valid. If the network carries together the two messages, we can assume that when the *target* leaves the `recvfrom` of the **open** datagram, the first octets of the **benchmark** datagram are already in the receive buffer.

The protocol recovers the measurement session after abnormal events like the loss of a datagram. This is obtained with timeouts and emergency exits.

The most frequent case is the loss of the **benchmark** datagram. To cope with this event, the *probe* sets a timer after sending the **benchmark** (default = 0.5 secs). If the timer expires before the **ack** datagram is received, the **benchmark** is re-sent a fixed number of times (default = 10 times), resetting the T_1 timestamp. If no ack is received, the next **open** is sent.

At the *target*, a timer is set, after the receive of a **open** to the timeout of the *probe*, times the number of retries, augmented by 10% (default 5.5 secs). When the timeout expires, the *target* re-enters the wait for the next **open**. Duplicate **benchmark** datagrams are discarded by the *target*, since they are received while the *target* is waiting for a **open**, or since the timestamp recorded in the header does not match the timestamp of the last **open**.

When the *probe* times out the first attempt, the measure is altered, and the *probe* should consider the message as lost. The successive round is also altered, since the round-trip measure will encompass also the waiting time at the *target*.

In figures 7 and 8 we show the state charts of the *probe* and *target* processes limited to the handling of abnormal events. We omit from the state chart of the probe that:

- the *probe* terminates when it finds the *target* socket closed, and
- under certain conditions the timing of the round is considered unsuitable for measuring the *one-way bandwidth* and the *jitter asymmetry*.

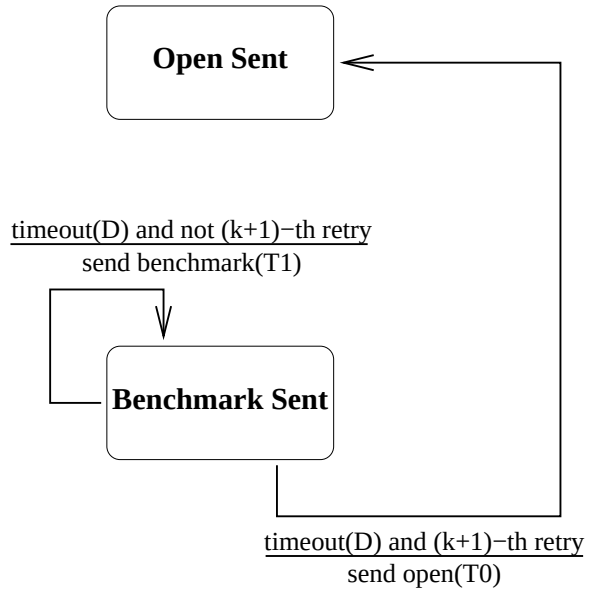

Fig. 7. The *probe* process state chart (exceptions handling)

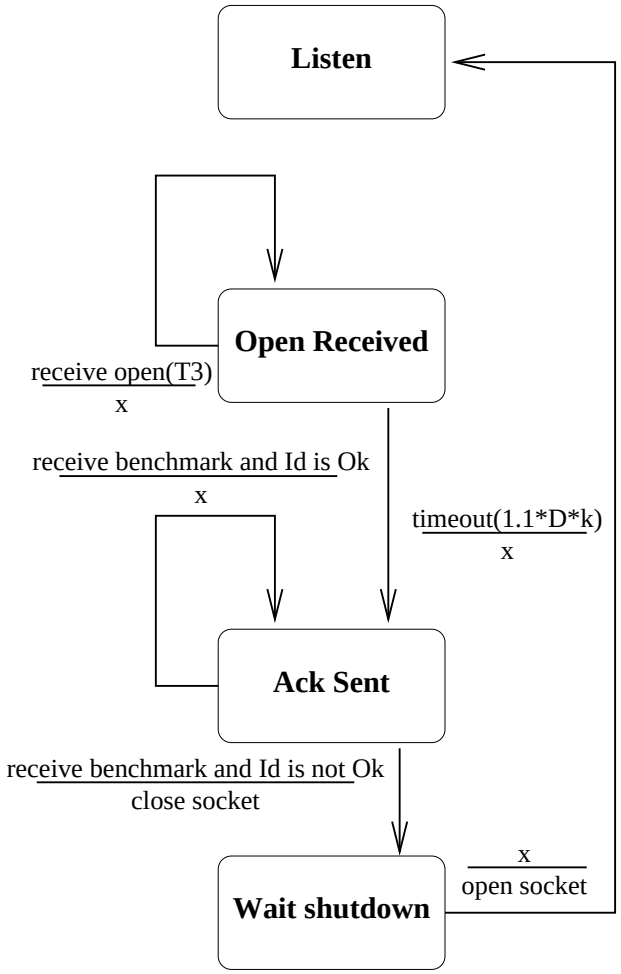

Fig. 8. The *target* process state chart (exceptions handling)

VII. Prototype implementation and experimental results

The above protocol has been implemented in Perl as a tool for the Debian/Linux platform, that we call *netprobe*. The tool consists of two programs: `probe.pl` running on the *probe* host, and `probed.pl` running on the *target* host. The latter program is configured as a daemon.

The main purpose for this experimental step is to prove that the metrics and the protocol illustrated in theory can be effectively implemented. Here we do not want to introduce a *production* network monitoring tool, but simply prove that the ideas illustrated in this paper can be used in practice. However, we believe that, with a few modifications and under not too restrictive assumptions, the prototype can be usefully applied to network monitoring.

The experimental results that we present here as not meant to demonstrate the application of the tool in a working environment: instead we use a simple benchmark system in order to check if the measures returned by our tool are consistent with a theoretical expectation. In addition, we opted to produce a tightly packed sequence of rounds, in order to check the internal timing of the protocol under stress. As we say at page 5, this results in an extremely intrusive behavior of the tool, and it is not an appropriate option for a routine monitoring in a working environment.

The following results have been obtained in a 10 Mbps Ethernet branch, and the only user of the network was *netprobe*. The experiment consisted in a sequence of 1000 tightly packed rounds, with a benchmark packet of 1000 octets. The delay between two successive rounds was approximately 3 msecs, that were spent in the arithmetic calculations needed to compute and store the values of the *one-way bandwidth* and *jitter asymmetry*, as illustrated in the previous sections. No special care was paid to optimize such computation.

The expectations are the following:

- as for the bandwidth, we consider that a benchmark packet fills partially the Ethernet frame, whose MTU is 1500 octets. Since each 1500 octets frame takes approximately 1.2 msecs to be transmitted on a 10Mbps line, we might expect that this is the time needed, in our experiment, to exchange 1000 octets. This justifies an expected bandwidth of approximately 800 Kbytes/sec. The approximation does not consider the length of the headers and buffering delays;
- since the hosts and the network are idle, except for the activity inherent the experiment, we should expect a jitter asymmetry concentrated around the value of 0 dB.

The experiment lasted approximately 5 seconds, and in the following figures we analyze an interval of 0.4 seconds centered around the 4th second, in order to have more readable graphs.

The measured clock offset Os (see figure 9) ranges around the value of 2 hours to simulate unsynchronized hosts. The clock offset remains constant during all the experiment, thought we may expect that it is periodically adjusted, and therefore might exhibit the kind of behavior illustrated in figure 2. It is interesting to note that the clock

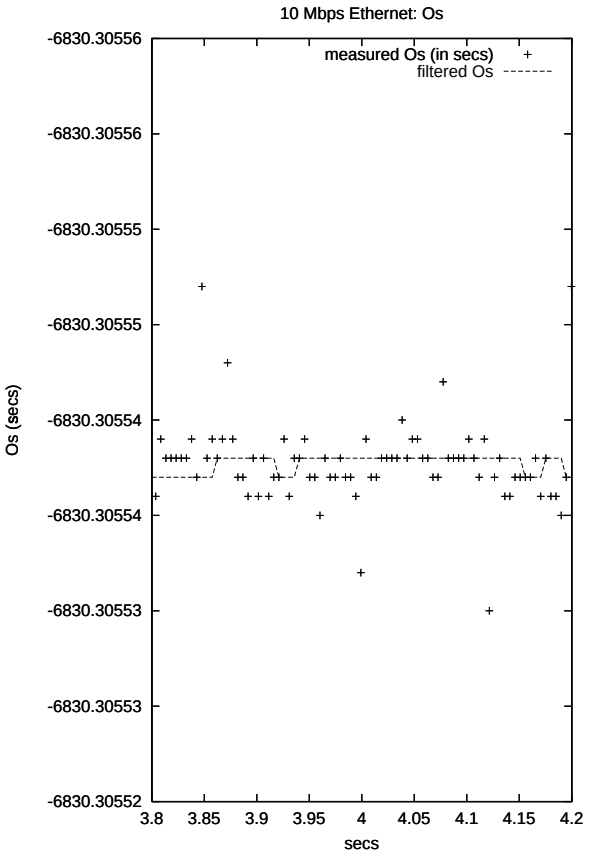

Fig. 9. Measured and filtered clock offset

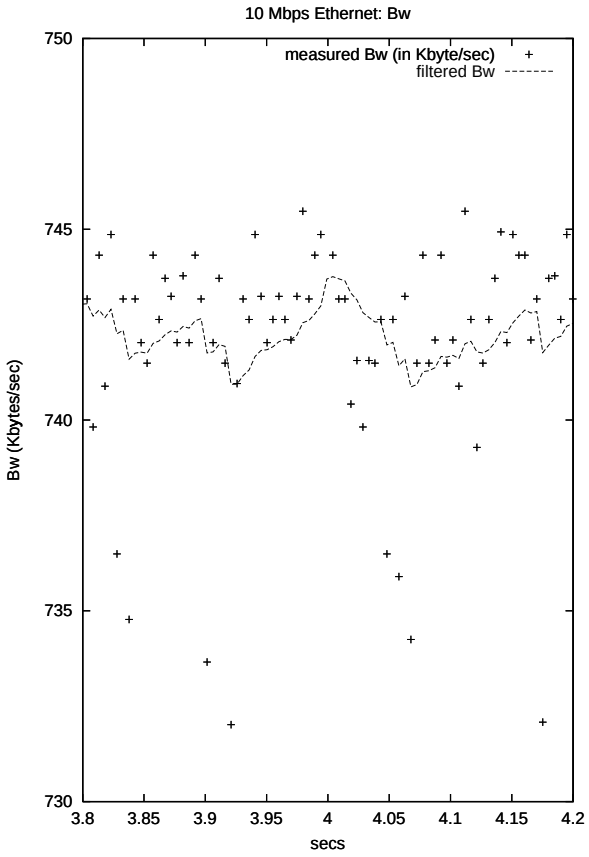

Fig. 10. Measured and filtered bandwidth

offset accuracy is excellent: the measured values exhibit a *quantum* of 1 μsec, due to the approximation introduced by the `print` statement. This proves that Perl is appropriate for measuring short time intervals, when using the `gettimeofday` Unix primitive.

As for the measured *one-way bandwidth* (see figure 10), it is approximately constant during the experiment, and the measured value (approximately 740 Kbytes/sec) is in excellent agreement with the expected value of 800 Kbytes/sec. Other experiments with different benchmark packet sizes confirm that bandwidth measurements are valid.

Finally, the measured *jitter asymmetry* (see figure 11) exhibits the expected concentration around the zero, justified by the absence of load in the system.

Another experiment of 1000 tightly packet rounds was run using a benchmark packet of 1400 octets, in a lightly loaded 10Mbps Ethernet. The graphs in figure 12 and 13 show the frequencies of the measures of *one-way bandwidth* and *jitter asymmetry*.

The values of *one-way bandwidth* are concentrated around a value of 600 Kbytes/sec, and reflect the fact that, although the frames are better exploited by a benchmark length of 1400 octets, the existing load lowers the available bandwidth.

As for the *jitter asymmetry*, we observe that positive and negative branches have different shapes: this is an evidence that the two branches represent distinct aspects of the data transfer. We infer that they represent the jitters experienced by forward and backward messages. The positive branch, that represents forward jitter, falls steeply and smoothly. The negative branch, representing backward jitter, presents relevant peaks at 8 dB, corresponding to a delay 6 times longer than expected.

Conclusions

We have illustrated two one-way metrics whose measurement does not need an accurate clock synchronization and exhibits a low overhead on the communication infrastructure. Their combination gives an insight of the quantity and quality of resource availability in the communication infrastructure between two sites.

References

[1] G. Almes, S. Kalidindi, and M. Zekauskas. A one-way delay metric for IPPM. Technical Report rfc2679, Network Working Group, 1999.

[2] Allen B. Downey. Using pathchar to estimate internet link characteristics. In *Measurement and Modeling of Computer Systems*, pages 222–223, 1999.

[3] D. Ferrari. Client requirements for real-time communication services. Technical Report rfc1193, Network Working Group, November 1990.

[4] M. Garrett and M. Borden. Interoperation of controlled-load service and guaranteed service with ATM. Technical Report rfc2381, Network Working Group, August 1998. bucket.

[5] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. Technical Report rfc2330, Network Working Group, May 1998.

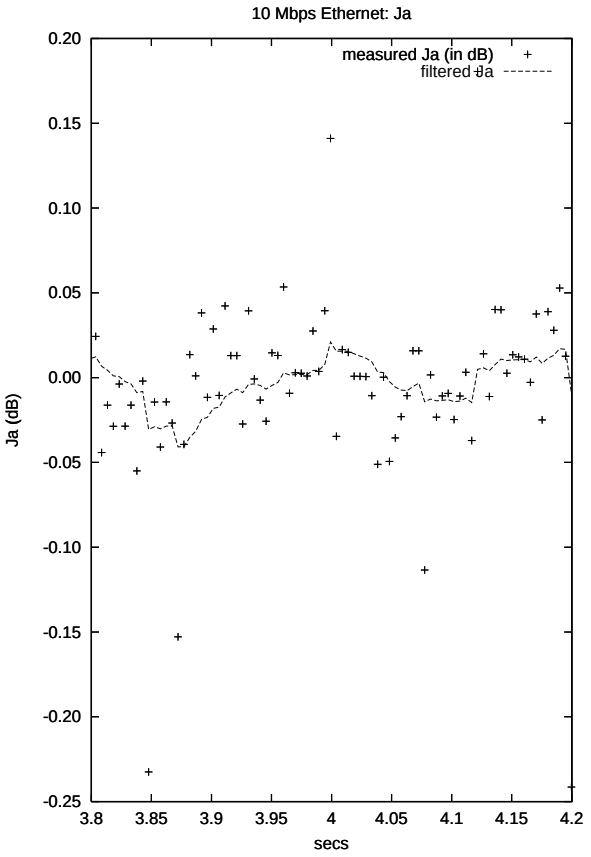

Fig. 11. Measured and filtered jitter asymmetry

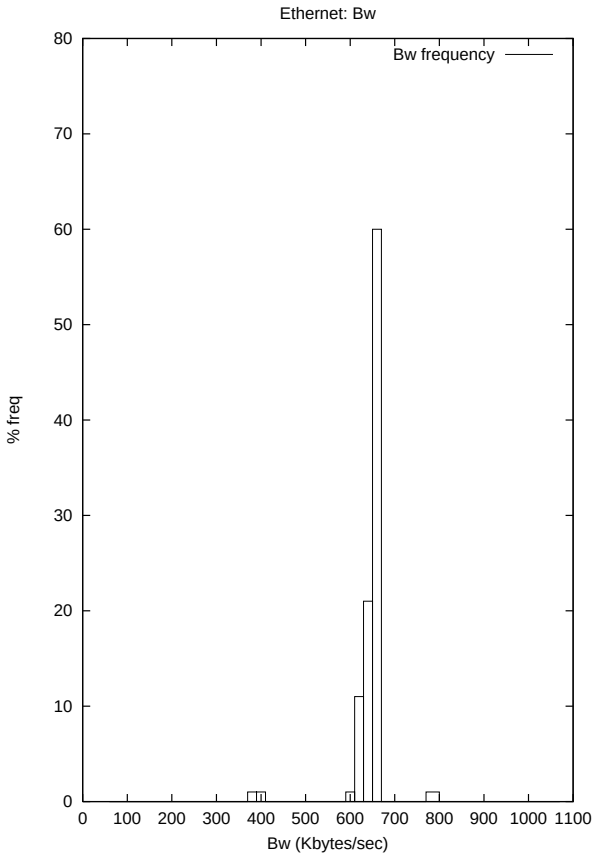

Fig. 12. Distribution of *one-way bandwidth*

[6] J. Postel. Internet control message protocol. Request for Comment 792, Network Working Group, September 1981.
[7] H. Shultzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. Request for Comment 1889, Audio-Video Transport Working Group, January 1996.
[8] Rich Wolski. Dynamically forecasting network performance using the network weather service. Technical Report TR-CS96-494, University of California at San Diego, 1996.

Appendix

I. Proof of lemma 1

We introduce the following non-measurable quantities:
• $\Delta = clock_{pb}(t) - clock_{tg}(t)$ be difference between the local clocks of the target and of the probe, that we assume constant during a session;
• $\delta(m)$ the delay, referenced to the Universal time, between the write of the first octet and the read of the last octet of message m.

The above figures are related by the following equalities:
Lemma 2:

$$\begin{aligned} \text{diff}(fwd(n)) &= \delta(fwd(n)) - \Delta \\ \text{diff}(bwd(n)) &= \delta(bwd(n)) + \Delta \end{aligned}$$

that can be proved using the definition of Δ and equation 4.

We assume that Δ is not measurable, since its computation is functionally equivalent to the synchronization of the clocks of the two sites. As a consequence, the two δ are unknown as well.

We use the equations in lemma 2 to rewrite the definition of Os in terms of unobservable delays:

$$Os(n) = \frac{\delta(bwd(n)) - \delta(fwd(n))}{2} + \Delta \qquad (11)$$

Note that, when the forward and backward delays are identical, Os corresponds to the clock offset; the error of this measure is bound by the whole round-trip.

Next we use equations 11 and 4 to rewrite the value of Ja in terms of non measurable quantities, and, using a property of the logarithms, we obtain:

$$\begin{aligned} Ja(n) \geq 0 \quad \Leftrightarrow \quad & \delta(fwd(n)) - \overline{\delta}(fwd(n-1)) \geq \\ & \delta(bwd(n)) - \overline{\delta}(bwd(n-1)) \end{aligned}$$

By the definition of *jitter*, in equation 3:

$$Ja(n) \geq 0 \Leftrightarrow jitter_{fwd}(n) \geq jitter_{bwd}(n)$$

□

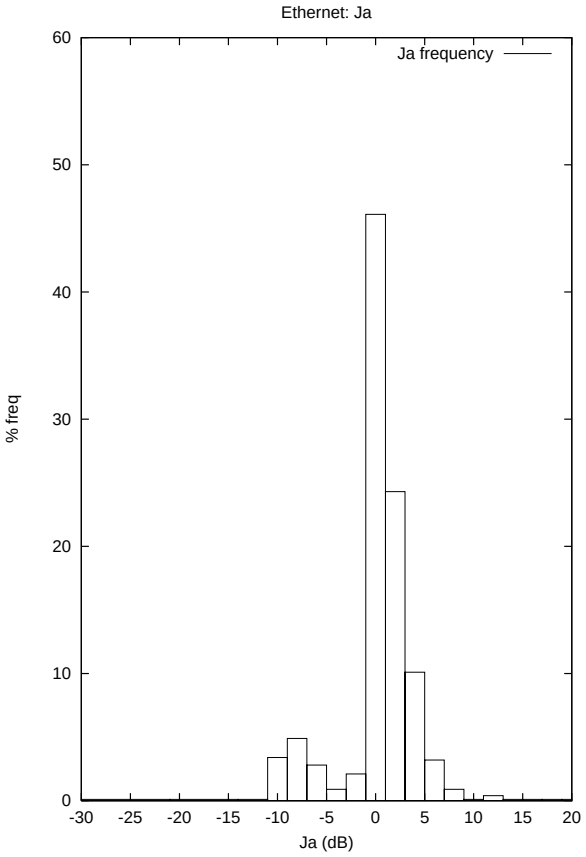

Fig. 13. Distribution of *jitter asymmetry*