Università degli Studi di Catania

Dipartimento di Ingegneria Elettrica, Elettronica e Informatica

Corso di Laurea Magistrale in
Ingegneria Informatica

Passive analysis of queueing delay in the Internet

TESI DI LAUREA DI :
Andrea Giuseppe Araldo

RELATORE:
Chiar.mo Prof. Giacomo Morabito

CORRELATORE:
Chiar.mo Prof. Dario Rossi

Anno Accademico 2012/2013

A mio padre e mia madre

Contents

Introduction

In the last 15 years Internet has evolved at an astonishing speed. The technological advances, particularly in the physical layer, now permit to reach very high bandwidth and very small delays that no one could imagine some years ago. But saying that the user experience has "improved" is, at least, too superficial. While in the past a user could be completely satisfied downloading a prevalently textual web page in a reasonable time, now the user needs to run a bunch of applications that exchange data on Internet, each of them with different requirements: she wants to participate in a video skype call, to synchronize her data on the cloud, to watch youtube videos, to download large contents with peer-to-peer applications.

Roughly speaking, to satisfy users, Internet has to provide much more bandwith and much less delay than before. On the other hand, much more users use Internet for much more time during the day and the mass of data traveling on Internet has hugely increased.

To complete the picture, we have to take into account that users access Internet with different devices (desktop pcs, mobiles, ...) attached to different layer 2 local networks (ethernet, 802.11, ...), so that the paths that packets traverse are very heterogeneous.

The implications of each piece of the puzzle just depicted will be clearer in the following chapters. For now, suffice it to say that Internet has to face very complex challenges and maybe it's not the "best Internet" we can have. Although from a different perspective, [Han06] states that "Internet only just works".

With this picture in mind, we can get convinced that we need conceptual and operational tools to evaluate the Internet performances, to find possible issues and to investigate their root causes.

In this work, we develop a methodology to study the queueing delays over the Internet, in particular the time that the packets pass in the buffers of the user access points, modem routers, end-host software stacks and network interfaces. Our

methodology is passive, i.e. it can be applied to traces of real traffic, either recorded in the past or sniffed on line, and does not require to alter the real traffic with the injection of other packets. We make our open source implementation of the methodology, based on Tstat [FMM+11], validate it and evaluate it in terms of computation overhead. We then apply the tool to some real traces and statistically characterize the observed queueing delay. We propose also a methodology to correlate the amount of queueing delay seen by the hosts with the applications running on them, with the aim to discover the applications that are more likely to induce high queueing delays and, on the other hand, the applications that are more likely to suffer high queueing delays, taking into account their requirements. We apply this methodology to the output of Tstat processing and provide some results.

It's worth highlighting that our focus is to build a solid methodology, rather than providing full-blown measurement campaign.

Chapter 1 describes the so called "bufferbloat", i.e. how the TCP congestion control coupled with the large size of buffers of the user access points, modem routers, end-host software stacks and network interfaces can lead to high queueing delays.

Chapter 2 focuses on the methods to measure the queueing delays. After a brief survey of the existing approaches, our methodology is depicted.

Chapter 3 outlines our implementation on Tstat, provides the validation and performance analysis results and highlights the Tstat parameters that can influence the quality of the results.

Chapter 4 shows a statistical characterization of the queueing delay computed on some real traces. The methodology to find the correlation between queueing delays and running applications is proposed and some results are showed

1. The bufferbloat issue

People usually think that the higher is the bandwidth that the Internet Service Provider offers, the better Internet access will be. It is why Cheshire exclaims "It's the latency, stupid" ([Che96]), to point out that delay, more than bandwidth, impacts on the user experience.

Despite the steady growth of link capacity, the delays that users experience can be still too high, considering also that users are becoming more and more demanding and expect a quick interaction when performing their jobs (e.g. when browsing or playing online videogames): "people time out before packets" ([GN11]). Jim Gettys (Bell Labs, Alcatel-Lucent) and Kathleen Nichols (Pollere Inc.) claim that the culprit is bufferbloat ([GN11]) which is "the existence of excessively large and frequently full buffers inside the network".

1.1. Why large buffers can alter the TCP congestion control

Buffers are memory areas of network devices in which the packets are enqueued before going out on the output link. They are located in the network card of computers, mobile phones (or any other device that has access to a network), user access points, switches, routers. If the arrival rate of the packets that have to go out on a certain link is more than the rate at which they are serviced, the queue in the buffer grows. If the queue fills the entire buffer, there is no more space for new packets that have to be dropped.

Without going into details, when a TCP sender observes a packet loss, it retransmits the packet and decreases its sending rate. The rationale behind it is that a possible cause of the loss is a network congestion[1] that a non-decreasing sending rate may

[1]Section 3 of [Jac88] states: "Packets get lost for two reasons: they are damaged in transit, or

worsen. If the losses are heavy, a network may become clogged with retransmitted packets and the data throughput may slow to trickle. For this reason, in 1986 the Internet had its first documented *congestion collapse* which led to a sudden factor-of-thousand drop in data throughput ([Jac88]). Since that time, other algorithms were introduced in TCP to avoids similar events but heavy losses may be still a problem.

Because the memory is significantly cheaper than in the past, manufacturers have increased the size of the buffers to prevent any packet loss. In some cases, the result is that the buffers are oversized and this may prevent TCP to work properly. Despite heavy losses are to be avoided, *some* losses are not only not-a-concern but they are vital to let TCP congestion control calibrate the sending rate. Indeed, if the buffers are large, they can host very long queues. The packets are not lost, but they pass a very long time in the buffers before going out. The TCP sender notices no loss and does not decrease its rate[2]. In this way the queues become longer and longer and the queueing delays higher and higher. As a consequence, users can perceive high latency that hampers QoE of Web browsing ([BOB12], [SdDF+11]), multimedia ([HPC+12], [CHHL06]) or peer to peer applications ([TR11]).

A closer look at TCP congestion control

We have just given a qualitative picture but we can go more deeply into the TCP congestion control details (see [Jac88]) to correlate them with the bufferbloat.

The TCP sender maintains a value *cwnd* (congestion window). The sender can send a number of packet at most equal to[3]:

$$n = \min(cwnd, rwnd)$$

without waiting for the respective acknowledgements. In other words, if the last acknowledged segment is the i-th, it can send the segment $i + 1$-th, $i + 2$-th, ..., $i + n$-th. Suppose, for the sake of simplicity, that always $cwnd < rwnd$, so that

the network is congested and somewhere on the path there was insufficient buffer capacity. On most network paths, loss due to damage is rare (1%) so it is probable that a packet loss is due to congestion in the network. ”

[2] Actually, it's not completely true: to be more precise, the sender does not decrease its *cwnd*, as explained later

[3] *rwnd* is a value imposed by the TCP receiver and it's called “receiver advertised window”

$n = cwnd$. $cwnd$ determines the maximum packet rate that is:

$$r = \frac{cwnd}{RTT} \tag{1.1}$$

where RTT is the round trip time, i.e. the time that passes from the sending of the $i+1$-th segment and the arrival of the relative acknowledgement. During the RTT, the sender is allowed to send at maximum $cwnd$ packets.

The $cwnd$ is initialized to 1 and is updated during a TCP connection. After the connection has been established, the *slow start* phase starts. In this phase, $cwnd$ is incremented by one for each received acknowledgement. When $cwnd$ reaches a threshold (often indicated as $ssthres$ - slow start threshold) the *congestion avoidance* phase starts: $cwnd$ is still increased, but with a slower rate ($cwnd$ is incremented by 1 every $cwnd$ segment received).

If there is a loss, the congestion window is reduced [4] and consequently the sending rate decreases.

Now we can be more precise in depicting why large buffers can hamper the TCP congestion control: even if the queues on the networks are very long, a TCP sender continues to increase its congestion window (because no loss is detected) and does not properly limit its sending rate, making the queues even longer and thus exacerbating the queueing delay. We should observe that high queueing delays imply an increase of the RTT and (1.1) guarantees that the sender implicitly reacts to them by decreasing the sending rate. But unfortunately, this is not always enough.

The mechanism to update the $cwnd$ permits to the TCP sender to have a coarse estimate of the available bandwidth along the end to end path. The main goal is to use all this bandwidth for the transmission. An interesting interpretation of what just explained is the following: while TCP was originally conceived to efficiently fill the pipe between the sender and the receiver (i.e. to efficiently use the available bandwidth), now it ends up rapidly filling the buffers. Large buffers "distort" the image of the pipe that TCP builds up, making it believe that there is more available bandwidth than there is in the reality. In other terms, some bytes of buffer memory is mistaken for available bandwidth.

[4] The way this reduction is operated depends on the TCP implementation. See http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm for more details.

Large buffers and *cwnd* decrease

Suppose that there is an "overbuffering" in the path that goes from a TCP sender to a TCP receiver. To avoid a long queue in this path, we would prefer sender decrease its *cwnd*. TCP sender does this only if a loss event occurs. The loss is detected at the sender side through these two "loss evidences":

- the reception of a triple duplicated acknowledgement or
- the elapsed RTO (Retransmission Time Out)

A receiver sends a duplicated acknowledgement when it has correctly received and acknowledged the i-th segment, and receives a $i+k$-th segment, with $k > 1$. In this case, the receiver sends another acknowledgement for the i-th segment. When the sender receives this last acknowledgement, it can infer that the segment $i+1$ was not correctly delivered, but this does not necessarily mean that the segment was lost. It is possible that some segments arrive at the receiver out of order, and eventually the $i+1$ segment may be correctly delivered. This is why the reception of *three* duplicated acknowledgments is interpreted as a loss evidence.

For the description of the Retransmission Timer, we refer to [PA00]. Every time a data segment is sent[5], if the timer is not running, the sender starts it. When an acknowledgement related to data waiting to be acknowledged is received, the timer is restarted. If the Retransmission Timer expires, this means that the sender has not received new acknowledgements for a time equal to RTO. This is interpreted as a loss evidence.

We now show that in case of overbuffering both the loss evidences seldom arise and thus *cwnd* is unlikely to decrease.

If the buffers are large enough, almost all the segments *eventually* reach the receiver that does not send any triple duplicate acknowledgement. Therefore, the first loss evidence seldom arises.

Segments start to be dropped only after the entire buffer is full and in this case they do not reach the destination. The sender can be aware of this because the RTO expires without receiving any acknowledgement. However, the RTO is generally very large and also tends to increase with the queueing delays. Indeed, as specified by [PA00], it must be at least 1 second and it is updated by means of an exponential

[5]Including a retransmission

moving average on the basis of the RTT estimation: i.e. if the RTT increases, the RTO increases too. As a consequence, if queueing delays become larger, the RTT estimates become larger and so the RTO. Therefore, the second loss evidence seldom arises.

A side effect of bufferbloat: the unfairness

Suppose a long-lived TCP flow, characterized by a large *cwnd* and that is sending data segments into an over-buffered link. The previous subsection shows that *cwnd* will seldom decrease. If a new TCP flow has to compete with the long-lived one for the same link, it will loose: it won't have the chances to increase its *cwnd* because its segments have to traverse a buffer already saturated. This is argued by [GN11] on page 8.

1.2. Relation between the size of the buffers and the link rate

Buffers are more critical in the network devices in which fast-to-slow transitions happen: i.e. the input link is faster than the output link thus requiring to store the incoming packets before they can be serviced. Fast to slow transitions generally occur near the network edge:

- a router that connects an ethernet LAN (Local Area Network) to Internet
- a Wireless Access Point that connects a 802.11 network to Internet

But they can also occur on a router on the boundary between different Internet Service Providers

For example, an ADSL link provides a bandwidth of about 1Mb/s that is much less that the bandwidth provided by ethernet or 802.11.

The typical rule of thumb for buffer sizing is to have a bandwidth-delay product (BDP) worth of buffer, multiplying the bandwidth B of the output link (1Mb/s in the case of an ADSL modem) and a canonical value of RTT (100 ms, a continental delay for U.S and Europe). The goal is to have buffers that can host the simultaneous bits in transit between the transmitter and the receiver. Supposing that

TCP correctly estimates the bandwidth B, we have that the $\frac{cwnd}{RTT} \cdot M$ is equal to the bandwidth of the link:

$$\frac{cwnd}{RTT} \cdot M = B \tag{1.2}$$

being M the maximum TCP segment size. We already explained that a TCP sender sends "burst" of *cwnd* packet every RTT. Therefore, the bytes that can be simultaneously in transit (and that can require to be placed in the buffer) are $cwnd \cdot M$ and, thanks to the (1.2), this number can easily be found as:

$$B \cdot RTT$$

Today, this rule of thumb could be considered obsolete. [AKM04] points out that it is incorrect for the backbone links because of the large number of TCP connections multiplexed together. It shows that a link a with n flows requires a buffer not larger than $\frac{B \cdot RTT}{\sqrt{n}}$. [GN11] observes also that the 100-ms delay assumption has been weakened by the advent of CDNs (content delivery networks) and other services engineered to bring common RTTs down to 10-30 ms. All these arguments all agree on having buffers smaller than the ones calculated with the obsolete rule of thumb.

[DTMK12] shows that nowadays most home gateways have a fixed buffer size, depending simply on the vendor. Just to give an idea, buffer sizes discovered in [DTMK12] vary from 22 to 365 KB. In [CK10] was argued that the equipment had two dominant buffer sizes: 128 KB and 256 KB, without any care of the operation access rate. For a typical uplink rate of 1.2 Mbps, any buffer large than 150 KB will introduce more than one second of TCP queuing delay under load, which is prohibitively large for interactive applications. Moreover, this delay would increase to 2.3 seconds for a 512 Kbps uplink, and so on.

[GN11] highlights also that equipping the network device with the right buffer is not a simple design task. Memory is very cheap today and it's difficult to buy RAM chips small enough for the buffering in edge devices. In addition, these devices have no mechanisms for self-limitation. Commodity network devices now span many downward-compatible generations: Ethernet has gone from 10 Mbps to 10 Gbps; wireless operates from 1 Mbps to 100 or more Mbps. The result is a single buffer statically sized for larger bandwidths but much too large for lower- bandwidth links. For example, the 256 packets of buffering found in many of today's 802.11 device

drivers alone translates to more than three seconds at 1 Mbps, which is all the bandwidth you may have on some wireless networks.

Usually, when people experience high latency on Internet, they tend to think of it as a network congestion. We discovered that, on the contrary, probably they might have the culprit inside their home: a buffer on one of their network devices.

1.3. How to face bufferbloat

The previous section suggests that more insights on the correct value of the buffer size in network devices may be useful.

Another suggestion comes from the 1998 "RED manifesto" ([BCC+98]) that proposes to introduce AQM (active queue management), which attempts to keep the queues from growing too large by monitoring the growth of the packet queue and signaling the TCP sender to slow down by dropping or marking packets in a timely fashion. Nevertheless [GN11] states that AQM is not widely configured in routers, it's completely unavailable in many devices and many ISPs are running without AQM often in circumstances where they really should.

Ideally, we argue that the adoption of congestion control algorithms to replace the standard TCP one, though proposed with other goals, may mitigate, as a side effect, the bufferbloat issue. For example, the authors of FAST TCP ([JWL04]) claim that the congestion control based on the delay (rather than on the loss, like in TCP) is a more accurate congestion measure, especially in the scenarios in which losses are rare events. Simply speaking, the sender should regulate its sending rate not only when some packets are lost but also when it starts to notice high latency in the network, in order to detect as soon as possible any problem. Other delay based congestion have been proposed for many years but they will hardly be adopted: replacing TCP in all the Internet is inconceivable.

Another alternative is the introduction of application layer congestion control solutions (for example on top of UDP) with the aim to coexist with the TCP congestion control (rather than replacing it) An example is LEDBAT [SHIK10] (Low Extra Delay Background Transport), a delay based congestion control suitable for supporting background applications with the goal to operate without interfering with the performance of more delay-sensitive foreground applications. It controls the

sending rate in accordance with a queueing delay estimation that is continuously performed (more details in sec. 2.4).

1.4. Related work on the queueing delay measurement

1.4.1. Active vs passive approaches

In general, when a measurement of a phenomenon of whatever sort has to be performed, this can be accomplished with an active approach or a passive approach.

In abstract terms, an active measurement implies the modification of the phenomenon, in order to induce some desired behaviors and get useful results. In most cases, the experimenter desires the modification to be as small as possible, because he wants to preserve the hypothesis that the aspects of the phenomenon that he is measuring behave like without the "artificial" modification (or, at least, in a very similar) way.

On the contrary, a passive measurement permits to achieve the results only "observing" the phenomenon, without introducing any modification. Roughly speaking, it's more straightforward to state the hypothesis that the results are the "real-world" results, i.e. the ones that hold in the phenomenon itself, regardless of the measurement activity.

Speaking of network measurements, an active method implies that the measurement tools inject some traffic in the network in order to get the measures. On the contrary, in a passive method, the measurement tools only sniff traffic.

The methodology that we present is passive and can be applied to traces of real traffic, either recorded in the past or sniffed on line. For the reasons described above, it is unobtrusive and can provide a realistic characterization of the real-world traffic.

1.4.2. Active measurements

With few exceptions, most related work relies on active measurement.

Active measurements like [CK10], [MDF12], [HJR12], [SdDF+11], [BOS+11], [BOB12] measure latency under controlled load. Therefore they tend to give *maximum* (rather than *typical*) queueing delay.

Moreover, a parameter that influence the active measures is the rate at which the "artificial" packets are injected. Generally, this rate is not so high and the results exhibit coarse granularity, with the exception of [SdDF+11], where samples are still spaced out by 6 seconds in the best case.

Nevertheless, the most notable limitation of the active methods is that they cannot have the knowledge of the applications that the users are running and cannot correlate them to the observed queueing delay.

1.4.3. Passive measurements

Some recent work tackles the problem of passive measurement of queuing delay ([GCCK13], [All12], [CRT+13], [CR]). [CRT+13], [CR] propose methodologies to infer remote host queues exploiting transport layer information available in packet headers, for both uTP [CR] (the new protocol proposed by BitTorrent as TCP replacement for data swarming - see sec. 2.4) and TCP [CRT+13] (using RFC1323 TimeStamp option [JBB92]). The authors of [CR] implemented in Tstat their methodology and the implementation that we present here is obtained generalizing their implementation and then adapting it to the TCP case (see chapter 3 for details).

Contrarily to [CRT+13] and [CR], in this work we focus on the local host queue (since we have full knowledge of all traffic on that host) and adopt a more general methodology, of which we outline some important differences. First, notice that while uTP timestamps allow to precisely gauge the remote queue (even in presence of cross-traffic toward unseen hosts) observations are limited in both space (to hosts that are running BitTorrent) and time (precisely when they run it). This constrains measurement campaign [CR] on the one hand, and possibly induces a biased view of the Internet bufferbloat on the other hand (since BitTorrent is a data-intensive application) – problems that this work instead avoids. Second, contrarily to [CRT+13], we avoid relying on timestamps carried in packet headers for TCP, increasing the reach of the methodology (despite growth of TCP TimeStamp option usage, this still accounts for modest 5%-30% at our vantage points).

Closer to our work is [All12] that uses Bro scripts ([Pax99]) as a basis for the measurements. Bro is an open source, Unix-based Network Intrusion Detection System that passively monitors network traffic and looks for suspicious activity. [All12] employs a similar methodology to ours, relying on TCP data/ acknowledgement pairs, using trace timestamps as opposite to TCP Timestamp option and takes care of rejecting RTT samples from retransmitted segments. But this methodology is not validated in a testbed and does not take care of tweaking the measurement parameters that can influence the measures (see sec. 3.3.2 for potential issues). We also point out that [All12] provides measures of Internet traffic of an experimental high speed network (more details in sec. 3.1) and cannot be representative of what a "normal" user experiences.

Finally, [GCCK13] focuses on a memory-efficient bufferbloat measurement methodology, by keeping approximate TCP state in a probabilistic data structure (that can fit the cache of current MIPS and ARM processors used in home DSL gateways), at the price of a minimal accuracy loss (error is less than 10 ms in 99% of the cases, compared to tcptrace as a baseline). However, as the focus of [GCCK13] is on the relative accuracy of the methodology, it reports differences with respect to the baseline rather than absolute bufferbloat measurement. Our approach is instead complementary and, assuming a high performance dedicated measurement box (i.e., no memory constraint), implements a methodology to accurately gauge current Internet bufferbloat (incidentally, building over tcptrace, of which Tstat is an evolution).

We point out that, to the best of our knowledge, this work is the first to report a detailed per-application view of the Internet queuing delay.

1.5. Is bufferbloat a real concern?

It is known that queuing delays can potentially reach a few seconds under load stress: [CK10] purposely fills the pipe and learns the maximum queueing delay. These delays have been also anecdotically observed by [GN11]. Some authors (like [Jac98] in 1998) advocate that countermeasures to the bufferbloat are necessary for the health of the Internet. However it is unclear how common they are for end-user daily experience – which is precisely what the methodology that we propose attempts to measure.

Section 4.2.3 of [DHGS07] (2007) studies the queueing delay of DSL users both in the downstream and in the upstream direction. It reveals very large queues in the upstream direction that negatively affect interactive traffic. [CK10] measures severe overbuffering. [GN11] claims that bufferbloat is a serious problem which has immediately to be faced.

On the other hand, [All12] points out that the magnitude of the phenomenon seems to be quite modest. The results that it presents show that queueing does happen within the network, as expected, but the author argues that whether this constitutes "bloat" is a subjective judgment. Nevertheless, it's worth remarking that the measurement of [All12] is performed on an experimental high speed network and cannot be representative of what a "normal" user experiences.

The work cited in this section shows that authors often provide contradictory results on the extent of bufferbloat. Reasonable causes of these may be:

- different input data (different networks under observation, different periods of observation)
- different ways to produce data and statistically characterize the queueing delay

This scenario asks for more investigation on the analysis of the queueing delay in the Internet.

2. Measuring the queueing delay: methodology

2.1. Assumptions and definitions

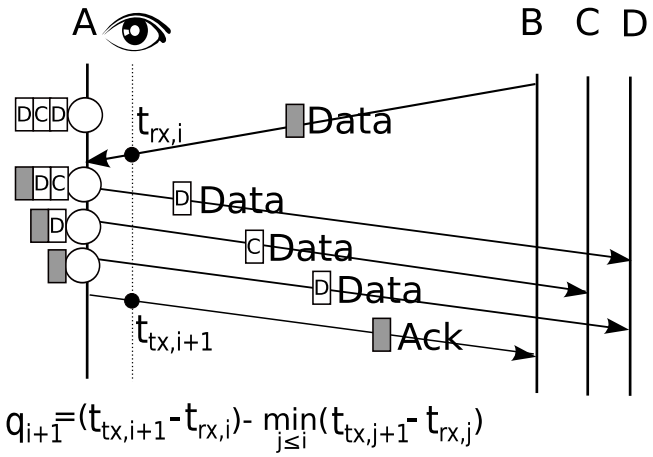


Figure 2.1.: Queueing delay estimation

We infer queuing delay of local hosts simply as depicted in Fig. 2.1, where the queueing delay on host A is calculated and the monitoring device is supposed to be near A and is represented by the eye. In this section, we will add some simplifying assumptions. We will discuss these assumptions later.

With respect to Fig. 2.1, assume that the monitor observes a data segment `DS` directed to A at time $t_{rx,i}$. Suppose that when the data segment arrives at A, its queue contains packets directed toward hosts B, C and D. Then, the TCP receiver issues the corresponding acknowledgement `ACK` and the monitor sees it at instant $t_{tx,i+1}$.

Assumption 1. *Here we assume that the TCP receiver issues the acknowledgement as soon as data is received and also that*

- *`ACK` is not a delayed acknowledgement*
- *there is no reordering of the data segments (if the n-th byte of the data stream was the last received one, `DS` starts with the $n+1$-th byte)*
- *the data segment is not a retransmitted one, otherwise the monitor could not distinguish if `ACK` was related to some previous transmission of the data segment or to the present one*

`ACK` will be serviced after the already queued packets. We consider the queueing delay at A as the time that `ACK` spends in the queue waiting for the previous packets to exit from A (in this case, packets toward B, C and D) before it can exit too. Suppose that monitor sniffs the acknowledgement at time $t_{tx,i+1}$. The two instants are correlated by the following equation:

$$t_{rx,i} + T_{pr,i+1} + T_{el,i+1} + q_{i+1} + T'_{pr,i+1} = t_{tx,i+1}$$

where $T_{pr,i+1}$ is the propagation time that the data segment spends to go from the monitor to A, $T_{el,i+1}$ is the elaboration time, i.e. the time that the TCP receiver requires to produce the acknowledgment, q_{i+1} is the queueing delay and $T_{pr,i+1}$ is the propagation delay that the acknowledgment spends to go from A to the monitor. Defining

$$\Delta T_{i+1} \triangleq t_{tx,i+1} - t_{rx,i}$$

we have:

$$q_{i+1} = \Delta T_{i+1} - \left(T_{pr,i+1} + T_{el,i+1} + T'_{pr,i+1}\right)$$

In order to calculate q_{i+1}, we have to find the term $T_{pr,i+1} + T_{el,i+1} + T'_{pr,i+1}$.

Assumption 2. *The variation of the elaboration time is negligible, so that we can introduce a number T_{el} such that, for every j:*

$$T_{el,j} \simeq T_{el}$$

Assumption 3. *The variation of the propagation delays is negligible too, so that*

we can introduce a number T_{pr} such that, for every j:

$$T_{pr,j} + T'_{pr,j} \simeq T_{pr}$$

Thanks to these assumption, the following equation holds for every j:

$$q_j \simeq \Delta T_j - (T_{el} + T_{pr}) \tag{2.1}$$

What we miss now is to calculate $T_{el} + T_{pr}$. We call it *baseline* and indicate it with β. We need the following assumption

Assumption 4. *At least one acknowledgement has experienced no queueing delay, i.e. there exists a $k \leq i + 1$ such that $q_k = 0$. As a consequence, we have*

$$\Delta T_k = T_{pr,k} + T_{el,k} + T'_{pr,k}$$

and, using assumptions 2 and 3:

$$\Delta T_k \simeq T_{pr} + T_{el} = \beta$$

Taking a $j \leq i + 1$, we observe that

$$\Delta T_j \simeq q_j + \beta \geq \beta$$

Therefore we can state that the baseline is

$$\beta \simeq \Delta T_k = \min_{j \leq i+1} \Delta T_j$$

2.2. Validity conditions

First of all, we need to guarantee that the assumption 1 holds. Therefore, each acknowledgment sent by A is classified as valid or invalid and only the former ones are used for our measurement. The invalid acknowledgments are the ones related to reordering or retransmission. The valid/invalid classification is made by our

implementation of Tstat[1] in accordance with the algorithm described in [MMM06] and with the method that Tcptrace uses to filter out invalid round trip times samples.

Data segment retransmission

As observed in [MMM06] (section 2), retransmissions may stem from manifacturing apparatuses, routing loops, mis-configured networks, or retransmissions at the link layer (e.g., when a MAC Layer acknowledgement is lost in a Wireless LAN forcing the sender to retransmit the frame), retransmission by a TCP sender merely for flow control purposes or because a Retransmission Timer has fired or the fast retransmit mechanism has been triggered (i.e., three or more duplicate acknowledgements have been received for the segment before the retransmitted one) or the previous TCP acknowledge for that data segment hes been lost. Section 3 of [KP87] examines the *retransmission ambiguity*: it cannot be possible to know if an acknowledgement is in response to the original transmission of a packet or a retransmission. Considering an acknowledgement related to a segment that was transmitted twice, if we suppose that it is in response to the original transmission, we obtain a certain value of ΔT_{i+1}, while if we suppose that it is in response to the retransmission, we get $\Delta T'_{i+1} < \Delta T_{i+1}$. Our choice is to ignore acknowledgements referred to retransmitted data segment ([All12] (section 2.1) operates in the same way). To neglect the acknowledgement of retransmitted data segments, Tstat (that runs on the monitor) maintains a database of descriptors, one for each sniffed data segment, indicating the sequence numbers of the first and the last byte[2]. When a new data segment arrives, Tstat can check if the bytes that it carries were previously sniffed[3]. If it contains some byte that was already sniffed in the past (considering the fact that also partial retransmit can be performed), the data segment is labeled as retransmitted [4]. When Tstat sniffs an acknowledgement, it searches [5] its database for the descriptor of the data segment

[1]This classification is performed by the function `ack_in(..)` called by `tcp_flow_stat(..)` (`tstat/tcp.c`). The implementation of `ack_in(..)` is in `tstat/rexmit.c` and the return value is an enum value that can assume one of the following values: `NORMAL`, `AMBIG`, `NOSAMP`, `CUMUL`, `TRIPLE`. We perform queueing delay measures only when the return value is `NORMAL`. This is the only case when the acknowledgement can be considered valid.
[2]see the function `addseg(..)` in `tstat/rexmit.c`
[3]This is done by the function `retransmit(..)` in `tstat/rexmit.c`
[4]See the field `retrans` of `struct segment` (in `tstat/struct.h`)
[5]See `ack_in(..)` in `tstat/rexmit.c`

that is being acknowledged[6] and checks if it's retransmitted [7]. If not, the queueing delay calculation is performed [8], otherwise the acknowledgement is invalid and is ignored.

Reordering

Out of sequence segments can occur as a consequence of a retransmission of lost segments or of network reordering [9]. For each acknowledgement, Tstat verifies that the acknowledged data segment is the last sniffed one [10]. If it is not the case, the acknowledgement is invalid.

Duplicate acks

In addition, for each acknowledgement, Tstat checks if the relative data segment has already been acknowledged in the past. If it is the case, the acknowledgment is invalid [11].

Delayed acknowledgements

The TCP receiver may generate a delayed acknowledgement. In this case, it does not generate the acknowledgement as soon as a data segment is received but delays the acknowledgement for a certain amount of time. The goal is to wait that TCP receiver have data ready to send back in order to piggyback the acknowledgement with those data. The aim is to increase the efficiency. Section 4.2 of [APS+99] imposes that an acknowledgement must be generated within 500ms. If the $i+1$-th acknowledgement is a delayed one, the sample q_{i+1} is overestimated, because the time $\Delta T_{i+1} - \beta$ contains not only the queueing delay but also the time in which the TCP receiver has deliberately been idle. This is a potential issue of the methodology that we do not address and that requires further investigation.

[6] It is the segment such that the sequence number of its last byte is the acknowledgement number -1

[7] This check is performed in `rtt_ackin(..)` in `tstat/rexmit.c`

[8] See the line of `rtt_ackin(..)` of `tstat/rexmit.c` where `bufferbloat_analysis(..)` is called

[9] Section 2 of [MMM06] proposes a technique to distinguish between the two cases

[10] See the local variable `intervening_xmits` in `ackin(..)` in `tstat/rexmit.c` . If this variable is true, the data segment is not the last sniffed one and the acknowledgement is classified as `NOSAMP`.

[11] In particular, it is marked as CUMUL or TRIPLE (see `ack_in(..)` in `tstat/rexmit.c`)

2.3. Computing the baseline delay

We can suppose that assumption 2 holds because usually the elaboration time is negligible and its variation even more so. Regarding the assumption 3, propagation delay can vary as a consequence of a route change. This is not a great concern in our case because in our experimental setup (sec. 3.1) we place the monitor at the network edge. Therefore, considering a TCP flow, all packets tend to follow the same path going from the vantage point to the hosts and then from the hosts to the vantage point again. The same argument is given by [All12] (in section 2.1). Therefore, we can suppose that assumptions 2 and 3 hold.

The assumption that deserves more attention is the 4. It is too strong for the real cases. It may happen that, considering a flow of a certain host, from its starting instant to the instant $t_{tx,i+1}$, it has always seen a non-empty buffer. This is especially true when $i+1$ is small. Therefore, we cannot calculate:

$$\beta \simeq T_{pr} + T_{el}$$

All that we can do is calculating β_{i+1}, an estimation of the baseline at the instant $t_{tx,i+1}$, as:

$$\beta_{i+1} \triangleq \min_{j \leq i+1} \Delta T_j$$

Observe that

$$\beta_{i+1} \simeq \min_{j \leq i+1} \bar{q}_j + T_{el} + T_{pr} \geq \beta$$

where $\bar{q}_j$ is the real queueing delay[12] of the acknowledgement sniffed at the instant t_j. This demonstrates that β_{i+1} overestimates the real baseline β.

The monitor can then estimate the queuing delay q_{i+1} incurred by the $(i+1)$-th acknowledgement as

$$q_{i+1} = \Delta T_{i+1} - \beta_{i+1}$$

i.e. the difference between the current sample ΔT_{i+1} and the minimum among the

[12]This value is "hidden" inside $\min_{j \leq i+1} \Delta T_j$ and cannot be separated from T_{el} and T_{pr}

previously observed samples of that flow.

It should be clear, from the formula above, that q_{i+1} is an overestimation of the real queueing delay $\bar{q}_{i+1}$:

$$q_{i+1} \geq \bar{q}_{i+1}$$

Time evolution of the baseline

Notice that the sequence of the baseline estimates$\beta_1, \beta_2, \ldots$ is monotonically non decreasing and that, going ahead, the baseline estimation becomes more and more refined and has more chance to approach the real β. As a consequence, considering a single TCP flow, the last queueing delay estimates may be more accurate than the first ones.

This observation suggests an alternative way to get the baseline estimate at time $t_{tx,i+1}$: calculating the minimum not only among the ΔT_j samples seen until $t_{tx,i+1}$ but among all the ΔT_j samples, until the end of the flow:

$$\beta_{i+1} \triangleq \min_{\forall j} \Delta T_j$$

This solution is feasible only for offline processing in which we know in advance all the TCP flow, but we want to develop a methodology suitable for both offline and online processing, so we do not implement this enhanced calculation. As future work, an analysis of the impact of the baseline calculation in the accuracy of queueing delay estimation may be conducted.

2.4. Similarity with LEDBAT

Low Extra Delay Background Transport (LEDBAT) ([SHIK10]) is an experimental delay-based congestion control algorithm designed for background bulk-transfer applications. It is the congestion control algorithm used by Micro Transport Protocol (uTP), a BitTorrent peer-to-peer file sharing protocol. A LEDBAT sender implement a Proportional Integral Derivative (PID) controller to adjust the congestion window taking the queueing delay estimate as the input (see section 1.2.2 of [Chi12]).

There are similarities between our queueing delay estimation and LEDBAT, but in this section we justify the differences between the two methods.

LEDBAT requires that each data segment carries a "timestamp"[13] from the sender, based on which the receiver computes the one-way delay from the sender and sends this computed value back to the sender(see section 2.2 of [SHIK10]). The LEDBAT sender keeps track of all these one way delays and use them to calculate a baseline β_i and the queueing delay estimate q_i:

$$q_i = owd_i - \beta_i$$

where owd_i is a value obtained applying a filter function to the recent one way delays with the aim of eliminating the outliers (for the sake of simplicity we can think of owd_i as the i-th one way delay). The calculation of β_i is very complex compared with ours. LEDBAT maintains a base history vector. The first element of this vector is the minimum of the owd_i seen in the current minute. The second element is the minimum calculated in the previous minute and so on. Every minute the base history vector rolls over: the last element is removed, all the other elements are shifted back and the first element can be used to update the minimum of the owd over the minute just begun. The goal of this procedure is to address the possible changes in the network state or in the path followed by the packets. It is why the baseline is not calculated as the minimum owd_i over the entire communication, but only over the recent interval (the interval covered by the base history vector). As explained in sec. 2.3, we can neglect changes in network state or in the path and can perform a simpler calculation of the baseline.

The main difference between LEDBAT queueing delay estimation and ours is the goal: our aim is to measure the queueing delay itself, while for LEDBAT q_i is only the input for the PID controller and a precise estimation of it is not the main goal for a PID controller can tolerate also imprecise inputs.

2.5. Queueing delay aggregation

We batch consecutive samples belonging to the same TCP flow into windows whose duration relates with the timescale typical of user dynamics. We consider a TCP

[13]The application (and not the transport layer) has to take care of this timestamp

flow as the communication between two TCP endpoints in a certain direction, i.e. if the TCP connection between A and B exchange data in the two directions, each of them will be considered as a separate flow. An endpoint is identified by a (IP address, TCP port) pair.

For each flow, we calculate the queueing delay samples as explained in this chapter. In every window of 1 second, we calculate the aggregated queueing delay as the average of the collected queueing delay samples during that time window. In case there are no queueing delay samples in a window, no aggregated queueing delay is calculated (or, in other words, the aggregated queueing delay is an `NA`).

The motivation behind the choice to aggregate the queueing delay is twofold.

First, we argue that the aggregated queueing delay is useful from the user perspective for it describes what happens to the "entire" flow at a certain time and not to its segments. In the specific case of uTP, [CR] shows that queuing delay statistics can be biased (precisely queueing delay is underestimated) in case each packet is counted as a sample (as opposite to windows of equal duration). In our case, a similar issue arises. As Section sec. 3.3 will show, in conjunction with high queueing delays, the number of acknowledgements that are observed generally decreases. Therefore, we tend to have more samples representing the intervals of low queueing delays than the ones representing the intervals of high queueing delays. As stated in [CR] we risk to couple the measurement process with the flow dynamics and if we do not take into account this detail, global statistics as the cumulative distribution function or the probability density function of the queueing delay samples tend to be "shifted" a bit towards lower values.

Second, aggregating the queueing delays permits to store less information to accomplish our measurements (see sec. 3.4 for more details). This is a crucial concern if our methodology has to be applied to very large traffic traces or if it has to run in an online monitoring system continuously sniffing large slices of Internet traffic.

3. Measuring queueing delay: implementation

3.1. Measurement scenario

We now describe the experimental setup used to obtain the trace on which we applied our methodology. We claim that our methodology can be applied also to other scenarios but, in this case, care of the accuracy of the results should be taken, e.g. the assumption 3 of sec. 2.1 might be less straightforward.

Our monitor is placed near the DSLAM (Digital Subscriber Line Access Multiplexer) of an Internet Service Provider, as in Fig. 3.1. Our goal is to measure the queueing delay inside user devices by inferring how much time the acknowledgments coming from them spend enqueued in the buffers. We claim that the queueing delay inside user devices is sufficiently representative for the queueing delay in the Internet, considering that (i) buffers are easily filled in the bottlenecks (i.e. the devices in which fast-to-slow transitions happen) as stated in sec. 1.2 and that (ii) the broadband link is the bottleneck on the Internet path (see section 3.4.2 of [DHGS07]).

Packets are timestamped at the measurement point via Endace DAG cards, so that timestamp is reliable. A DAG card is a hardware that could capture traffic at very high line rates with a timestamp resolution of less than 1 microsecond.

Our experimental setup is similar to the one used by [All12]: the main difference is that [All12] vantage point is placed between the CCZ users and the Internet. CCZ (Case Connection Zone) in an experimental fiber-to-the-home network which connects roughly 90 homes adjacent to Case Western Reserve University's campus with bi-directional 1 Gbps links. On the contrary we are able to monitor the traffic of "normal" users.

We consider each internal IP as a single host. This is known to be simplistic as, due

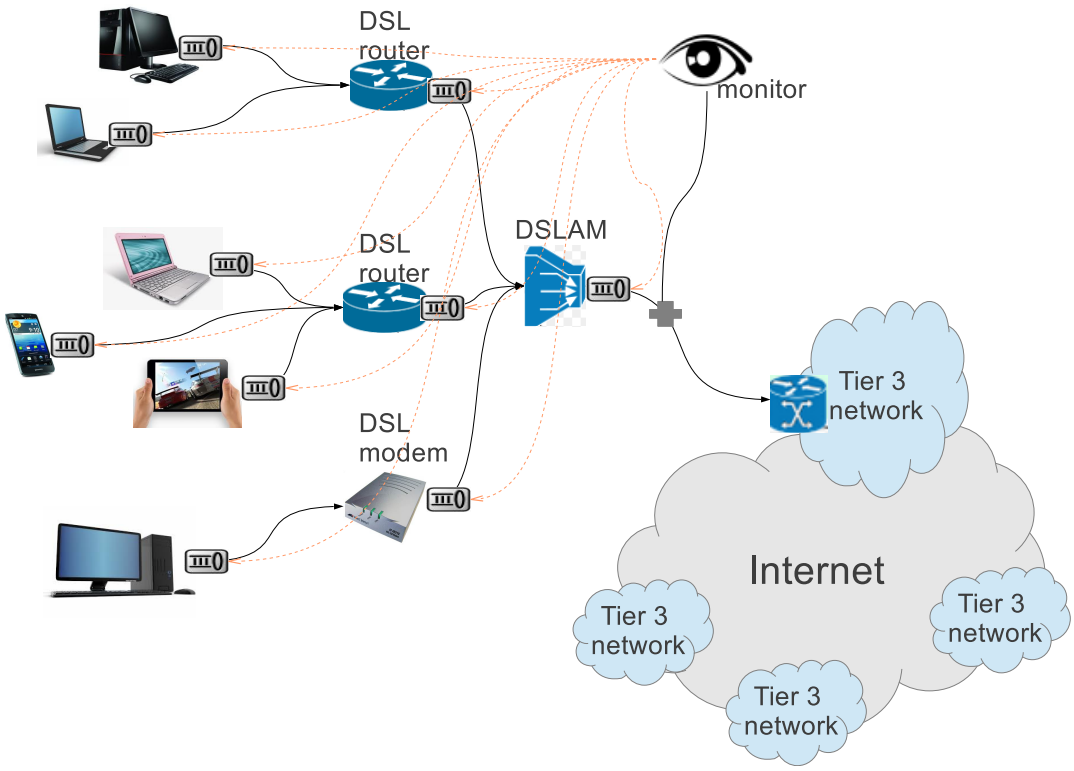


Figure 3.1.: Vantage point

to the penetration of NAT devices, the same IP is shared by multiple hosts (50% of the cases [MSF11]), that are possibly active at the same time. Yet we point out that this simplification has no impact for our methodology, since these potentially multiple hosts share the same access bottleneck link (see chapter 1).

3.2. A look at the implementation

3.2.1. Tstat overview

Tstat [FMM+11] is an open source passive monitoring tool developed by the networking research group at Politecnico di Torino[1] since 2000. It started as an evolution of tcptrace[2] and offers live monitoring (specifying the interface to monitor) and offline analysis (specifying the input file to analyze, e.g. a pcap trace). It is written in ANSI C for efficiency and allows sophisticated multi-gigabit-per-second traffic analysis to be run live using common hardware and Libpcap [JLM94], the de facto standard application programming interface (API) to capture packets.

Tstat sniffs IP packets and aggregate them in *flows*. Each flow is typically defined as the sequence of packets characterized by the same flowID that have been observed in a given time interval, where flowID = (ipaddress1, port1, ipaddress2, port2, direction) so that TCP and UDP flows are considered. The direction indicates if the data go from ipaddress1 to ipaddres2 or in the opposite direction. For this work, only TCP flows are studied. The start of a new flow is commonly identified when the TCP three-way handshake is observed; similarly, its end is triggered when either a proper TCP connection teardown is seen, or no packets have been observed for some time.

By configuration files, it is possible to specify the hosts that are considered as internal or external and the traffic is organized in

- Incoming traffic: The source is external and the destination is internal.
- Outgoing traffic: The source is internal and the destination is external.
- Local traffic: Both source and destination are internal.
- External traffic: Both source and destination are external.

[1] Tstat Homepage: http://tstat.tlc.polito.it
[2] TCPTrace Homepage: http://www.tcptrace.org

We monitor only outgoing traffic and divide internal and external hosts according to Fig. 3.1.

Tstat collects several network-layer as well as transport-layer measurements, which are described in full details in http://tstat.polito.it/measure.shtml. Part of the Tstat output is represented by plain text logs. In the offline analysis (the one that we are performing), for each pcap trace Tstat produces a folder with different log files. Examples of log files are: `log_tcp_complete`, `log_tcp_nocomplete` , `log_udp_complete` , `log_skype_complete`, `log_chat_complete`, `log_streaming_complete`. Their form is quite similar: they are plain text files where each row corresponds to a different flow and each column is associated to a specific measure (for `log_tcp_complete` we have number of data packets, number of SYN messages, maximum receiver window announced, ...).

Tstat is also able to identify the application that generated the traffic. Primarily, Tstat implements a deep packet inspection (DPI) technology.

3.2.2. Queueing delay measurement implementation

Bufferbloat-related code

When implementing our methodology in Tstat, our goal was to isolate the code related to the queueing delay calculation from the rest of Tstat. Tstat is now a complex tool able to accomplish different measurement tasks and it's important to maintain it modular: the experimenter should be able to activate and deactivate what he needs for his measurement. We know that not all the experimenters will be interested in queueing delay measures and we want that they should be able to use Tstat for their goals simply ignoring the existence of portions of code related to queueing delay.

All the C procedures related to queueing delay measurement are in `tstat/bufferbloat.h` and `tstat/bufferbloat.c`. In particular, the functions declared here are:

- `bufferbloat_analysis(..)`:
  - called in `tstat/tcp.c`, `tstat/ledbat.c`, `tstat/rexmit.c`
- `chance_is_not_valid(..)`:
  - called in `tstat/tcp.c`

- `check_direction_consistency(..)`:
  - called in `tstat/tcp.c`, `tstat/ledbat.c`, `tstat/rexmit.c`

These function calls are activated only if the precompilation option `BUFFERBLOAT_ANALYSIS` is enabled (see "Configuration parameters").

The code in `tstat/bufferbloat.c` is an evolution of the code developed by Chiara Chirichella [Chi12] to study the bufferbloat by monitoring the uTP flows (see sec. 2.4). We generalize and extend the code to make it work both with uTP monitoring (as in Chirichella's work) and TCP monitoring (as in our work). As we observed in sec. 2.4, the calculation method used for the uTP case is very similar to ours. They both estimate the queueing delay collecting the samples of *gross delays*. Gross delay is the general term with which we indicate the one-way delay in the uTP case and the data-to-acknowledgement time in our TCP case. Therefore, if `bufferbloat_analysis(..)` is called by `tstat/rexmit.c`, the following parameter is passed:

```
last_gross_delay = etime_rtt/1000
```

where `etime_rtt` is the data-to-acknowledgement time in microseconds.

On the contrary, if `bufferbloat_analysis(..)` is called by `tstat/ledbat.c`, the `last_gross_delay` will be assigned a value depending by `time_diff`, that is the one-way delay.

We inserted comments in the code with the aim to make it self-explanatory and thus we do not dwell on the code description.

Configuration parameters

Our implementation on Tstat is highly configurable by precompilation options[3].

The general options are

- `BUFFERBLOAT_ANALYSIS`: if enabled, the queueing delay estimation will be performed. Otherwise, the "classical" version of Tstat will run
- `DATA_TRIGGERED_BUFFERBLOAT_ANALYSIS`: if enabled, in addition to the queueing delay inference method that we are presenting in this work, a different method will run. It is based on the ack-to-the-following-data-segment time (rather then on data-to-ack time).We implement this method but do not analyze it. This method will be ignored in this work.

[3]See `tstat/Makefile.conf`

- `FILTERING`: if enabled, all the LEDBAT-like filtering operations to calculate the baseline will be used (see sec. 2.4).
- `FORCE_CALL_INLINING`: if enabled, gcc compiler will be forced to compile using the optimization technique called "call inlining"[4]: in the compilation phase, the calls to a function are replaced with the body of the function. This generally permits shorter running times, because, at run time, the overhead of the function call (branch, parameter passing, allocation of space in the stack, return parameter handling, ...) is avoided. The disadvantage is the increase of the object code size.

The debug options are:

- `SEVERE_DEBUG`: If it is enabled, a lot of redundant and overabundant checks will be performed to check for inconsistent states or data. If an inconsistent state is detected, Tstat terminates and issues an error message. This can be useful for further editing of the code to be sure that the modifications do not introduce inconsistency.
- `SAMPLE_VALIDITY`: If it is enabled, Tstat will perform the calculations to check how many acknowledgements are considered valid or invalid (see sec. 2.2).
- `SAMPLE_BY_SAMPLE_LOG`: If it is enabled, the log files with all the queueing delay samples will be produced (in addition to the log files of the aggregated queueing delays).
- `ONE_FLOW_ONLY`: it can be used when, for debugging purposes, it is necessary to be sure that Tstat is monitoring only one flow.

The default configuration is with only `BUFFERBLOAT_ANALYSIS` and `FORCE_CALL_INLINING` are enabled.

3.2.3. Output log file

In addition to the other Tstat log files, the following files will be produced[5]:

- `log_tcp_windowed_qd_acktrig`: if `BUFFERBLOAT_ANALYSIS` is enabled
- `log_ledbat_windowed_qd`: if `BUFFERBLOAT_ANALYSIS` is enabled

[4] See http://www.greenend.org.uk/rjk/tech/inline.html for a good howto on the question
[5] Notice that the term "windowed" stands for "aggregated"

- `log_tcp_windowed_qd_datatrig`: if `BUFFERBLOAT_ANALYSIS` and `DATA_TRIGGERED` are enabled
- `log_ledbat_qd_sample`: if `BUFFERBLOAT_ANALYSIS` and `SAMPLE_BY_SAMPLE_LOG` are enabled
- `log_tcp_qd_sample_acktrig`: if `BUFFERBLOAT_ANALYSIS` and `SAMPLE_BY_SAMPLE_LOG` are enabled
- `log_tcp_qd_sample_datatrig`: if `BUFFERBLOAT_ANALYSIS`, `SAMPLE_BY_SAMPLE_LOG` and `DATA_TRIGGERED` are enabled

For the measures that we present here, only `log_tcp_windowed_qd_acktrig` is used.

Each row of this file describes a 1-second-window of a certain flow pair characterized by the same (ipaddress1, port1, ipaddress2, port2)-tuple. Considering the role of the host that sends the packet, for each tuple like above Tstat distinguishes between client and server, i.e., host that opens a connection and and host that replies to connection request. The role of client or server does not correspond to the classification in internal or external host.

The columns of the log file are as follows:

1. `timestamp` (in seconds)
2. `ip_addr_1`: IP address of the client
3. `port_1`: port of the client
4. `ip_addr_2`: port of the server
5. `port_2`: port of the server
6. `internal_src`: 1 if the host identified as client is internal; 0 otherwise[6]
7. `internal_dst`: 1 if the host identified as server is internal; 0 otherwise
8. `connection_type` (client-to-server direction[7]): a string in the form `<con_type>:<p2p_type>`. `<con_type>` can be P2P, HTTP, SMTP, ... (see `tstat/protocol.h` for the complete list). If the connection is of type P2P, `<p2p_type>` indicates more detail about the type of connection. In our work, we use only the information in `<con_type>`.

[6]For more details, search `tstat/struct.h` for "internal_src"

[7]Actually, this is a redundant information beacause the two directions are always of the same type.

9. `aggregated_queueing_delay` (client-to-server direction): in milliseconds
10. `window_error` (client-to-server direction): in milliseconds
11. `qd_max_w1` (client-to-server direction): in milliseconds, it indicates the maximum of the queueing delays collected in the last 1-second-window
12. `chances_in_win` (client-to-server direction): how many acknowledgements (either valid or invalid) are collected in the 1-second-window. Each acknowledgment is considered as a "chance"[8] to calculate the queueing delay. When `SAMPLE_VALIDITY` is disabled, this column is always equal to "`-`".
13. `aggregated_grossdelay` (client-to-server direction): this is the average data-to-acknowledgement time [9] (milliseconds) of all the samples collected in the 1-second-window.
14. `connection_id` (client-to-server direction): this is meaningful only in the LEDBAT contest.
15. `samples_in_win` (client-to-server direction): number of queueing delay samples collected in the 1-second window. The aggregated queueing delay is calculated on the base of these samples.
16. `not_void_windows` (client-to-server direction): meaningless, used only for debugging purposes
17. `qd_measured_sum` (client-to-server direction): in milliseconds, it is the sum of the queueing delay samples seen from the beginning of the flow
18. `aggregated_qd_sum` (client-to-server direction): in milliseconds, it is the sum of the aggregated queueing delay of the 1-second-windows seen from the beginning of the flow.
19. `sample_qd_sum_until_last_window` (client-to-server direction): in milliseconds, it is the sum of all the queueing delay samples from the beginning of the flow to the last closed 1-second window.
20. `baseline` (client-to-server direction): in milliseconds: it is the last baseline calculated in the 1-second window. Notice that this value is not used for the

[8] The log file `log_tcp_windowed_qd_datatrig` (that we do not use) has the same form presented here. In that contest, every data segment is considered as a chance to calculate the queueing delay. This is why we do not call this column "number_of_acks" and prefer the more general name "chance"

[9] ΔT in sec. 2.1

calculation but only written here for debugging purposes. Indeed, the baseline used for the queueing delay calculation is recomputed at every sample.

21. `connection_type` (server-to-client direction)
22. `aggregated_queueing_delay` (server-to-client direction)
23. `window_error` (server-to-client direction)
24. `qd_max_w1` (server-to-client direction)
25. `chances_in_win` (server-to-client direction)
26. `aggregated_grossdelay` (server-to-client direction)
27. `connection_id` (server-to-client direction)
28. `samples_in_win` (server-to-client direction)
29. `not_void_windows` (server-to-client direction)
30. `qd_measured_sum` (server-to-client direction)
31. `aggregated_qd_sum` (server-to-client direction)
32. `sample_qd_sum_until_last_window` (server-to-client direction)
33. `baseline` (server-to-client direction)

The procedures responsible for writing the log file are `print_last_window_general(..)` (it writes the first five columns of each row) and `print_last_window_directional(..)` (it writes all the other columns, starting from the ones related to the client-to-server direction and then writing the columns related to the server-to-client direction). Both the procedures are in `tstat/bufferbloat.h` and `tstat/bufferbloat.c`. They are used to produce not only `log_tcp_windowed_qd_acktrig`, but also `log_ledbat_windowed_qd` and `log_tcp_windowed_qd_datatrig`. This is why, for the sake of generality, some columns are meaningless in our contest and some column names may seem too abstract.

3.2.4. Post-processing

All the results are obtained by means of calculations that have the log file described above as input. The script files that implement these calculations are in the folders `offline_analysis`, `performance_evaluation` and `ping_validation`.

Most of them are Linux bash scripts and use tools like awk[10]. We use Gnuplot ([Jan09]) for some of the plots.

For more advanced data analysis, we use R ([IG96]), a programming language for statistical computing and graphics. We choose to use this language because

- it permits to associate to raw data semantic information that are essential for further complex processing
- it provides ready-to-use statistical functions and plotting capabilities
- it permits to combine data in a very natural way, similar to SQL
- it is suitable for mathematical computation, like Matlab

The major limitation that we find in R is that it reads data into memory by default. The data needed for some of our computations are huge and cannot be contained in the RAM only. The result is that, by default, R run into "cannot allocate memory" problems. Fortunately, R has few packages for big data support[11]. We use *ff* package, that provides file-based access to datasets that cannot fit in memory: the programmer invokes the same functions[12] as if the data set were entirely in the memory and the ff package performs the underlining operations to map from disk to memory only necessary/active parts of the data.

All the functions we use are in
`offline_analysis/R_scripts/get_affected_flows.r` . The best way to use these functions is editing the file above in its last part and inserting the function calls that are needed; then, launching

```
sh offline_analysis/Rlauncher.sh
```

from the shell. This script will execute the R code: all the function definitions will be loaded and the function calls executed. A log file will be written in the file specified inside `get_affected_flows.r` to check for possible errors.

[10] In a Linux shell, run `man awk` for more details

[11] See
http://www.bytemining.com/2010/08/taking-r-to-the-limit-part-ii-large-datasets-in-r/
for a good introduction to this topic

[12] Actually, we found that this is only partially true

3.3. Validation of measurement accuracy

In a local testbed, we compare our measures to the ones obtained by considering the ping RTT, as ground truth. We also change the data rate at which the network interface works in order to understand in what scenario our measurement is more accurate. We repeat the experiment with different values of a parameter (the quadrant size) to evaluate its impact on the accuracy.

3.3.1. Testbed description

We send bidirectional TCP traffic between two hosts connected via an Ethernet LAN, as in Fig. 3.2. We perform the measurement of the queueing delay of the queue on host A. We start three flows:

- the *monitored TCP flow*: host B sends data segments to host A that replies with the acknowledgements.
- the *cross traffic flow*: host A sends burst of data to B
- the *ICMP flow*: host B sends an ICMP echo requests to host A at 1 Hz that replies with an ICMP echo replies.

Our monitor runs on host B and estimate the queueing delay on A observing the data-to-ack time on the monitored flow, applying the methodology of chapter 2. In particular, the monitor infers how much time the acknowledgments (that host A sends on the monitored flow) pass in the local queue. The cross traffic flow is used to periodically form the queue inside A, filling it with bursts of data segments with which the host A acknowledgements in the monitored flow have to compete. Compare Fig. 2.1 with Fig. 3.2 to have a better idea of how the methodology is applied in the testbed.

Crafting traffic

The monitored and cross traffic flow are artificially created with *Iperf* ([TQD+05]). Iperf is a tool that consists of two softwares: iperf client and iperf server.

To create the monitored flow, we first launch iperf server on host A listening on port 5011. Then we launch the iperf client on host B specifying to open a TCP

connection to the port 5011 on host A. In this way, iperf client starts to send data segment to iperf server that replies with acknowledgements.

To create the cross traffic flow we launch iperf server on host B listening on port 5012, then we launch iperf client on host A a series of times, separated by 15 seconds, for 2 seconds each time. iperf client acts to inject traffic at a data rate such to use the entire available bandwidth. Therefore, during the intervals of 2 seconds in which iperf client is active, we recreate the scenario of full link utilization and we impose a large queue in the exit link of host A, hampering in this way the acknowledgments of host A on the monitored flow.

The ICMP flow is created launching the Linux utility *ping* on host B, setting host A as destination.

Creating the bottlenecks

To create some artificial bottlenecks, we use Ethtool[13] and Netem[14].

Ethtool permits to physically slow down the Ethernet interface[15]. Our hosts are equipped with a 100 Mbps Ethernet interface and we slow down the interface of host A to 10 Mbps.

On the other hand, Netem creates software bottlenecks. We set Netem to use the Hierarchical Token Bucket (HTB) method to implement the bottleneck. HTB is now part of the official Linux Kernel. We use Netem to limit the rate at which host B sends packets to host A in the in the monitored flow. The limit we impose is 7Mbps . In the same way, we impose a limit of 1 Mbps in the data rate at which the host A send packets to host B. Doing this way, we want to simulate the bottleneck represented by the ADSL uplink.

It should be noticed that, to slow down the link exiting from host A, we use both Netem and Ethtool. We do this beacuse during our experiments we observed that slowing down the 100Mbps ethernet interface only via Netem may lead to an instable evolution of the RTT we use as ground thruth (as can be seen in Fig. 3.3 - for now observe only RTT (green line) and ignore the rest). In particular, the RTT goes up and down very rapidly. We suppose that this is a consequence of the token bucket

[13]See http://en.wikipedia.org/wiki/Ethtool
[14]See http://www.linuxfoundation.org/collaborate/workgroups/networking/netem
[15]On some network interfaces it may not work

mechanism: when there are free tokens, they are rapidly consumed by iperf client that can send a lot of packets and fill the queue, thus imposing high round trip times[2]. In the time intervals when there are no more tokens, the packets of iperf client cannot go out and have to wait until they are recreated. In these intervals the queue has the chance to partially empty and the round trip times decrease. Fig. 3.3 shows also that when Netem is used in conjunction with Ethtool, the RTT evolution is more natural, because the Ethernet interface physically works at 10 Mbps and Netem has to slow down the link only by a factor of 10 (instead of slowing down by a factor of 100 like when using only Netem).

Because we want to avoid these irregular and unnatural behaviors, we choose to use Netem and Ethtool together.

Fig. 3.4 completes Fig. 3.2 with some more details about the tools that we use.

Replicating the experiments

The experiments that we propose for the validation are easily reproducible. They are automated by means of distributed bash scripts. To have an idea of what the scripts are and what process are launched, see Fig. 3.5.

Two identical version of our Tstat implementation are present on both host A and host B. Some preliminary configuration variable may be set up[16]

Then, launching `ping_validation/exp2-brain-iperf.sh`, the entire experiment is performed. This script is the "orchestrator" and is responsible to launch all the softwares we need. It runs in host A but is able also to launch processes in host B[17]. This

[16] In `ping_validation/exp2-brain-iperf.sh` on the host A side and in `ping_validation/exp2-ping-netbook.plot.sh` and `ping_validation/exp2-netbook-variables_conf.sh` on the host B side. The word "netbook" that appears in the names of the host B side scripts is not meaningful. It appears only because in our experiment we originally used a netbook as host B.

[17] Some network settings are needed on both host A and host B:

- `/etc/hosts` file must be edited in host A, to associate the IP address of host B to the name "netbook"
- `/etc/hosts` file must be edited in host B, to associate the IP address of host A to the name "desktop"
- from host A it must be possible to access via SSH the host B as root user without password. A good howto about this is:

  http://www.linuxproblem.org/art_9.html
- `ping_validation/exp2-brain-iperf.sh` must be launched with root privileges

script also calls a script on host B, `ping_validation/exp2-ping-netbook.plot.sh`, that is delegated to launch other processes on host B.

First, the bottlenecks are created on host A and host B. Then, the iperf servers start (both the one for the monitored flow and the one for the cross traffic flow) to listen on their respective ports. On host B, tcpdump starts to sniff the monitored flow. At the end of the experiment, it will produce a pcap trace[18], useful for further analysis of the experiment. Then host B starts to ping host A. The round trip times are recorded in the ping output file, together with the relative timestamps [19]. Finally, the monitored and the cross traffic flows start.

Analyzing the results

After running the experiment, it's possible to obtain plots like Fig. 3.3 using the the pcacp trace and the ping output file produced with the previous procedure. Launching `ping_validation/exp2-netbook-offline-plotting.bash` Tstat analyzes the pcap trace producing its log files[20]. At the end of the process, a gnuplot script examines the Tstat log file and the ping output script and generate the plots [21]. Separating the experiment runs from the analysis of results permits to compare the results produced by different configurations of Tstat on the same pcap trace.

To have an idea of the data taken as input, the data produced during the analysis and the scripts and sofwares involved, see Fig. 3.6.

3.3.2. Impact of parameters and surrounding conditions

The impact of the quadrant size

At low level, tcptrace (and thus Tstat too) keeps sequence numbers in a circular data structure named quad (i.e., after the "quadrants" the structure is divided into,

[18] The file name is specified by the variable `PCAP_TRACE` of `ping_validation/exp2-netbook-variables_conf.sh`

[19] The file name is specified by the variable `PING_OUT_FILE` of `ping_validation/exp2-netbook-variables_conf.sh` .

[20] All the log files written by Tstat will be in the folder specified by the variable `TSTAT_OUT_FOLDER` of `ping_validation/exp2-netbook-variables_conf.sh` .

[21] An eps file is generated. Its path is specified in `ping_validation/build_gnuplot_script.bash` .

to speed-up lookup). In case the quad has a fixed size, it may happen that, if the number of outstanding segments grows larger than the quad size, then sequence numbers are overwritten – so that acknowledgements cannot be paired with data and RTT samples are lost. We show an occurrence of this problem in Fig. 3.7, where we configured two values of the quad size[22] – one is fixed (to a purposely small value) and the other is variable and can grow arbitrarily large to avoid overwriting outstanding sequence numbers (notice that variable size is handled with linked lists, so that in this validation phase we take precisely the opposite direction to [GCCK13], as we do not want to compromise accuracy).

Fig. 3.7 and Fig. 3.8 represent three measurements on exactly the same traffic trace[23]. Each measurement correspond to a configuration of Tstat. For each of the three configurations, we provide three plots: in the top plot we compare the ground truth (the RTT of the ICMP flow) with our estimation. In the bottom plot we represent the number of acknowledgments that host A sends to host B every second in the monitored flow and the validity ratio, i.e. the ratio between the number of valid acknowledgements observed in a second (see sec. 2.2) and the total number of acknoledgements (either valid or invalid). In the middle plot, we represent how many queueing delay samples Tstat is able to calculate. These samples are the ones that Tstat uses to calculate the aggregated queueing delay, every second (see sec. 2.5).

First, we easily observe that when the cross traffic is inserted, it saturates the link from host A to host B. Therefore, the queueing delay increases, as confirmed by the ground truth. As expected, the number of acknowledgements that succeed in arriving to host B decreases and thus Tstat has less information to infer the queueing delay, as confirmed by the decreasing of the number of queueing delay samples per second. Nevertheless, the qd_samples/sec value does not depend only on the number of acknowledgements but also on the quadrant size. From Fig. 3.7, it is clear that with a small quadrant size the number of qd_samples/sec that can be calculated is very small for sequence numbers are possibly overwritten. Observing the validity ratio plot, we can grasp why it happens: with a small quadrant size, most of the acknowledgements are marked as invalid, and cannot be used for our estimation. For the configuration `MAX_SEG_PER_QUAD`=2, the validity ratio never exceeds 0.25, while for the configuration `MAX_SEG_PER_QUAD`=INFTY it reaches 1.0 .

As a consequence, for small values of the quadrant size, Tstat estimation is very

[22]The quadrant size is represented by `MAX_SEG_PER_QUAD` in `tstat/param.h`
[23]We sniff the traffic with tcpdump and then apply the measurement on the offline trace

coarse, while, increasing the quadrant size, Tstat better approaches the ground truth (in Fig. 3.7, compare the top plot of the two configurations, in particular after second 55). We point out that, when quadrant size is small, the error is not "random" but systematically may lead to an underestimation of the queueing delay: in particular, during our experiments we observed that when the ICMP RTT is not so high Tstat gives a good estimation but when the RTT is high Tstat cannot produce its estimates. Providing estimates of the queueing delay only when it is low clearly is a measurement bias.

The Fig. 3.8 represents an intermediate case: the validity ratio is slightly lower than with `MAX_SEG_PER_QUAD`=INFTY, and so the number of queueing delay samples per seconds; nonetheless Tstat succeeds in estimating the queueing delay.

The results that are presented in the following chapter are obtained with `MAX_SEG_PER_QUAD`=INFTY. The considerations that we've given in this section highlight that to infer the queueing delay (like when performing any other measurement), attention should be paid in tweaking the measuring tool, otherwise the results may be distorted. We point out that other authors do not report much on tweaking the tool they use.

Dependency on the data rate

In this subsection, we show that the expected data rate should be taken into account when tweaking Tstat. For the data rates that we have when using the bottlenecks (Netem and Ethtool), a `MAX_SEG_PER_QUAD`=5 is enough to have a good estimation of the queueing delay. Now, we provide the results obtained replicating the experiments of sec. 3.3.1 without any bottleneck (neither in host A nor in host B, neither Netem nor Ethtool). Both the monitored flow and the cross traffic flow are free to run at the maximum speed.

Fig. 3.9, Fig. 3.10 and Fig. 3.11 show that the number of acknowledgements in the monitored flow is more than a magnitude larger than the previous experiments, but the validity ratio is almost 0 even with MAX_SEG_PER_QUAD=15 and thus the number of queueing delay samples per second is not sufficient. In the previous experiments, with
MAX_SEG_PER_QUAD=2 the estimation was coarse but at least possible. Now, with that configuration, Tstat is not able to provide any estimation. While in the previous example MAX_SEG_PER_QUAD=5 guaranteed a good estimation,

now the estimation is very coarse. To have a good estimation, we have to set MAX_SEG_PER_QUAD=15.

3.4. Performance analysis and analysis of the overhead

In this section, we coarsely analyze the overhead imposed on Tstat by the queueing delay estimation both in terms of required disk space for the output files and of execution time.

We run Tstat on a set of 59 compressed pcap traces. The total size of the traces is 43 GB. With bufferbloat analysis disabled, Tstat running time is 23 minutes and the output log files are 1,462GB. Enabling bufferbloat analysis, the log files increase to 2,328 GB and running time increases to 25 minutes. We can conclude that the bufferbloat analysis imposes an overhead of 60% in terms of required space on the disk and of 8% in terms of running time.

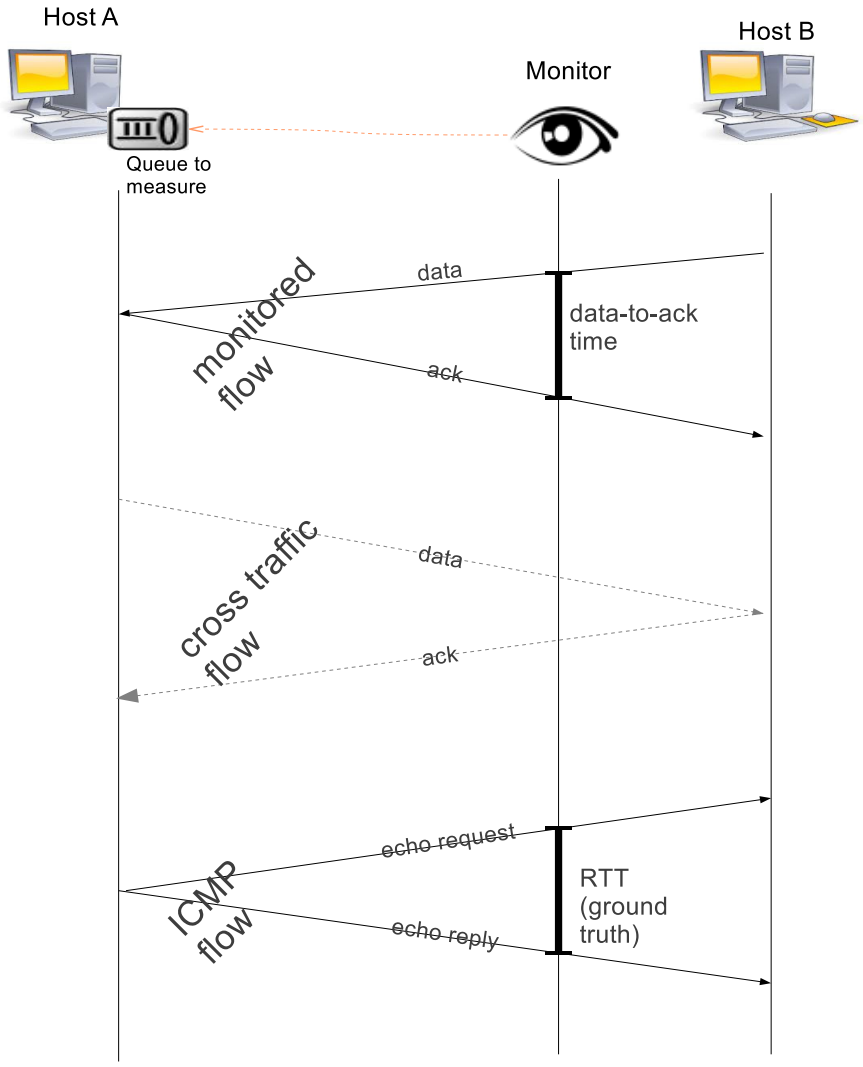


Figure 3.2.: Local testbed

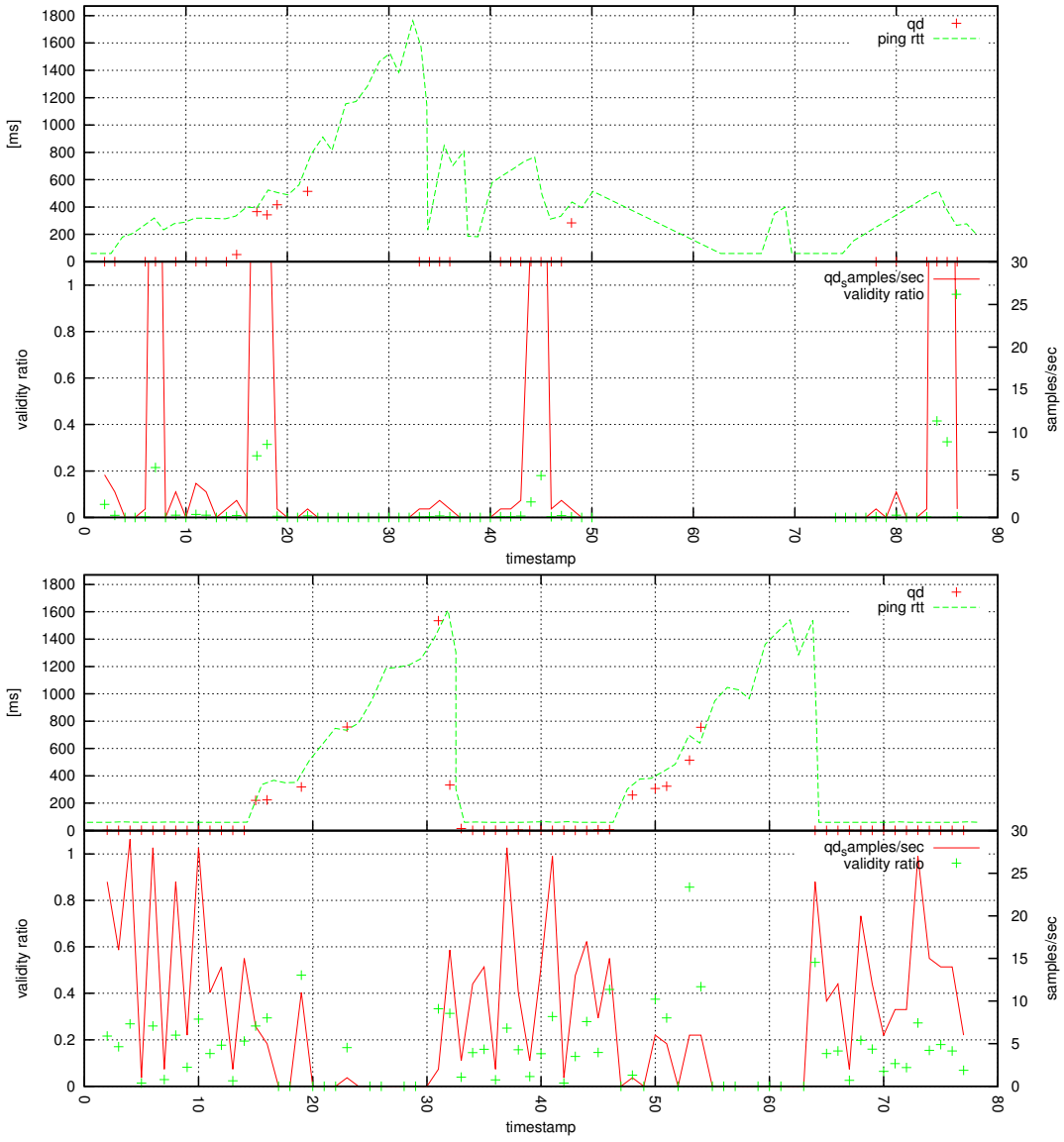


Figure 3.3.: (top) RTT (green line) when only Netem is the host A bottleneck. (bottom) RTT (green line) when Netem is used in conjunction to Ethtool.

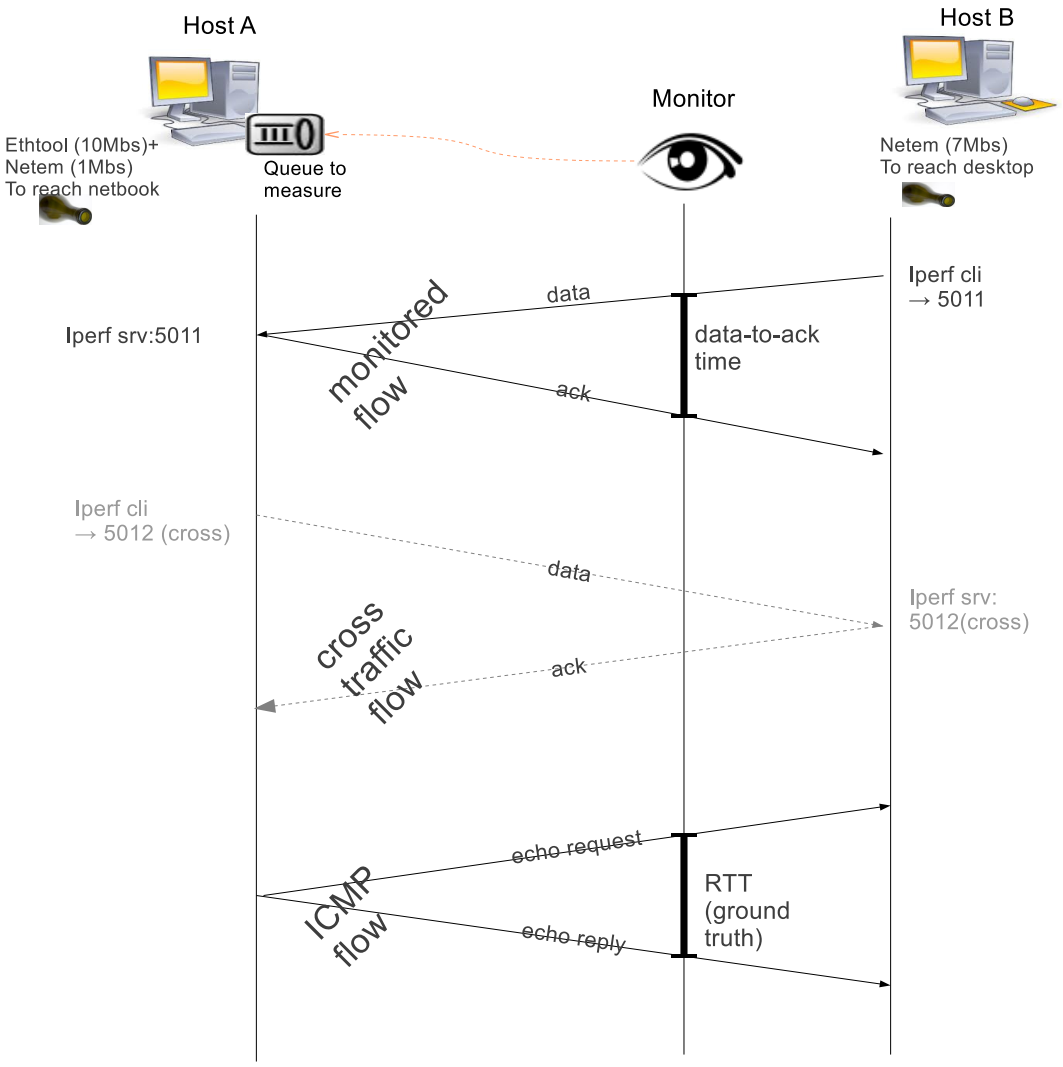


Figure 3.4.: Local testbed: detailed schema

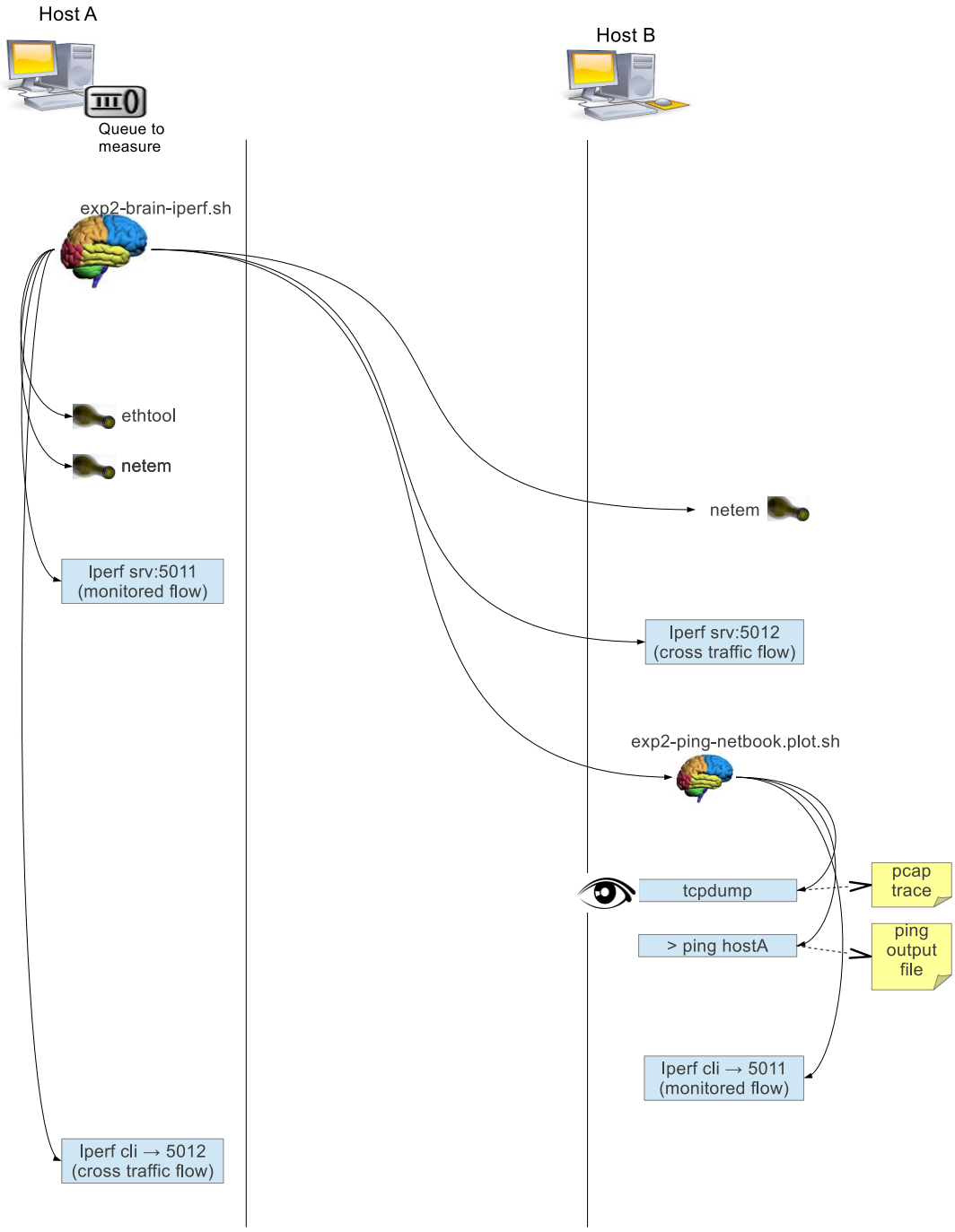


Figure 3.5.: Testbed automating scripts

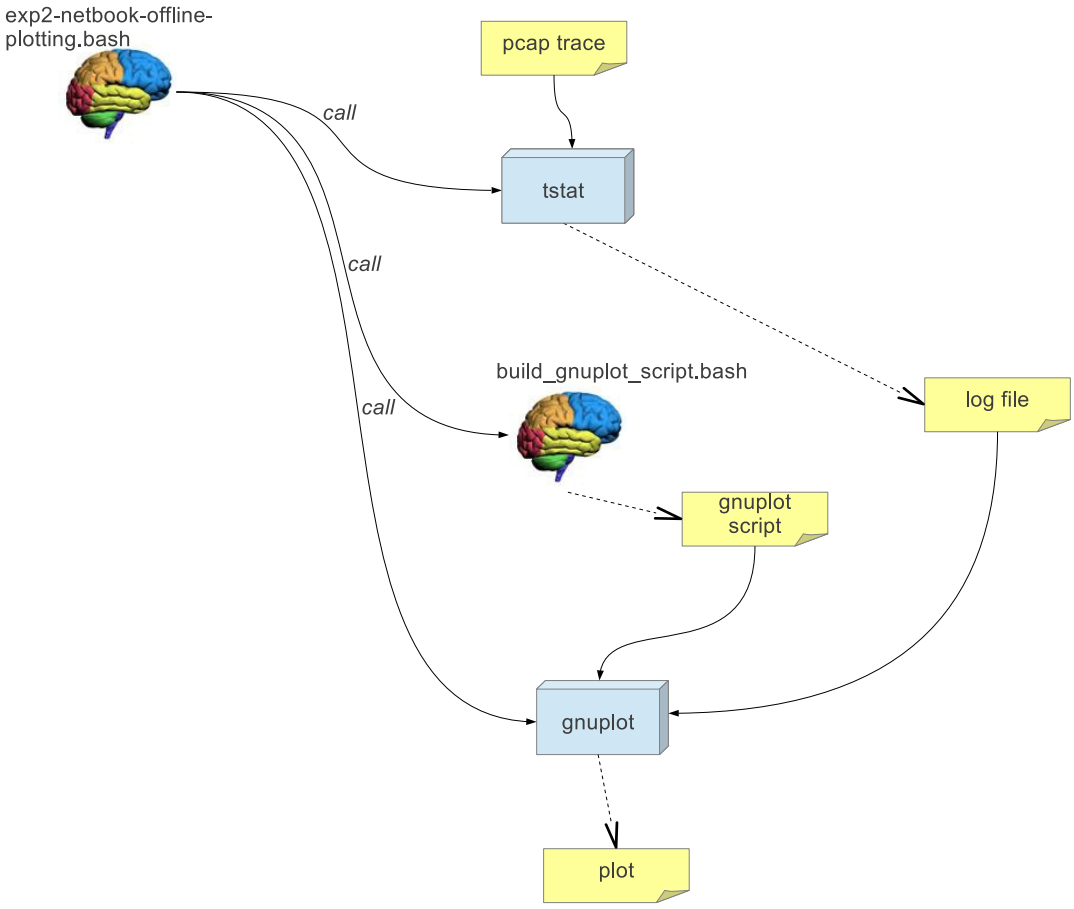
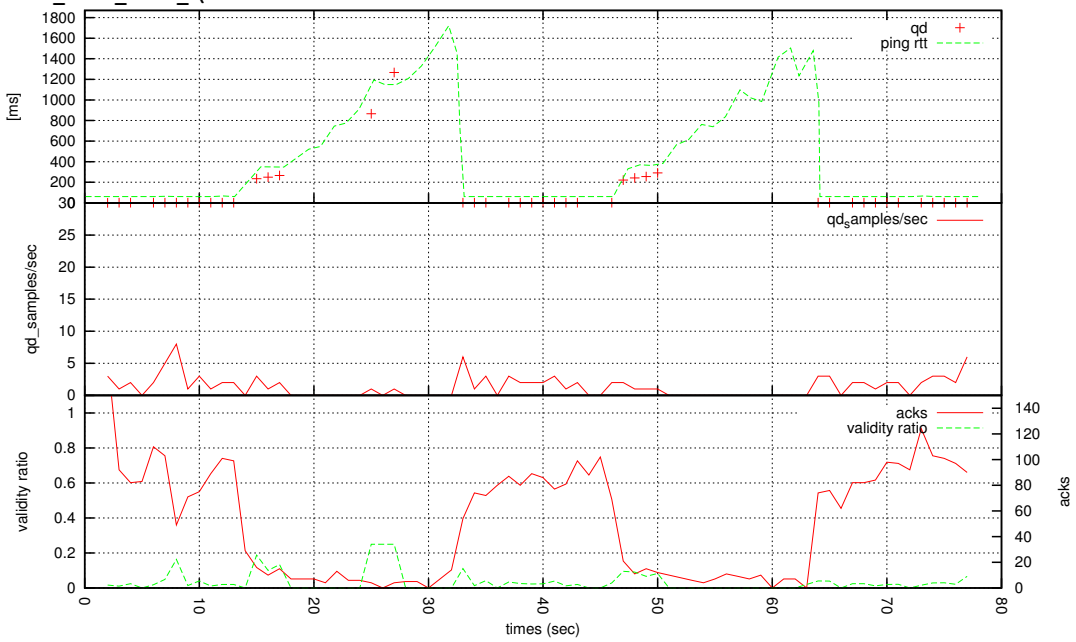


Figure 3.6.: Offline analysis process

MAX_SEG_PER_QUAD=2



MAX_SEG_PER_QUAD=INFTY

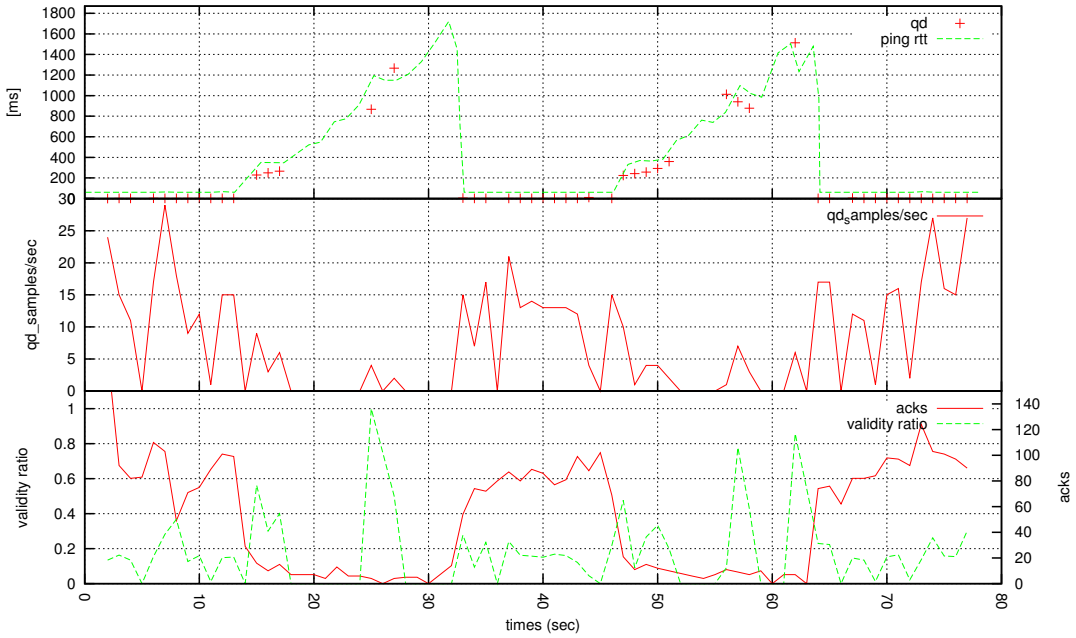


Figure 3.7.: Queueing delay estimation using different values of quadrant size. MAX_SEG_PER_QUAD=2 in the top figure and MAX_SEG_PER_QUAD is INFTY in the bottom figure (INFTY means that Tstat keeps track of all the segments: the only limit is the system meory).

MAX_SEG_PER_QUAD=5

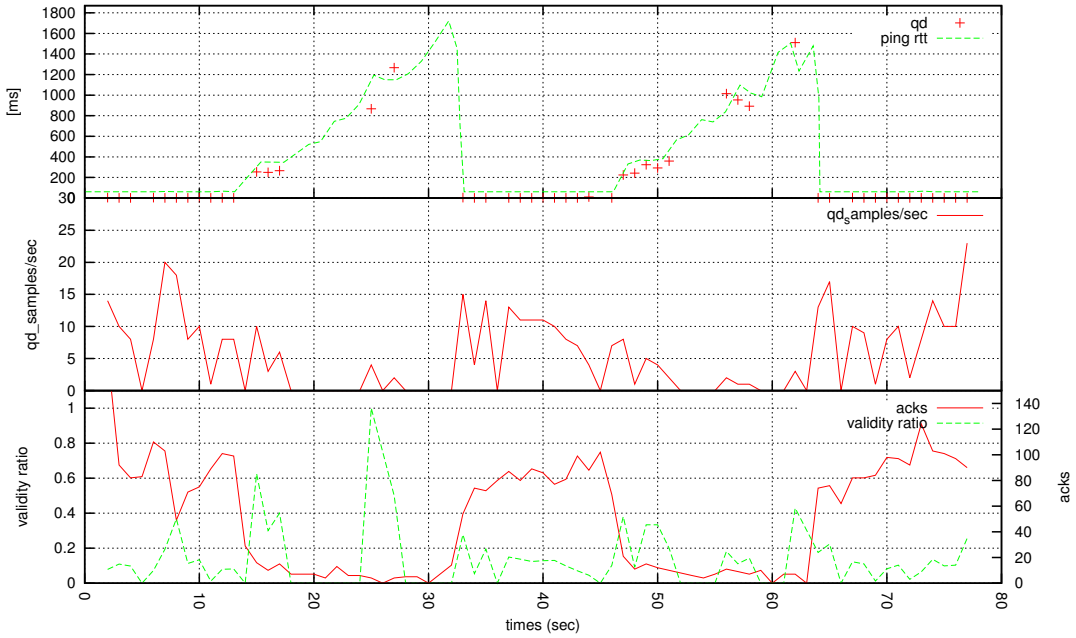
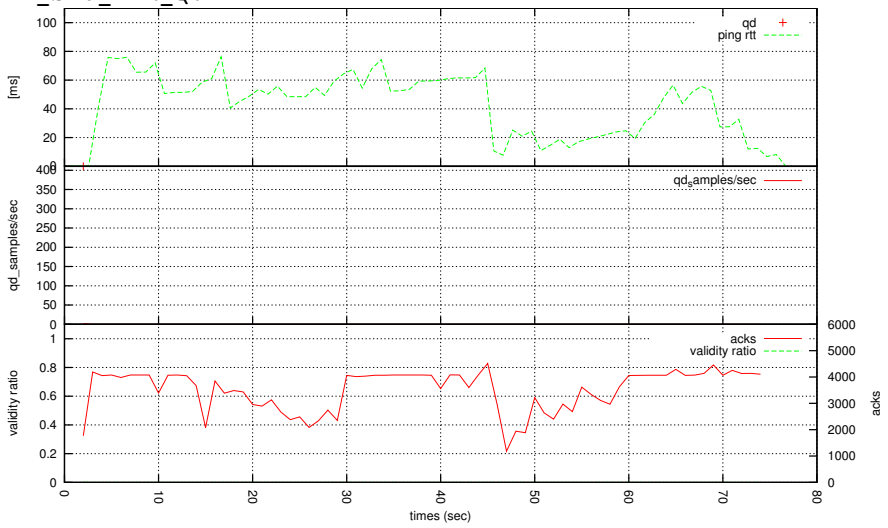


Figure 3.8.: Queueing delay estimation with MAX_SEG_PER_QUAD=5

MAX_SEG_PER_QUAD=2



MAX_SEG_PER_QUAD=5

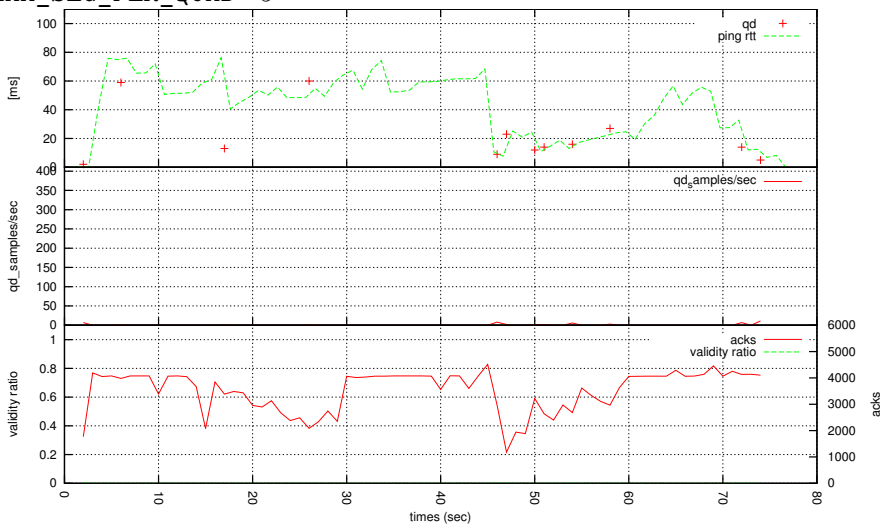
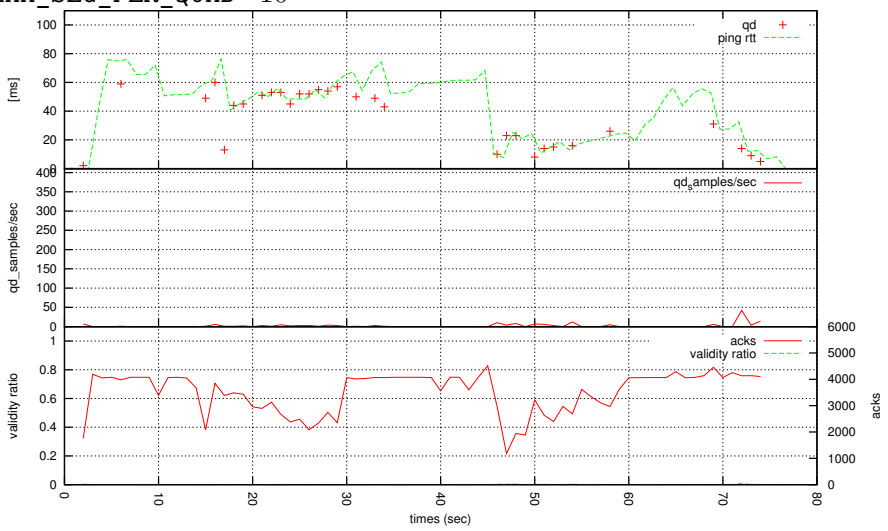


Figure 3.9.: Experiment without bottlenecks

-

MAX_SEG_PER_QUAD=10



MAX_SEG_PER_QUAD=15

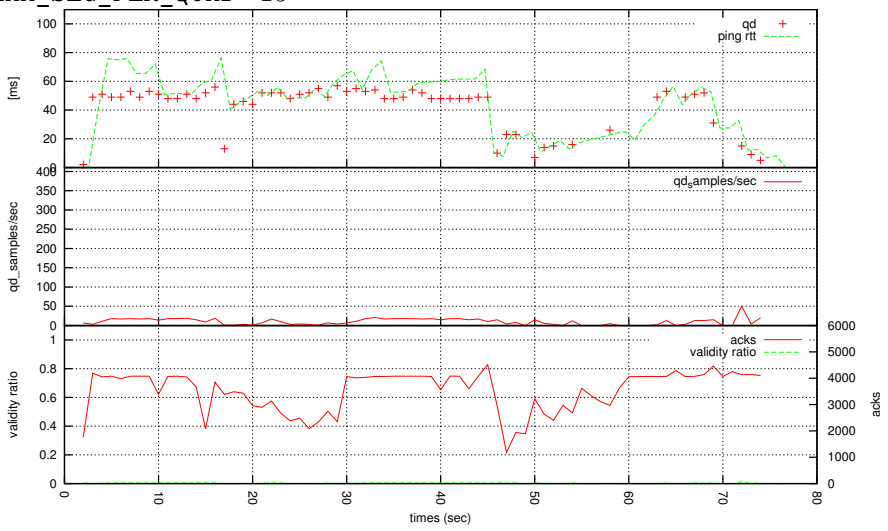


Figure 3.10.: Experiment without bottlenecks

MAX_SEG_PER_QUAD=INFTY

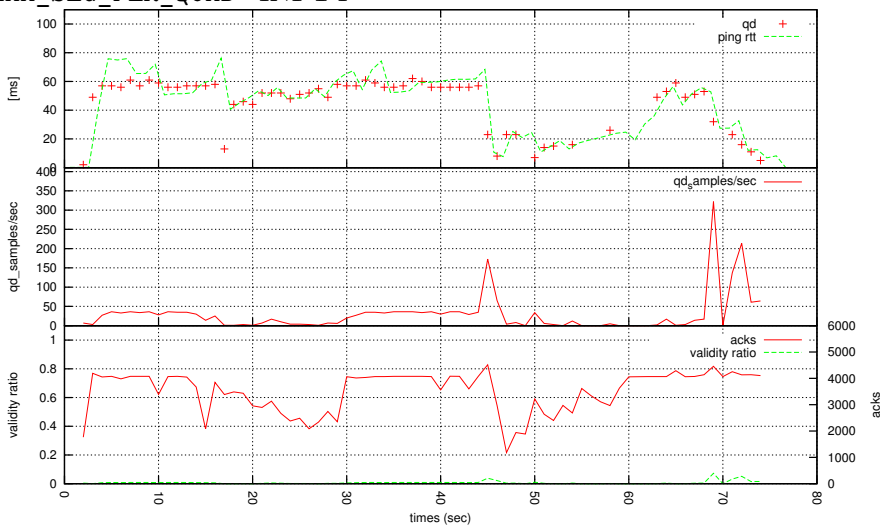


Figure 3.11.: Experiment without bottlenecks

4. Statistical characterization of queueing delay

4.1. Introduction

As our focus is to build a solid methodology, rather than providing full-blown measurement campaign, for the time being we analyze offline traces gathered in a single ISP network during FP7 NapaWine [LMH+08] project, when P2P was still fairly popular (we consider a 8 hr-long period starting at 14:00, gathered during 2009).

We consider only the flows in which we observe acknowledgements originated by an internal host and directed to an external host[1]. According to Fig. 3.1, internal hosts are the ones that access Internet connecting to the DSLAM that we are monitoring. The rest of the hosts are considered external.

For the sake of simplicity, we will often refer to three ranges of aggregated queueing delay (aqd):

- LOW: $aqd < 100ms$
- MID: $100ms \leq aqd < 1000ms$
- HIG: $aqd \geq 1000ms$

We make the assumption that performance of interactive multimedia (e.g., VoIP, video-conference and live-streaming) or data-oriented applications (e.g., remote terminal or cloud editing of text documents) require to have queueing delay in the range LOW to guarantee a good quality of service; mildly- interactive applications require at least the MID range; the other applications do not suffer much even if the queueing delay is in the HIG range.

[1] An host (identified by a an IP address) is specified to be internal or external by configuration files (see sec. 3.2.1)

4.2. Global characterization

Fig. 4.1 shows how the aggregated queueing delay is distributed. We observe that most of the aggregated queueing delays fall in the LOW range, but that the MID region is not negligible.

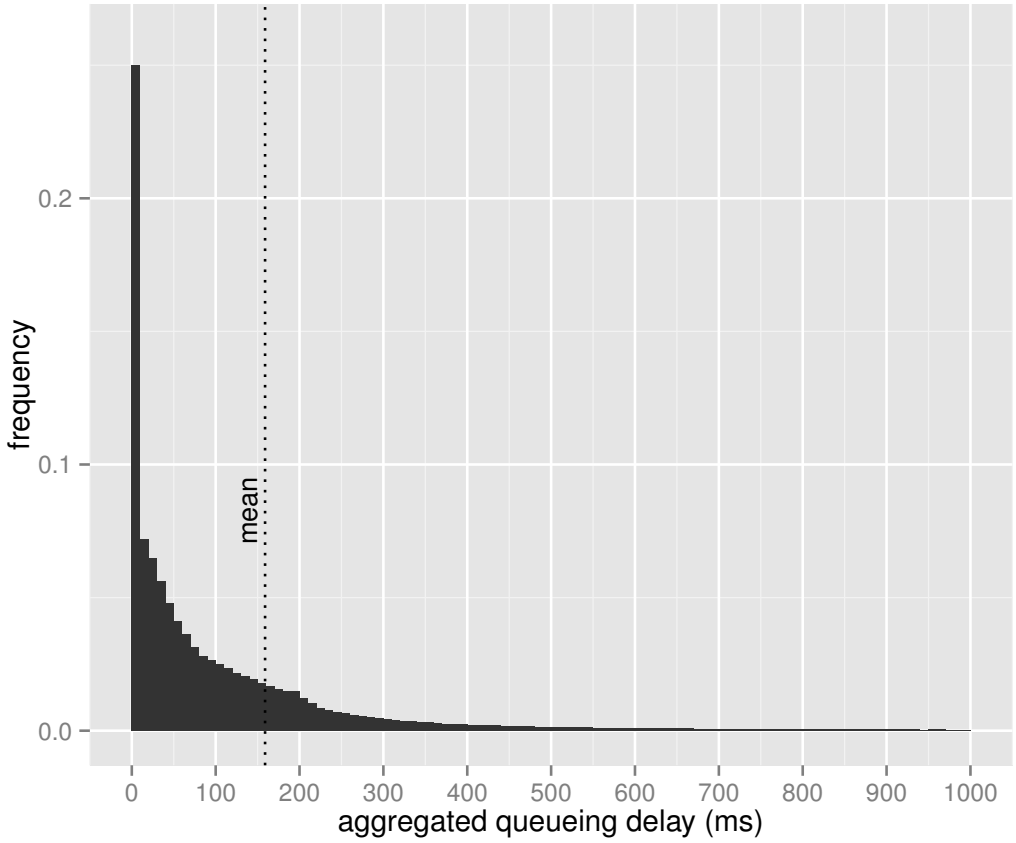


Figure 4.1.: Normalized frequency plot of the aggregated queueing delay (cutting away from the plot the queueing delays greater than $1000ms$). Every vertical bar represents a 10-ms large interval. The height of each bar is the number of aggregated queueing delays that fall in that interval divided by the total number of aggregated queueing delays.

This emerges in a clearer way from Fig. 4.2. For each flow, we calculate the 50th, 90th, 95th, 99th percentiles and we plot the cumulative distribution function (CDF) of these values.

More precisely, indicate with $\mathcal{F}$ the set of all the flows and take a certain flow $F \in \mathcal{F}$.

Without ambiguity, we can indicate with the symbol F, either a flow itself or the set of aggregated queueing delays of that flow, neglecting the void windows, i.e. the windows in which no queueing delay sample has been observed. The 50-th percentile of F is the value p_F such that:

$$\frac{|\{aqd \in F | \, aqd \leq p_F\}|}{|F|} = 50\%$$

i.e. p_F is the value such that the 50% of the aggregated queueing delays does not exceed it.

The 90th, 95th and 99th percentile for each flow are calculated as above.

To plot the 50th-percentile curve, we consider the population $P = \{p_F | \, F \in \mathcal{F}\}$ consisting in the set of the 50th percentile value p_F for all the flows. The curve represents the cumulative distribution function of that population, i.e.

$$CDF\,(x) = \frac{|\{p_F \in P | \, p_F \leq x\}|}{|P|}$$

The complementary cumulative distribution function is

$$CCDF\,(X) = \frac{|\{p_F \in P | \, p_F > x\}|}{|P|} = 1 - CDF\,(x)$$

The curves relative to the 90th, 95th and 99th percentiles are calculated as above.

From the CCDF of Fig. 4.2 we see that almost the 20% of the medians (50th percentiles) fall in the MID or HIG ranges. This means that almost the 20% of the flows experience, in the 50% of their 1-second-windows, a delay greater or equal to 100 ms.

In Fig. 4.3, we provide the frequency plots of the percentiles of the aggregated queueing delay. The 50th percentile plot, for example, is obtained calculating the 50th percentile for each flow and then observing the frequency at which these values occur. The same holds for the other plots of Fig. 4.3.

From these plots, we can grasp interesting insights. For example, considering the 90th percentile plot, we can see that the average 90th percentile (calculated over all the flows) is greater than 100ms. Roughly speaking, we can conclude that, taking an "average flow", the 10% of its 1-second-windows fall in the MID or HIG range. This is a clue for a non-negligible buffering.

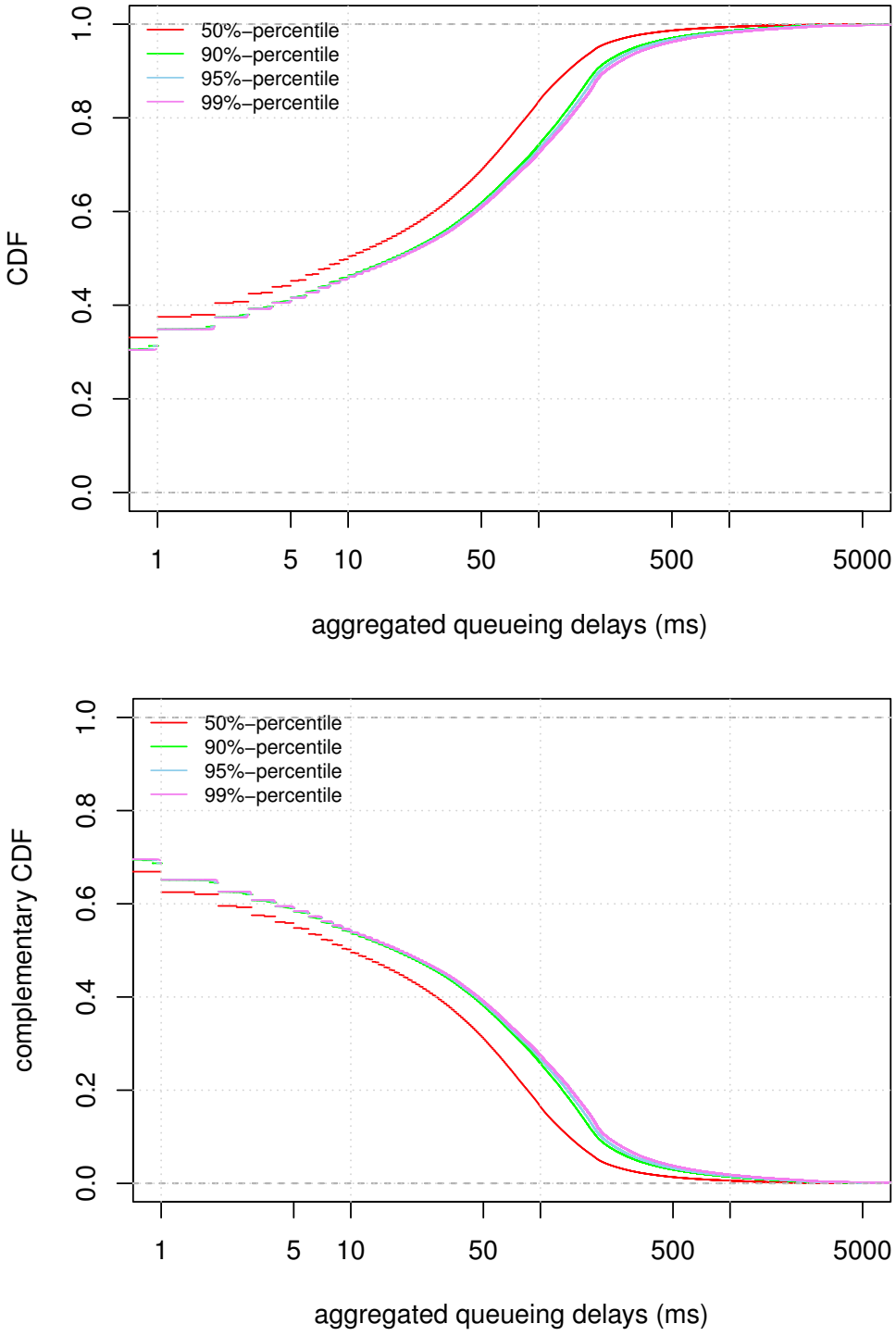


Figure 4.2.: Cumulative distribution function (CDF) and complementary cumulative distribution function (CCDF) of the aggregated queueing delay percentiles (calculated on a per-flow basis)

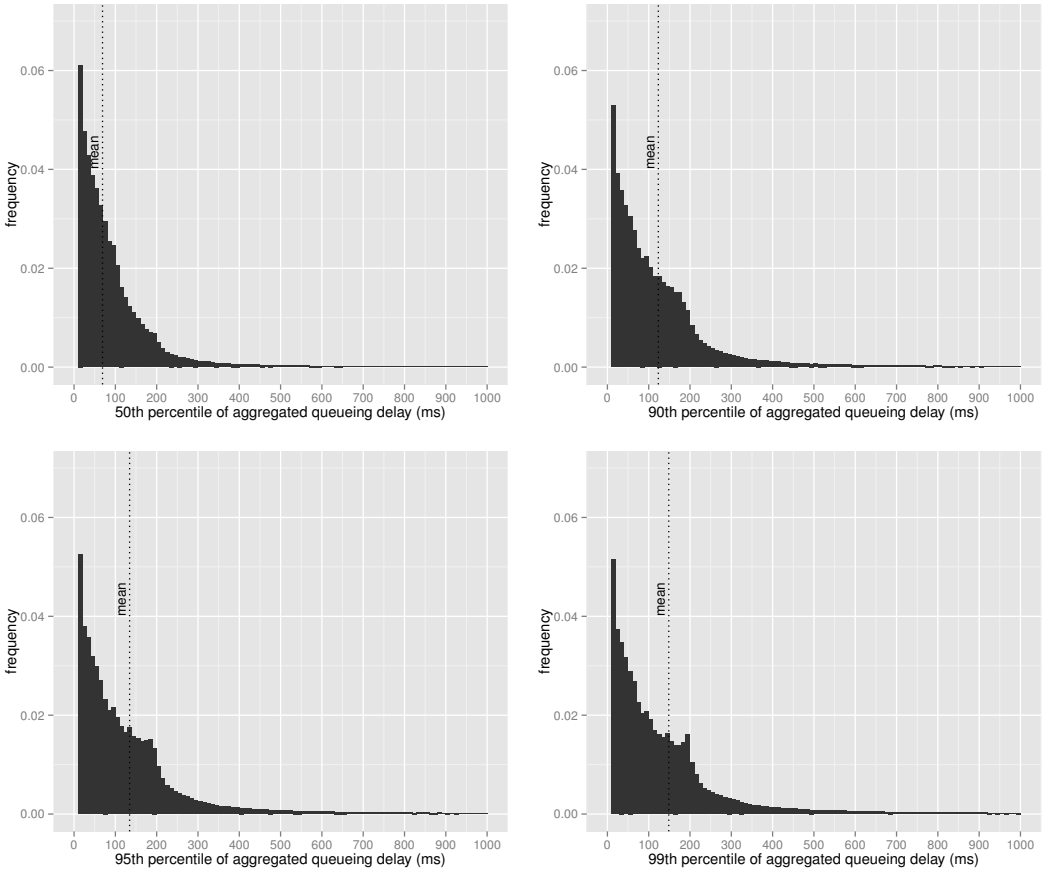


Figure 4.3.: Normalized frequency plots of the aggregated queueing delay percentiles. Every vertical bar represents a 10-ms large interval. The height of each bar is the number of the flows that have the percentile that fall in that interval divided by the total number of flows.

Fig. 4.4 shows the evolution of the quantiles of the aggregated queueing delay in the observation period. The plot has the granularity of 2 minutes: taking for example the 0.5th quantile curve, the point corresponding to a certain time value t is calculated as follows:

- take all the 1-second-windows that fall in the interval $[t, t + 2\text{min}]$, regardless of the flow that contains them
- take the aggregated queueing delays of these windows
- Calculate the 0.5th quantile of all these values

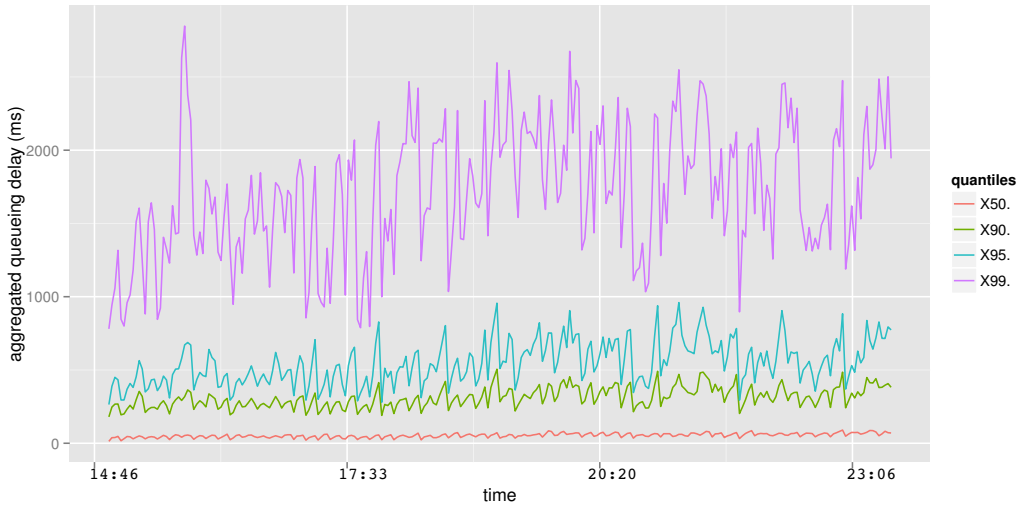


Figure 4.4.: Evolution of the quantiles during the 8-hour long observation.

4.3. Per application view of queueing delay

Each delay sample that Tstat measures carries an application label, obtained through Tstat Deep Packet Inspection (DPI) and behavioral classification capabilities. Though Tstat is capable of fine-grained classification of different applications ([FMM+11]), we cluster similar applications into few classes depending on the service they offer. They are listed in the Tab. 4.1. For each class of traffic, the last column indicates the queueing delay range (see sec. 4.1) which is suitable for that class to guarantee a good QoE (Quality of Experience). This classification is made according to studies assessing the impact of delay on the QoE: [BOB12, SdDF+11] for the WEB application, [HPC+12] for MEDIA (i.e. multimedia applications) , [CHHL06] for VOIP, [TR11, CR] for P2P. Nevertheless, it's worth highlighting that when the queueing delays fall in the HIG region, also less demanding traffic classes as P2P may start to experience a severe degradation of performance ([TR11, CR]).

All flows that Tstat is not able to classify are considered as OTHER.

4.4. Queueing delay distribution over classes

In Fig. 4.5 we depict a jittered density map of the aggregated queuing delays (y-axis) for different application classes (x-axis), along with boxplots reporting the quartiles

Class	Protocols	Requirements
P2P	Bittorrent ED2K obfuscated ED2K EMule Kazaa DirectConnect Gnutella Soul Seek	LOW
MAIL	SMTP POP3 IMAP	LOW
WEB	HTTP SSL	MID
MEDIA	RTMP (flash video) ICY (SHOUTcast)	MID
CHAT	MSN YMSG (yahoo messanger proto) XMPP (Jabber or Google Talk)	HIG
SSH	SSH	HIG
VOIP	RTSP RTP+RTSP	HIG

Table 4.1.: Classes of traffic

(and 5th, 95th percentiles). Applications are ordered, left to right, in increasing order of delay sensitivity. It can be seen that, for most applications, the 75% of windows experience less than 100ms worth of queuing. The only exceptions are constituted by, rather unsurprisingly, P2P applications and, somehow more surprisingly, Chat applications, with median delays exceeding 100ms.

In the ideal case, the most delay-sensitive traffic classes should be better serviced (i.e. they should experience less queueing delay) than the other ones. Therefore we would expect the values of the percentiles to go down from left to right (i.e. from the less sensitive traffic classes to the more sensitive ones), but Fig. 4.5 does not confirm this. Intuitively, we can conclude that some traffic classes are better serviced than other more sensitive ones. For example MAIL, WEB and MEDIA are better serviced than CHAT. MAIL is serviced in quite a similar way to WEB. Fortunately, almost all the aggregated queueing delay of VOIP flows fall into range LOW.

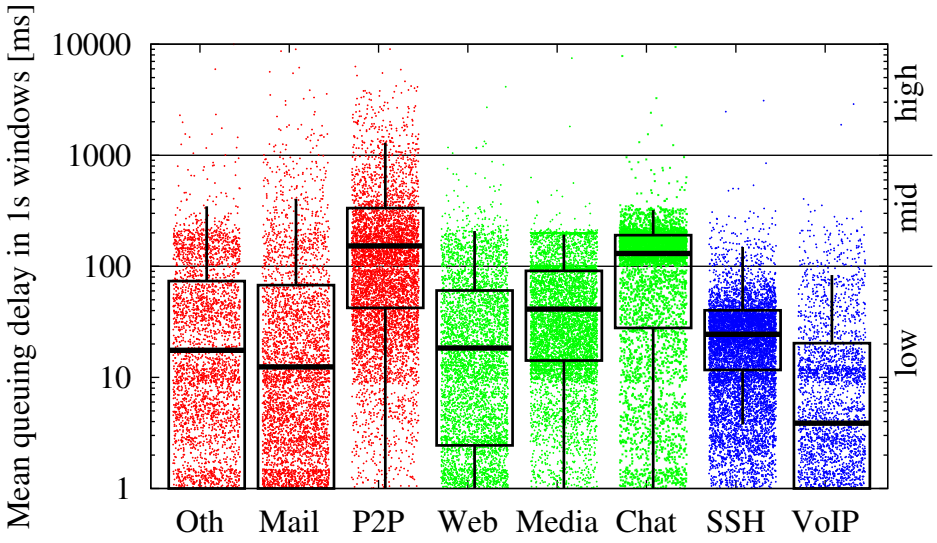


Figure 4.5.: Breakdown of queuing delay per application. Plot reports jittered density maps and (5,25,50,75,95)- th percentiles.

A simplified view of the information carried by Fig. 4.5 is given by Fig. 4.6. As expected, a big part of the aggregated queueing delays of P2P flows fall into MID and also HIG range. Surprisingly, the MID part of the CHAT bar is even bigger than the corresponding part of P2P. On the other hand, the HIG part of CHAT is almost of 0-size, while in P2P it is relevant. Similarly, LOW part of MAIL is bigger than the LOW part of WEB, but MAIL shows a non-null HIG part that is non existent in WEB. The more demanding classes, VOIP and SSH, are partitioned in a quite similar way: most of the aggregated queueing delays fall in LOW and the HIG part is null.

To have a more precise picture of the information contained in Fig. 4.6, the Tab. 4.2 tabulates the percentage of 1 sec windows for each application that fall into either of the three delay regions, where we use boldface values to highlight possible QoE degradation. It follows that, in practice, bufferbloat impact on QoE appears to be

modest. A limited 0.1% of Web and Chat sessions may be impacted by significant delay, and 2.2% (1.4%) of VoIP (remote terminal) sessions may be impacted by moderate delay. P2P clearly stand out, raising the odds to induce high delays (2.9%) followed by MAIL (0.6%), though with likely minor impact for the end-users.

	CHAT	MAIL	MEDIA	OTHER	P2P	SSH	VOIP	WEB
LOW	45.7	93.2	86.2	89.7	58.4	98.6	97.8	91.9
MID	**54.1**	6.2	**13.2**	10.2	38.7	**1.4**	**2.2**	8.0
HIG	0.1	0.6	0.0	0.1	2.9	0.0	0.0	**0.1**

Table 4.2.: Table report the percentage of per- application aggregated queueing delays falling into either of the three delay regions (boldface values highlight possible QoE degra- dation)

The plots in Fig. 4.7 are obtained partitioning all the aggregated queueing delays on their range and then, for each range, representing the count of the traffic classes (i.e. how many aggregated queueing delay we observe for that class). What clearly emerges is that the monitored traffic is dominated, at least in our per-flow view, by the P2P. A motivation behind this is that most P2P applications use multiple connections to support various functions.

Excluding P2P and OTHER from the plots in Fig. 4.7, we obtain Fig. 4.8 which permit a more comfortable analysis. We see that, not considering P2P, WEB is the most popular class of traffic. All classes are sufficiently represented in the LOW range. In MID range CHAT is quite frequent. MAIL is particularly frequent in the HIG range, as we may hope.

4.5. Root cause analysis

4.5.1. Methodology

As we have full knowledge of the traffic generated and received by hosts, we can correlate the queuing delay with the traffic active on the hosts. As before, we limitedly consider the upstream traffic direction, and divide time in 1 second long slots. For all active flows of each host, we log the application label and its aggregated queueing delay during that window. Notice that in sec. 4.3 we independently consider all the aggregated queueing delays: in other words, the delay seen by packets of an

application flow can be induced by other flows active on the same time in the same host (or household). In this section, we leverage standard data mining techniques to, if possible, pinpoint the root cause of the observed delay. In particular, we use Apriori ([AS+94]), a classic algorithm for frequent item set mining.

In our analysis, we face known, yet non trivial problems. At this stage, we make the problem tractable by performing stratified sampling, initially including 8,000 random aggregated queueing delays per-application (64,000 overall). In this way, we also cope with class imbalance: taking the same number of aggregated queueing delays for all the applications permits to avoid the bias due to the fact that there are very popular classes of traffic and other less popular ones. Let us denote by (t, h, a, q) an aggregated queuing delay q gathered on host h during the time window t and corresponding to an application a. Evidently, we need to ensure that our population includes all other aggregated queueing delays corresponding to applications that were active on h during t. We therefore complement the initial population and achieve an overall population of 107,825 aggregated queueing delays. To guarantee the statistical relevance of our analysis, we further verify that statistics of the sampled population correspond to those reported in Fig. 4.5.

For each host h and time window t, we next compute the mean of the aggregated queueing delays seen by all applications during t. As rule-inference techniques are known to be largely ineffective on continuous variables, and in reason of the previous QoE considerations, we quantize queuing delay in a low, medium and high score (see sec. 4.1).

We also aggregate all application labels, and denote with $\surd$ (or ++) applications that have generated respectively one (or more) flows during t. For the sake of illustration, consider 2 applications are active at host h during $t = 0$. One is a Chat application, consisting of a single flow, the other is a P2P application with 3 active flows during $t = 0$: we thus aggregate the application labels as (Chat$\surd$, P2P++). Notice that this criterion again implements an aggressive quantization into classes: intuitively, we argue that there is more information in knowing coarsely whether there was a single or more flows, rather than precisely distinguishing the exact number of flows. Following this spirit, we additionally encode the previous example with purely binary indicators as in (Chat$\surd$, P2P++).

We yet have to perform a final step in order to facilitate the root cause analysis. Namely, we group together windows, irrespectively of the host, having the same

overall amount of active applications [2]. Intuitively, as windows with x active flows for an host will not have the same frequency as windows with y concurrently active flows in our dataset, grouping windows by the number of active flows allows to let infrequent (i.e., with small support) but still interesting (i.e., with high confidence) rules to emerge.

4.5.2. Representation of the rules

Each rule is defined as $A \rightarrow C$, where A is the antecedent of the rule and C is the consequent.

The antecedent A of a rule is a description of a possible (h, t) pair, i.e. the description of the outgoing traffic of host h observed in the 1-second window t. More precisely, A is a tuple with the following fields:

- Num: the number of applications that are simultaneously active in the host h during t
- Other: void if there is no active flow labeled as OTHER (i.e. an application that Tstat is not able to classify),$\surd$ if there is one active flow for such applications, ++ if there is more than one flows of that kind
- Mail: as above
- P2P: as above
- Web: as above
- Chat: as above

Each antecedent of rule is characterized by the support. The *support* $Supp\,(A)$ is the number of pairs (h, t) that are described by tuple A.

In our dataset, we can associate to each (h, t) pair the mean of the aggregated queueing delays of the outgoing flows of host h, observed in the 1-second window t. For the sake of simplicity we consider only the range which this value falls in: the range can be Low, Medium or High (corresponding, respectively, to LOW, MID, HIG of sec. 4.1). Now, we can explain what is the consequent of the rule.

[2]Notice that, for example, if there are 3 active flows for the application "P2P" and 1 active flow for the application "CHAT", we will count 2 active applications.

The consequent of a rule is a nothing more than a delay range. Evaluating a rule like

$$A \rightarrow C$$

means finding "how much" it is true that if a pair (h, t) is described by A, then the mean of the aggregated queueing delays of the flows on host h, observed in the 1-second window t, falls in the range indicated by C. Each rule is characterized by a confidence.

The *confidence* of $A \rightarrow C$ is

$$conf\,(A \rightarrow C) = \frac{n\,(A \wedge C)}{Supp\,(A)}$$

where $n\,(A \wedge C)$ is the number of pair (h, t) that are described by A and have the mean of the aggregated queueing delays falling in the region indicated by C. In other words, the confidence is the fraction of (h, t) pairs described by A that actually fall into the range C.

4.5.3. Experimental results

Results of Apriori are reported in Tab. 4.3 for both medium and high delays, where for each rule we report confidence and support. Rules are reported, top to bottom, for increasing size (i.e., number of application labels in the rule). For each rule size, rules are sorted, top to bottom, for decreasing confidence. Already from this small dataset, several interesting observation can be made.

First, notice that Mail can cause bufferbloat: this is summarized by rule (a), which can be expected due to the use of persistent TCP connections, used to send multiple messages, each of which encapsulates possibly large e-mail bodies due to the use of MIME to encode attachments of various kind (e.g., pictures, music, archives, etc.). Notice that mail application does not otherwise frequently generate delay, confirming the intuition that small textual messages have no impact on the QoE of other applications.

Second, high delays are also due to (b) multiple concurrent P2P flows or (c) single persistent HTTP connections. Interestingly, in the specular case in which (j) a P2P application has a single active flow, or (k) multiple HTTP connections are

Delay	Applications						Supp.	Conf.	(ref)
	Num	Other	Mail	P2P	Web	Chat			
High	1		✓				11	99	(a)
	1			++			10	96	(b)
	1				✓		10	64	(c)
	2			++	✓		20	98	(d)
	2	✓		++			28	87	(e)
	2	✓			✓		23	80	(f)
	2	++		++			14	61	(g)
	2			++	++		19	54	(h)
	3	✓		++	✓		13	99	(i)
Medium	1			✓			35	76	(j)
	1				++		44	53	(k)
	2			✓		✓	10	85	(l)
	2	++		✓			11	88	(m)
	2			✓	✓		19	85	(n)
	2			✓	++		10	77	(o)
	2	✓		✓			15	76	(p)
	2	++			++		16	72	(q)
	2				✓	✓	10	51	(r)
	2	✓			++		20	50	(s)

Table 4.3.: Rule inference with Apriori

active in parallel, the delay generally remains bound to the 100 ms-1 sec range. The intuition is thus that users often browse the Web in parallel to P2P transfers. However, packets of short lived TCP Web connections often pile up behind bursts of TCP packets due to multiple P2P connections. An implicit confirmation of this intuition comes from the fact that BitTorrent recently([SHIK10]) redesigned the data transfer, by introducing the delay-sensitive uTP protocol, aimed at bounding the queuing delay to no more than a configurable target parameter (set to 100 ms by default). Reason why a single Web connection should generate larger uplink queuing delay is instead less obvious, and needs further investigation.

Third, various combinations of P2P, Web and uncategorized traffic (Other) jointly concur in creating bufferbloat (we notice that uncategorized traffic possibly include

P2P applications for which Tstat has no valid classification signature). We again notice that rule (d) combining a single Web and multiple P2P flows, has a higher confidence that rule (h). Moreover, as the support of (h) is included in the support of (d), it follows that including multiple Web connections weakens the rule significance – as already observed comparing (c) against (k), the delay seems to be inversely correlated with the number of Web connections. Finally, we see that Chat sessions often happen in parallel with (l) P2P or (r) Web traffic – with the former especially cause of delay suffered by Chat application, in reason of the relative confidence of rules (a) and (j) with respect to (l). Hence, user behavior of Chatting in parallel to other applications is the cause of relative high frequency of medium delays (54% fall in the 100 ms- 1 sec range) seen early in Fig. 4.5. From this observation, we can also conjecture that Chat users likely do not consider these levels of delay harmful for QoE. Indeed, since all other applications have lower delay statistics, this may follow from the fact that user naturally deactivate data-intensive background applications (e.g., P2P) and do not perform other activities (e.g., Web browsing) while delay-sensitive applications are ongoing (e.g., VoIP calls).

4.6. How to obtain the plots

Almost all plots of this chapter are obtained with R. To obtain them from scratch, edit `offline_analysis/Rscripts/get_affected_flows.r`, insert the following lines as the main instructions and run that script.

`eps` files will be produced as output. To locate them, you can check the code.

To obtain Fig. 4.1 and Fig. 4.3, insert:

```
build_outgoing_windows_df()
calculate_percentiles()
build_frequency_plots()
```

To obtain Fig. 4.2, insert:

```
build_outgoing_windows_df()
calculate_percentiles()
plot_percentiles()
```

To obtain Fig. 4.4, insert:

```
build_outgoing_windows_df()
build_quantile_time_evolution()
plot_quantile_time_evolution()
```

To obtain Fig. 4.6, insert:

```
build_outgoing_windows_df()
plot_percentage_bar()
```

To obtain Fig. 4.7 and Fig. 4.8, insert:

```
build_outgoing_windows_df()
plot_class_distinguished_frequency_plots()
```

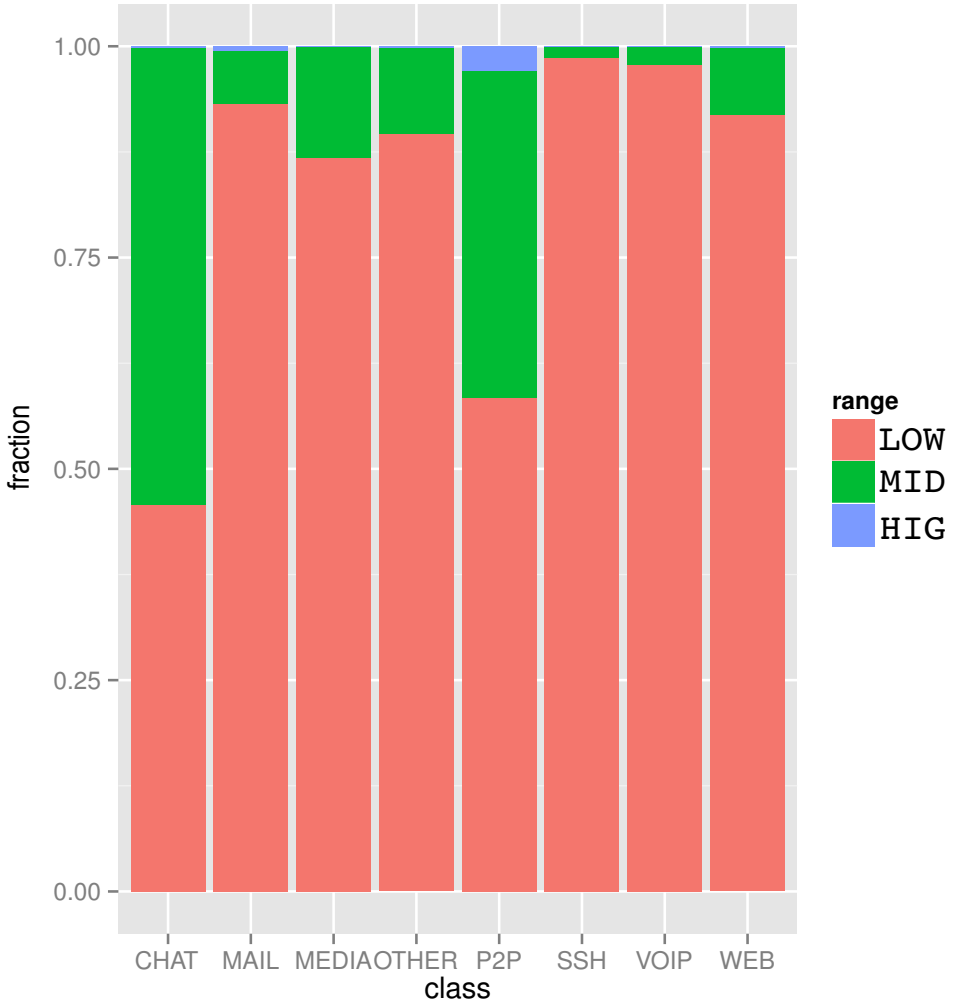
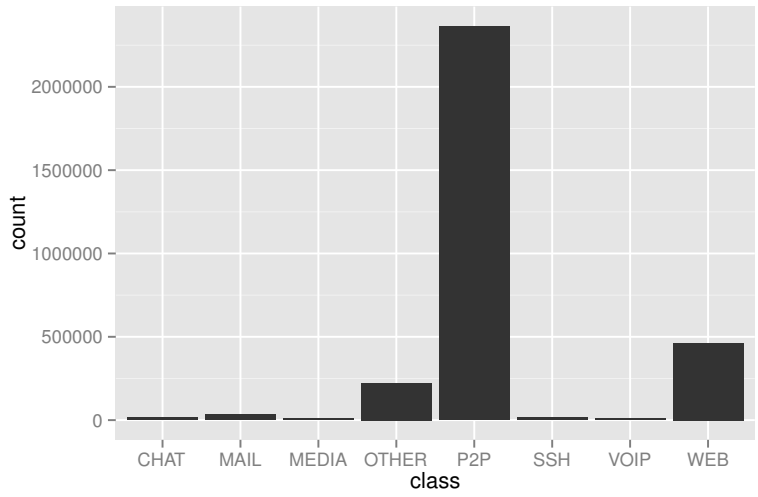


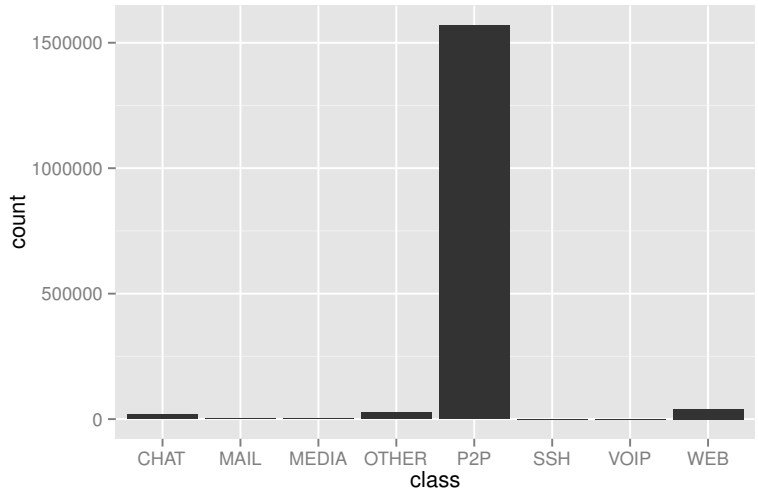
Figure 4.6.: Partition of the aggregated queueing delays into LOW, MID, HIG ranges, on a per application basis. In the y-axis we represent the fraction of queueing delay of a certain class that fall into one of the three ranges.

LOW



-

MID



-

HIG

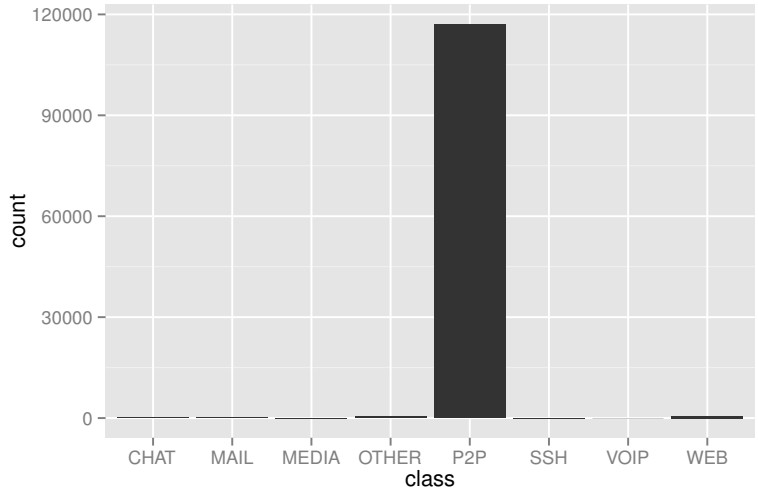
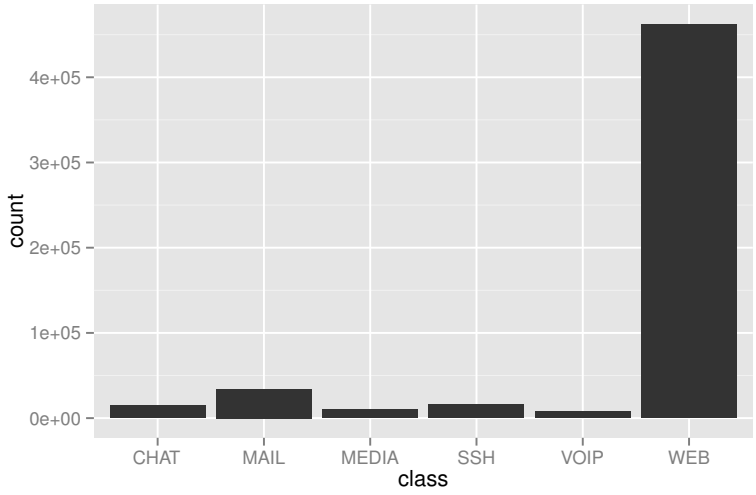


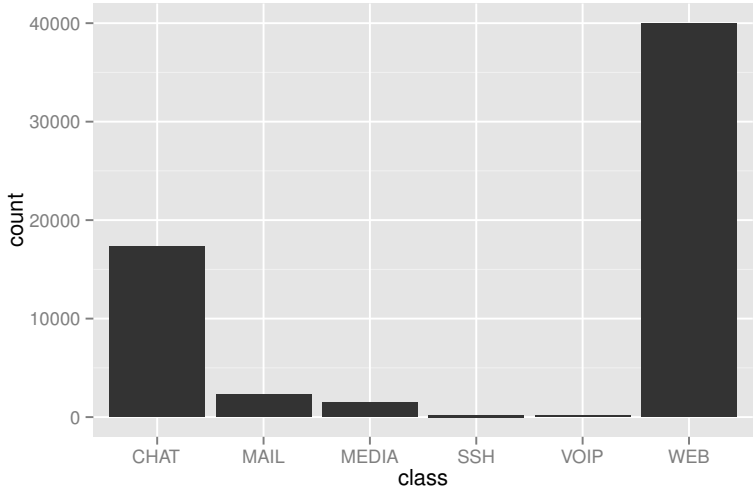
Figure 4.7.: Frequency plots for different aggregated queueing delay ranges

LOW



-

MID



-

HIG

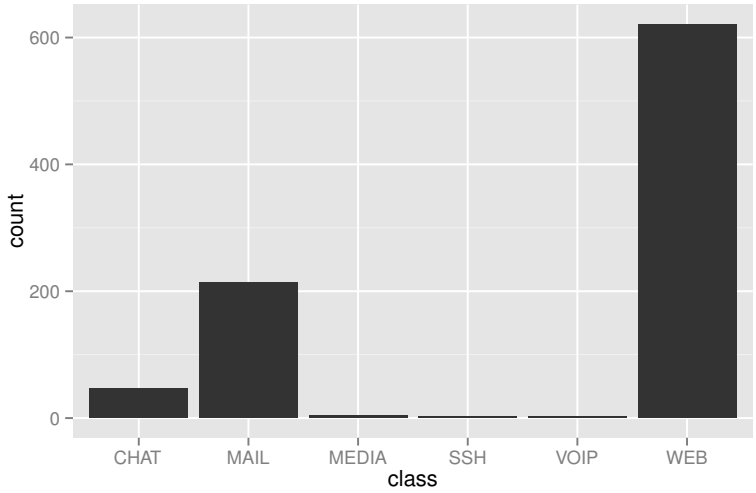


Figure 4.8.: Frequency plots for different aggregated queueing delay ranges (without OTHER and P2P)

5. Conclusion

5.1. Summary

This work proposes a methodology to passively observe Internet bufferbloat. Though preliminary, this work already conveys several useful insights (e.g., ranging from guidelines on settings of monitoring tools to avoid bufferbloat underestimation (sec. 3.3.2), to a per-application assessment of likely QoE impact, to rule inference, etc.) and especially allows to pinpoint its root cause.

Moreover, the monitoring tool that we realized in Tstat can be already used to apply our methodology to other real traffic traces and compare the results with the ones proposed by other authors (see sec. 1.5).

5.2. Future work

As future work, our methodology could be deeply analyzed and refined.

The impact of the delayed acknowledgement (see page18) could be further investigated. The quality of the results that our methodology produces could be systematically analyzed in different scenarios, e.g. experimenting with wireless devices that may lead to high variability of the data-to-acknowledgement time that is not correlated with buffering. The validation of sec. 3.3 could be extended to a testbed in which the devices are connected in the real Internet (rather than in a LAN).

While in this work we adopt a minimalistic approach, each delay sample carries additional information beyond the application label, concerning the amount of its activity during the window (i.e., number of packets, bytes and number of valid data-ack pairs). As part of our future work, we plan to leverage this information to refine the quality of our inference (e.g., aggregated queueing delays could be weighted

by the number of packets; the number of valid data-ack pair, raw or normalized over the number of packets in the window, could be used as an indication of the confidence of each aggregated queueing delay; the instantaneous sending rate over the last few windows correlates with the queuing delay of the subsequent windows, possibly assisting root cause analysis; etc.).

We also plan to apply our measurement to a measurement campaign with more representative real traffic traces (rather than the 8-hour trace we have worked on).

More importantly, we plan to deploy the modified version of Tstat on operational networks, in order to gather more statistically significant results from both spatial (over several ISPs and traffic mixes) and temporal perspectives (comparing the older dataset which we focus on in this paper with on- line monitoring results).

Acknowledgments

This work has been carried out during Andrea Araldo's intership at LINCS http://www.lincs.fr.
The research leading to these results has received funding from the European Union
under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

A. Short paper to ACM CoNEXT 2013

The work that we present here in an extended form led up to the submission of the following short paper to ACM CoNEXT 2013, the 9th International Conference on emerging Networking EXperiments and Technologies[1].

[1] http://conferences.sigcomm.org/co-next/2013/

Bufferbloat: passive inference and root cause analysis

Andrea Araldo,
Universita degli Studi di Catania, Italy
andrea.araldo@unict.it

Dario Rossi
Telecom ParisTech, Paris, France
dario.rossi@enst.fr

ABSTRACT

In this work, we propose a methodology to gauge the extent of queuing delay (aka bufferbloat) in the Internet, based on purely passive measurement of bidirectional TCP traffic. Leveraging on Deep Packet Inspection (DPI) and behavioral classification, we next show a per-application breakdown of the queuing delay in ISP networks, that assists in binding the queueing delay to the performance perceived by the users of that application. Finally, we report preliminary results to further correlate the amount of queuing delay seen by each host with the set of active applications for that host during small time windows, to find the root cause of bufferbloat.

1. INTRODUCTION

Despite the steady growth of link capacity, Internet performance may still be laggy in the early 2010. Already in 1996, a famous post [9] pointed out that delay, more than bandwidth, was an important metric for user perception. As confirmed by the recent resonance of the "bufferbloat" buzzword [16], this may still hold today. Shortly, excessive buffer delays (measured in seconds) are *possible* in today's Internet due to the combination of loss-based TCP congestion control coupled to excessive buffer sizes (e.g., in user AP and modem routers, end-host software stack and network interfaces) in front of slow access links.

While, through controlled testbed and experiments, it is clear that high latency can hamper user QoE of Web [5, 24], multi-media [17, 8] or even peer-2-peer [25] users, it is unclear how high is queuing latency in practice. Indeed, while it is known that queuing delays can potentially reach a few seconds [20] under load stress, and while these delays have been anedoctically observed [16], it however is unclear how *common* they are for end-users daily experience – which is precisely the goal of this paper.

Summarizing, our contributions are as follows: first, we propose a passive TCP queuing delay estimation methodology and make our open source implementation, based on Tstat[14], available to the community. Second, we quantify the typical bufferbloat seen by different applications, by leveraging on Tstat's Deep Packet Inspection (DPI) and behavioral classification capabilities. Third, we propose a methodology to find the root cause of bufferbloat, and report preliminary results that further correlate the amount of queuing delay seen by each host with the set of applications active on that host during small time windows.

2. BACKGROUND

Delay measurement over the Internet are definitively not a new subject – indeed, over 20 years passed since seminal work such as [6]. Yet, despite the Internet steady evolution, performance problems resurface that are actually *worsened* by technology advances: indeed, Moore law not only increased the memory size, but also the amount of packets standing in modem buffers. As such, recent effort has focused on explicitly measuring, among other performance indicators, the latency and queuing delay experienced by end-users. With few exceptions [3, 15, 10, 11], most related effort [20, 12, 19, 24, 4, 5, 1, 21] employs active measurement techniques.

Active vs Passive measurement. Our methodology relies on passive measurement that, due to its unobtrusiveness and realism of the user traffic, is the ideal candidate to answer our questions – i.e., what delay users observe in their daily activities? under what applications? Hence, due to lack of space, we invite the reader to a companion technical report [7] for a thorough overview and comparison of related work on active measurement. Here we limitedly observe that, though active methodologies are potentially simpler and more accurate, at the same time they exhibit some weak points of worth pointing out. First, many work measures latency under controlled load [20, 12, 19, 24, 4, 5], which tends to give *maximum* (rather than *typical*) bufferbloat. Second, periodic measures of latency have generally very coarse granularity (with the exception of [24], where samples are still spaced out by 6 seconds in the best case), so that bufferbloat can go unnoticed. Third, and most important, active techniques miss one crucial ingredient: the knowledge of the traffic that caused the bufferbloat. In other words, while [24] points out queuing delay to vary between 800ms

and 10s depending on modem make and model, active methods are unable to gauge how often users actually see bufferbloats in excess of 1s, or to pinpoint the application that caused it – which are precisely our goals.

Another interesting tradeoff between active vs passive measurements concerns their representativeness, in terms of number of users and networks observed. Scale of active measurements can be as small as O(10)–O(10^3) users in O(1) networks (resp. [19] and [24]), growing up to O(10^5) users in O(10^3) networks [20, 4] (though it is worth stressing that the user base is possibly gathered over several months [20]). Passive methodologies allow to observe a larger number of users at any given time. At the same time, as observation of both forward and backward paths is necessary, they can hardly be applied in the network core (due to routing asymmetry, only about 2%[15] flows are bidirectional) so that, generally vantage points are sited at the network edge (e.g., near to a DSLAM as in this work) and represents about O(10^4) users in O(1) networks.

Passive measurement. Some recent work tackled the problem of passive measurement of queuing delay [15, 3, 11, 10]. In our previous work [11, 10], we propose methodologies to infer *remote* host queues exploiting transport layer information available in packet headers, for both uTP [10] (the new protocol proposed by BitTorrent as TCP replacement for data swarming) and TCP [10] (using RFC1323 TimeStamp option[18]). Contrarily to [11, 10], in this work we focus on the *local* host queue (since we have full knowledge of *all* traffic on that host) and adopt a more general methodology, of which we outline some important differences. First, notice that while uTP timestamps allow to precisely gauge the remote queue (even in presence of cross-traffic toward unseen hosts) observations are limited in both space (to hosts that are running BitTorrent) and time (precisely when they run it). This constrains measurement campaign [10] on the one hand (a disadvantage shared with [4]), and possibly induces a biased view of the Internet bufferbloat on the other hand (since BitTorrent is a data-intensive application) – problems that this work instead avoids. Second, contrarily to [11], we avoid relying on timestamps carried in packet headers for TCP, increasing the reach of the methodology (despite growth of TCP TimeStamp option usage, this still account for modest 5%-30% at our vantage points).

Closer to our work is [3] that, using `bro`, employs a similar methodology to ours, relying on TCP data/acknowledgement pairs, using trace timestamps as oppostie to TCP Timestamp option and takes care of rejecting RTT samples from retransmitted segments (though the methodology is not validated in a testbed, see Sec. 3.1 for potential issues). Since our dataset significanly differ from [3] (in terms of US vs EU location, FTTH vs ADSL access, duration, etc.), we cannot attempt a *direct* comparison of bufferbloat results – but still point out that, as in [3, 10] we find delays above 1 sec to be rare in practice. Additionally, while [3] equally counts all RTT samples (i.e., equally weighting *packets* of the same TCP burst, so that users transferring large volumes are over-represented) we give a more unbiased view (equally weighting each *second* of all hosts). Finally, a more important difference is that while [3] measures TCP queuing delay blindly across all applications, we instead give a fine-grained per-application view – binding queuing delay to user QoE and explaining its root cause.

Finally, [15] focuses on a memory-efficient bufferbloat measurement methodology, by keeping approximate TCP state in a probabilistic data structure (that can fit the cache of current MIPS and ARM processors used in home DSL gateways), at the price of a minimal accuracy loss (error is less than 10 ms in 99% of the cases, compared to `tcptrace` as a baseline). However, as the focus of [15] is on the relative accuracy of the methodology, it reports differences with respect to the baseline rather than absolute bufferbloat measurement. Our approach is instead complementary and, assuming a high performance dedicated measurement box (i.e., no memory constraint), implements a methodology to accurately gauge current Internet bufferbloat (incidentally, building over `tcptrace`, of which Tstat is an evolution).

3. QUEUING DELAY INFERENCE

3.1 Methodology

We infer queuing delay of local hosts simply as depicted in Fig. 1. For each acknowledgement packet reporting a valid RTT sample (i.e., no reordered, nor retransmitted as in [3]), we compute the difference between the time $t_{tx,i+1}$ the acknowledgement packet is observed and the time $t_{rx,i}$ the data packet corresponding to the sequence number was observed. Packets are timestamped at the measurement point via Endace DAG cards, so that timestamp is reliable. With respect to Fig. 1, assume the local host queue A contains at time $t_{rx,i}$ packets directed toward hosts B, C and D. Neglecting for the sake of simplicity delayed-acknowledgement timers (that are by the way small compared to the bufferbloat magnitudo reported in [20]), as soon as data is received at about $t_{rx,i}$ at A, the TCP receiver issues an acknowledgement that will be serviced after the already queued data segments. The monitor can then estimate the queuing delay q_{i+1} incurred by the (i+1)-th acknowledgement segment as the difference between the current samples $t_{tx,i+1} - t_{rx,i}$ and the minimum among the previously observed samples of that flow (that represents the propagation delay component and, as the monitor is close to the end-user, is by the way expected to be small).

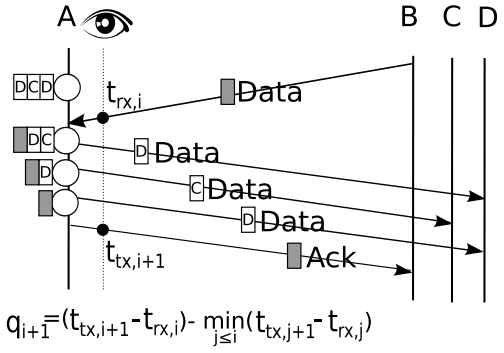


Figure 1: Synopsis of our passive methodology

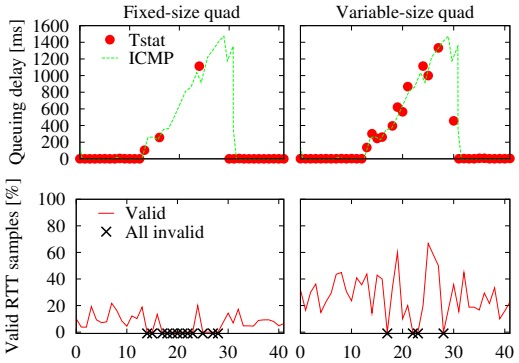


Figure 2: Testbed validation of the methodology

At low level, `tcptrace` keeps sequence numbers in a circular data structure named `quad` (i.e., after the "quadrants" the structure is divided into, to speed-up lookup). In case the `quad` has a fixed size, it may happen that, if the number of outstanding segments grow larger than the `quad` size, then sequence number are overwritten – so that ack cannot be paired with data and RTT samples are lost. We show an occurrence of this problem in Fig. 2, where we configured two values of the quad size – one is fixed (to a purposely small value) and the other is variable and can grow arbitrarily large to avoid overwriting outstanding sequence numbers (notice that variable size is handled with linked lists, so that in this validation phase we take precisely the opposite direction to [15], as we do not want to compromise accuracy).

In a local testbed, we send bidirectional TCP traffic between LAN hosts, mimicking Fig. 1. Host B sends rate-limited data (at application-layer) to A, whose acks are used to passively infer queuing delay samples at A (denoted Tstat in the picture). The uplink of host A is limited to 1Mbps with a token bucket shaper and, after about 10sec, we start a backlogged transfer between A and C, causing congestion to build up. To validate passive inference, we send ICMP echo requests to B at 1Hz, and compare average Tstat queuing delay during 1 sec long windows. To gather the impact of low-level Tstat settings, we record the packet trace and repeat the analysis with small and fixed (left) or variable (right) size `quad`. It can be seen that, in case the structure is under-dimensioned, sequence numbers are possibly overwritten when large amounts of packets are queued, so that possibly all samples during a 1 sec long window are lost. As large buffer sizes are common [13, 24], it follows that careful settings of monitoring tools are needed to avoid underestimating bufferbloat[1].

[1]Notice that, at 1Mbps, 1500ms worth of queuing delay correspond to more than 100 queued packets, which exceeds the default `quad`=100 size in Tstat – hence, at higher capacities, even lower delay could be underestimated for lower `quad` sizes.

3.2 Experimental results

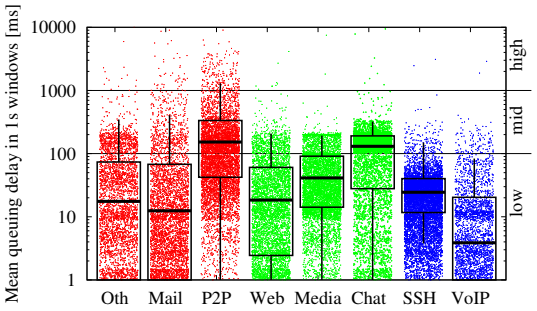
We now report experimental results of the inference. As our focus on this paper is to build a solid methodology, rather than providing full-blown measurement campaign, for the time being we analyze offline traces gathered in a single ISP network during FP7 NapaWine [22] project, when P2P was still fairly popular (we consider a 8 hr-long period starting at 14:00, gathered during 2009).

We argue that in order to give statistics that are useful from the user perspective, we need to batch[2] consecutive samples (e.g., belonging to the same TCP burst) into windows whose duration relates with the timescale typical of user dynamics. For the network under observation, we consider each internal IP as a single[3] host. For each host, we collect delay samples for each active flow, corresponding to the average queuing delay over short time windows of 1 second duration, as estimated by valid data-ack pairs of each flow. Overall, our processing gathers about 10^7 individual per-flow samples.

Each delay sample carries an application label, obtained through Tstat Deep Packet Inspection (DPI) and behavioral classification capabilities. Though Tstat is capable of fine-grained classification of different applications [14], we cluster similar applications into few classes depending on the service they offer (namely, Mail, Web, Multimedia, P2P, SSH, VoIP, Chat and other uncategorized applications). To the best of our knowledge, this

[2]In the specific case of uTP, we have already shown [10] that queuing delay statistics can be biased (precisely, queuing delay is underestimated) in case each packet is counted as a sample (as opposite to windows of equal duration).

[3]This is known to be simplistic as, due to the penetration of NAT devices, the same IP is shared by multiple hosts (50% of the cases [23]), that are possibly active at the same time (10% of the cases). Yet we point out that this simplification has no impact for our methodology, since these potentially multiple hosts share the same access bottleneck link



	Oth	Mail	P2P	Web	Media	Chat	SSH	VoIP
low	89.7	93.2	58.4	91.9	86.8	45.7	98.6	97.8
mid	10.2	6.2	38.7	8.0	**13.2**	**54.1**	**1.4**	**2.2**
high	0.1	0.6	2.9	**0.1**	0.0	**0.1**	0.0	0.0

Figure 3: Breakdown of queuing delay per application. Plots report jittered density maps and (5,25,50,75,95)-th percentiles. Table report the percentage of per-application samples falling into either of the three delay regions (boldface values highlight possible QoE degradation).

work is the first to report a detailed per-application view of the Internet queuing delay – that depends on the traffic mix and user behavior of each household. We present our results in Fig. 3 where we depict a jittered density map of queuing delay samples (y-axis) for different application classes (x-axis), along with boxplots reporting the quartiles (and 5th, 95th percentiles). Applications are ordered, left to right, in increasing order of delay sensitivity. It can be seen that, for most applications the 75% of windows experience less than 100ms worth of queuing. The only exceptions are consituted by, rather unsurprisingly, P2P applications and, somehow more surprisingly, Chat applications, with *median* delays exceeding 100ms.

Before dwelving the root cause of the above observations, let us dig further its implication. Due to studies assessing the impact of delay on the QoE of several applications such as Web [5, 24], multimedia [17, 8] or P2P [25, 10] applications, we can easily map a QoS metric such the queuing delay, into an coarse indication of QoE for the end-user. Based on the above work, we set two thresholds at 100 ms and 1 second, such that: (i) performance of interactive multimedia (e.g., VoIP, video-conference and live-streaming) or data-oriented applications (e.g., remote terminal or cloud editing of text documents) significantly degrades when the first threshold is crossed [17, 8]; (ii) performance of mildly-interactive application (e.g., Web, chat, etc.) significantly degrades when the second threshold is crossed [5, 24]; (iii) additionally, while bulk transfers (e.g., P2P, long TCP connections) are elastic in nature, it has been shown that also TCP performance degrades [16] in presence of excessive buffering (i.e., control becomes unstable due to absence/delay of feedback information) and furthermore queuing delay affect control plane of P2P applications [25, 10] – so that even these applications performance start to degrade when the second threshold is crossed.

Additionally, Fig. 3 tabulates the percentage of 1 sec windows for each application that fall into either of the three delay regions, where we use boldface values to highlight possible QoE degradation. It follows that, in practice, bufferbloat impact on QoE appears to be modest. A limited 0.1% of Web and Chat sessions may be impacted by significant delay, and 2.2% (1.4%) of VoIP (remote terminal) sessions may be impacted by moderate delay. P2P clearly stand out, raising the odds to induce high delays (2.9%) followed by SMTP (0.6%), though with likely minor impact for the end-users.

4. ROOT CAUSE ANALYSIS

4.1 Methodology

As we have full knowledge of the traffic generated and received by hosts, we can correlate the queuing delay samples with the traffic active on the hosts. As before, we limitedly consider the upstream traffic direction, and divide time in 1 second long slots. For all active flow of each hosts, we log the application label and its average queueing delay during that window. Notice that in Fig. 3 we independently consider all delay samples: in other words, the delay seen by packets of an application flow can be induced by another flows active on the same time in the same host (or household). In this section, we leverage standard data mining techniques to, if possible, pinpoint the root cause of the observed delay.

In our analysis, we face known, yet non trivial problems. First, techniques such as frequent itemset mining or rule-mining are known not to be scalable, and would be hard to apply to our full dataset. Second, these techniques are affected by class imbalance, that we need to cope with. At this stage, we make the problem tractable by performing stratified sampling, initially including 8,000 random samples per-application (64,000 samples overall). Let us denote by (t,h,a,q) a queuing delay sample q gathered on host h during the time window t and corresponding to an application a. Evidently, we need to ensure that our population includes *all* other samples corresponding to applications that were active on h during t. We therefore complement the initial population, to achieve an overall population of 107,825 samples. To guarantee the statistical relevance of our analysis, we further verify that statistics of the sampled population correspond to those reported in Fig. 3.

For each host h and time window t pair, we next compute the queuing delay average as seen by all applications during t. As rule-inference techniques are known to be largely ineffective on continuous variables, and in reason of the previous QoE considerations, we quantize

queuing delay in a *low*, *medium* and *high* score.

We also aggregate all application labels, and denote with ✓ (or ++) applications that have generated respectively one (or more) flows during t. For the sake of illustration, consider 2 applications are active at host h during $t = 0$. One is a Chat application, consisting of a single flow, the other is a P2P application with 3 active flows during $t = 0$: we thus aggregate the application labels as (Chat✓, P2P++). Notice that this criterion again implements an aggressive quantization into classes: intuitively, we argue that there is more information in knowing coarsely whether there was a single or more flows, rather than distinguishing precisely the exact number of flows. Following this spirit, we additionally encode the previous example with purely binary indicators as in (Chat✓, P2P✓).

We yet have to perform a final step in order to facilitate the analysis of the root cause analysis. Namely, we group together windows, irrespectively of the host, having the same overall amount of active flows. This is necessary to simplify interpretation of the support of the inferred rules. Intuitively, as windows with x active flows for an host will not have the same frequency as windows with y concurrently active flows in our dataset, grouping window by the number of active flows allows to let overally infrequent (i.e., with small support) but still interesting (i.e., with high confidence) rules to emerge.

Table 1: Rule inference with Apriori

Delay	Applications						Supp.	Conf.	
	Num	Other	Mail	P2P	Web	Chat			(ref)
High	1		✓				11	99	(a)
	1			++			10	96	(b)
	1				✓		10	64	(c)
	2			++	✓		20	98	(d)
	2	✓		++			28	87	(e)
	2	✓			✓		23	80	(f)
	2	++		++			14	61	(g)
	2			++	++		19	54	(h)
	3	✓		++	✓		13	99	(i)
Medium	1			✓			35	76	(j)
	1				++		44	53	(k)
	2			✓		✓	10	85	(l)
	2	++		✓			11	88	(m)
	2			✓	✓		19	85	(n)
	2			✓	++		10	77	(o)
	2	✓		✓			15	76	(p)
	2	++			++		16	72	(q)
	2				✓	✓	10	51	(r)
	2	✓			++		20	50	(s)

4.2 Experimental results

Results of Apriori [2] are reported in Tab. 1 for both medium and high delays, where for each rule (essentially, a group of contemporary application labels inducing a given delay) we report confidence and support (percentage). Rules are reported, top to bottom, for incresing size (i.e., number of application label in the rule). For each rule size, rules are sorted, top to bottom, for decreasing confidence. Already from this small dataset, several interesting observation can be made.

First, notice that SMTP can cause bufferbloat: this is summarized by rule (a), which can be expected due to the use of persistent TCP connections, used to send multiple messages, each of which encapsulates possibly large e-mail bodies due to the use of MIME to encode attachments of various kind (e.g., pictures, music, archives, etc.). Notice that mail application does not otherwise frequently generate delay, confirming the intuition that small textual messages have no impact on the QoE of other applications.

Second, high delays are also due to (b) multiple concurrent P2P flows or (c) single persistent HTTP connections. Interestingly, the specular case in which (j) a P2P application has a single active flow, or (k) multiple HTTP connections are active in parallel, the delay generally remains bound to the 100 ms-1 sec range. The intuition is thus that users often browse the Web in parallel to P2P transfers. However, packets of short lived TCP Web connections often pile up behind bursts of TCP packets due to multiple P2P connections. An implicit confirmation of this intuition comes from the fact that BitTorrent recently[4] redesigned the data transfer, by introducing the delay-sensitive uTP protocol, aimed at bounding the queuing delay to no more than a configurable target parameter (set to 100 ms by default). Reason why a single Web connection should generate larger uplink queuing delay is instead less obvious, and need further investigation.

Third, various combinations of P2P, Web and uncategorized traffic (Other) jointly concurr in creating bufferbloat (we notice that uncategorized traffic possibly include P2P applications for which Tstat has no valid classification signature). We again notice that rule (d) combining a single Web and multiple P2P flows, has a higher confidence that rule (h). Moreover, as the support of (h) is included in the support of (d), it follows that including multiple Web connections weakens the rule significance – as already observed comparing (c) against (k), the delay seem to be inversely correlated with the number of Web connections.

Finally, we see that Chat sessions often happens in

[4]Our dataset is before BitTorrent uTP became popular in late 2010.

parallel with (l) P2P or (r) Web traffic – with the former especially cause of delay suffered by Chat application, in reason of the relative confidence of rules (a) and (j) with respect to (l). Hence, user behavior of Chatting in parallel to other applications is the cause of relative high frequency of medium delays (54% fall in the 100 ms-1 sec range) seen early in Fig. 3. From this observation, we can also conjecture that Chat users likely do not consider these levels of delay harmful for QoE. Indeed, since all other applications have lower delay statistics, this may follow from the fact that user naturally disactivate data-intensive background applications (e.g., P2P) and do not perform other activities (e.g., Web browsing) while delay-sensitive applications are ongoing (e.g., VoIP calls).

5. DISCUSSION

This paper propose a methodology to passively observe Internet bufferbloat. Though preliminary, this work already convey several useful insights (e.g., ranging from guidelines on settings of monitoring tools to avoid bufferbloat underestimation, to a per-application assessment of likely QoE impact, to rule inference, etc.) and especially allows to pinpoint its root cause.

While in this work we adopt a minimalistic approach, each delay sample carries additional information beyond the application label, concerning the amount of its activity during the window (i.e., number of packets, bytes and number of valid data-ack pairs). As part of our future work, we plan to leverage this information to refine the quality of our inference (e.g., samples could be weighted by the number of packets; the number of valid data-ack pair, raw or normalized over the number of packets in the window, could be used as an indication of the confidence of each sample; the istantaneous sending rate over the last few windows correlates with the queuing delay of the subsequent windows, possibly assisting root cause analysis; etc.).

More importantly, as part of our future work, we plan to deploy the modified version of Tstat on operational networks, in order to gather more statistically significant results from both spatial (over several ISPs and traffic mixes) and temporal perspectives (comparing the older dataset which we focus on in this paper with online monitoring results).

Acknowledgements

This work has been carried out during Andrea Araldo intership at LINCS `http://www.lincs.fr`. The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

6. REFERENCES

[1] http://internetcensus2012.bitbucket.org/.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[3] M. Allman. Comments on bufferbloat. *SIGCOMM Comput. Commun. Rev.*, 43(1), Jan 2012.

[4] Z. Bischof, J. Otto, M. Sánchez, J. Rula, D. Choffnes, and F. Bustamante. Crowdsourcing ISP characterization to the network edge. In *ACM SIGCOMM Workshop on Measurements Up the STack (W-MUST'11)*, 2011.

[5] Z. S. Bischof, J. S. Otto, and F. E. Bustamante. Up, down and around the stack: ISP characterization from network intensive applications. In *ACM SIGCOMM Workshop on Measurements Up the STack (W-MUST'12)*, 2012.

[6] J.-C. Bolot. End-to-end packet delay and loss behavior in the internet. *ACM SIGCOMM Computer Communication Review*, 23(4):289–298, 1993.

[7] P. Casoria, D. Rossi, J. Auge, M.-O. Buob, T. Friedman, and A. Pescape. Distributed active measurement of internet queueing delays. Technical report, Telecom ParisTech, 2013.

[8] K.-T. Chen, C.-Y. Huang, P. Huang, and C.-L. Lei. Quantifying skype user satisfaction. 36(4):399–410, 2006.

[9] S. Cheshire. It's the latency, stupid! `http://rescomp.stanford.edu/~cheshire/rants/Latency.html`, 1996.

[10] C. Chirichella and D. Rossi. To the moon and back: are internet bufferbloat delays really that large. In *IEEE INFOCOM Workshop on Traffic Measurement and Analysis (TMA)*, 2013.

[11] C. Chirichella, D. Rossi, C. Testa, T. Friedman, and A. Pescape. Remotely gauging upstream bufferbloat delays. In *PAM*, 2013.

[12] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: a browser-based network measurement platform. In *ACM IMC*, 2012.

[13] L. DiCioccio, R. Teixeira, M. Mayl, and C. Kreibich. Probe and Pray: Using UPnP for Home Network Measurements. In *PAM*, 2012.

[14] A. Finamore, M. Mellia, M. Meo, M. Munafo, and D. Rossi. Experiences of internet traffic monitoring with tstat. *IEEE Network Magazine*, May 2011.

[15] S. Gangam, J. Chandrashekar, I. Cunha, and J. Kurose. Estimating TCP latency approximately with passive measurements. In *PAM*, 2013.

[16] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Communications of the ACM*, 55(1):57–65, 2012.

[17] O. Holfeld, E. Pujol, F. Ciucu, A. Feldmann, and P. Barford. BufferBloat: how relevant? a QoE perspective on buffer sizing. Technical report, 2012.

[18] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. IETF RFC 1323, 1992.

[19] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3G/4G networks. In *ACM IMC*, 2012.

[20] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *ACM IMC*, 2010.

[21] D. Leonard and D. Loguinov. Demystifying service discovery: implementing an internet-wide scanner. In *ACM IMC*, 2010.

[22] E. Leonardi, M. Mellia, A. Horvath, L. Muscariello, S. Niccolini, and D. Rossi. Building a cooperative P2P-TV application over a Wise Network: the approach of the European FP-7 STREP NAPA-WINE. *IEEE Communication Magazine*, 64(6), April 2008.

[23] G. Maier, F. Schneider, and A. Feldmann. Nat usage in residential broadband networks. In *PAM*, 2011.

[24] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband internet performance: a view from the gateway. In *ACM SIGCOMM*, 2011.

[25] C. Testa and D. Rossi. The impact of uTP on BitTorrent completion time. In *IEEE P2P*, 2011.

Bibliography

[AKM04] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4): 281–292, August 2004, http://doi.acm.org/10.1145/1030194.1015499.

[All12] Mark Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 43(1): 30–37, 2012.

[APS+99] Mark Allman, Vern Paxson, Wright Stevens, et al. Tcp congestion control. 1999.

[AS+94] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, volume 1215, pages 487–499, 1994.

[BCC+98] Bob Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, et al. Recommendations on queue management and congestion avoidance in the internet. 1998.

[BOB12] Zachary S. Bischof, John S. Otto, and Fabián E. Bustamante. Up, down and around the stack: Isp characterization from network intensive applications. *SIGCOMM Comput. Commun. Rev.*, 42(4): 515–520, September 2012, http://doi.acm.org/10.1145/2377677.2377778.

[BOS+11] Zachary S. Bischof, John S. Otto, Mario A. Sánchez, John P. Rula, David R. Choffnes, and Fabián E. Bustamante. Crowdsourcing isp characterization to the network edge. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*, W-MUST '11, pages 61–66, New York, NY, USA, 2011. ACM, http://doi.acm.org/10.1145/2018602.2018617.

[Che96] Stuart Cheshire. It's the latency, stupid, 1996.

[CHHL06] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. Quantifying skype user satisfaction. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 399–410. ACM, 2006.

[Chi12] C. Chirichella. A methodology to gauge the extent of Bufferbloat in the Internet: LEDBAT vs TCP . Tesi di laurea specialistica, Universita' degli Studi di Napoli Federico II, 2011/12.

[CK10] B. Nechaev V. Paxson C. Kreibich, N. Weaver. Netalyzr: Illuminating the edge network. *Internet Measurement Conference (IMC)*, 2010.

[CR] Chiara Chirichella and Dario Rossi. To the moon and back: are internet bufferbloat delays really that large?

[CRT+13] C Chirichella, D Rossi, C Testa, T Friedman, and Antonio Pescapé. Remotely gauging upstream bufferbloat delays. In *Passive and Active Measurement*, pages 250–252. Springer, 2013.

[DHGS07] Marcel Dischinger, Andreas Haeberlen, Krishna P Gummadi, and Stefan Saroiu. Characterizing residential broadband networks. In *Internet Measurement Conference: Proceedings of the 7 th ACM SIGCOMM conference on Internet measurement*, volume 24, pages 43–56, 2007.

[DTMK12] Lucas DiCioccio, Renata Teixeira, Martin May, and Christian Kreibich. Probe and pray: Using upnp for home network measurements. In *Passive and Active Measurement*, pages 96–105. Springer, 2012.

[FMM+11] A. Finamore, M. Mellia, M. Meo, M.M. Munafo, and D. Rossi. Experiences of internet traffic monitoring with tstat. *Network, IEEE*, 25(3): 8–14, 2011.

[GCCK13] Sriharsha Gangam, Jaideep Chandrashekar, Ítalo Cunha, and Jim Kurose. Estimating tcp latency approximately with passive measurements. In *Passive and Active Measurement*, pages 83–93. Springer, 2013.

[GN11] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11): 40, 2011.

[Han06] M. Handley. Why the internet only just works. *BT Technology Journal*, 24(3): 119–129, 2006, http://dx.doi.org/10.1007/s10550-006-0084-z.

[HJR12] K. Lee H. Jiang, Y. Wang and I. Rhee. Tackling bufferbloat in 3g/4g networks. *ACM IMC, 2012*, 2012.

[HPC+12] Oliver Hohlfeld, Enric Pujol, Florin Ciucu, Anja Feldmann, and Paul Barford. Bufferbloat: How relevant? a qoe perspective on buffer sizing. 2012.

[IG96] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3): 299–314, 1996.

[Jac88] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[Jac98] Van Jacobson. Notes on using red for queue management and congestion avoidance. *talk at NANOG*, 13, 1998.

[Jan09] Philipp K Janert. *Gnuplot in action: understanding data with graphs*. Manning Publications Co., 2009.

[JBB92] Van Jacobson, Robert Braden, and David Borman. Tcp extensions for high performance. 1992.

[JLM94] V Jacobson, C Leres, and S McCanne. libpcap, lawrence berkeley laboratory, berkeley, ca. *Initial public release June*, 1994.

[JWL04] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2490–2501. IEEE, 2004.

[KP87] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM SIGCOMM Computer Communication Review*, 17(5): 2–7, 1987.

[LMH+08] Emilio Leonardi, Marco Mellia, A Horvath, L Muscariello, S Niccolini, and D Rossi. Building a cooperative p2p-tv application over a wise network: the approach of the european fp-7 strep napa-wine. *Communications Magazine, IEEE*, 46(4): 20–22, 2008.

[MDF12] R. Teixeira C. Kreibich M. Allman N. Weaver M. Dhawan, J. Samuel and V. Paxson. Fathom. a browser-based network measurement platform. *ACM IMC*, 2012.

[MMM06] Marco Mellia, Michela Meo, and Luca Muscariello. Tcp anomalies: identification and analysis. In *Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements*, pages 113–126. Springer, 2006.

[MSF11] Gregor Maier, Fabian Schneider, and Anja Feldmann. Nat usage in residential broadband networks. In *Passive and Active Measurement*, pages 32–41. Springer, 2011.

[PA00] Vern Paxson and Mark Allman. Computing tcp's retransmission timer. Technical report, RFc 2988, November, 2000.

[Pax99] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23): 2435–2463, 1999.

[SdDF+11] Srikanth Sundaresan, Walter de Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband internet performance: A view from the gateway. *Proc. ACM SIGCOMM, Toronto, Ontario, Canada*, 2011.

[SHIK10] Stanislav Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (ledbat). *draft-ietf-ledbat-congestion-04. txt*, 2010.

[TQD+05] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[TR11] Claudio Testa and Dario Rossi. On the impact of utp on bittorrent completion time. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 314–317. IEEE, 2011.