Accuracy Evaluation of Application-Level Performance Measurements[1]

Katarzyna Wac, Patrik Arlos, Markus Fiedler, Stefan Chevul, Lennart Isaksson, Richard Bults

Abstract. In many cases, application-level measurements can be the only way for an application to evaluate and adapt to the performance offered by the underlying networks. Applications perceive heterogeneous networking environments spanning over multiple administrative domains as ”black boxes” being inaccessible for lower-level measurement instrumentation. However, application-level measurements can be inaccurate and differ significantly from the lower-level ones, amongst others due to the influence of the protocol stacks. In this paper we quantify and discuss such differences using the Distributed Passive Measurement Infrastructure (DPMI), with Measurement Points (MPs) instrumented with DAG 3.5E cards for the reference link-level measurements. We shed light on various impacts on timestamp accuracy of application-level measurements. Moreover, we quantify the accuracy of generating traffic with constant inter-packettimes (IPTs). The latter is essential for an accurate emulation of application-level streaming traffic and thus for obtaining realistic end-to-end performance measurements.

1 Introduction

Application-level measurements are, in most cases, the only way for an application to evaluate the performance offered by the underlying heterogeneous networking environment, spanning over multiple administrative domains. In this context it is hard, for not saying impossible, to insert probes along the application’s end-to-end communication path. An application sees the network as a ”black-box” transport system accessible via TCP or UDP, and may use measurements to adapt to the perceived network conditions. A complication rises from the fact that the observed application-level behavior can be different from the behavior observed at the link-level due to the influence of, for example the sender and receiver hosts’ protocol stacks on the packets’ generation and acquisition processes, the Operating Systems (OSs), the systems clock influencing application timestamps accuracy, or even the application itself. However, if a host is not overloaded and the protocol stack is properly implemented, parameters like an inter-packet-time (IPT) for a certain load, ideally should be the same at both application- and link-level. Assuming this,

[1] This paper has been presented and included in the proceedsings of the 3rd EURO-NGI Conference on Next Generation Internet Networks Design and Engineering for Heterogeneity (NGI07), 21-23 May 2007, Trondheim, Norway

we evaluate the accuracy of application-level measurements with comparison to the *reference* link-level ones.

We study two different application-level active measurements- based tools - a tool developed in the MobiHealth project [1] (referred further to as the *tool A*) and in the Personal Information in Intelligent Transport systems through Seamless communications and Autonomous decisions (PIITSA) project [2], [3] (referred further to as the *tool B*). The tools collect timestamps, which are then used to calculate other metrics. The tool A calculates an application-level Protocol Data Unit (PDU) one-way-transit time (OWTT), while the tool B calculates an application-level throughput and data loss ratio. The major differences between the tools are that the tool A uses TCP, while B uses UDP, A is implemented in Java, while B in C#, and they differ in functions used for generating PDUs at a steady rate and for PDU timestamping (Section III). These differences influence the accuracy of the generated IPTs and of the timestamps (which in turn influence the one-way-transit-time (OWTT) and throughput calculations). We evaluate the tools using the Distributed Passive Measurement Infrastructure (DPMI) [4], with Measurement Points (MPs)s equipped with DAG 3.5E [5] cards.

There are many specialized application-level active measurement tools, however, as indicated by Michaut et al. [6], there are no guidelines for measurements' quality assurance and in most cases, tools are simply not evaluated. Most of the tools are intended to operate under Unix/Linux OS, hence their authors assume a μs resolution for timestamps [7]. Some authors theoretically calculate influence of timestamps accuracy and hosts' synchronization method on the results obtained with their tools [8]. For example, the *pathchar* authors indicate that their tool uses network delays estimations (with 5% accuracy) to calculate bandwidth, and these are not the main source of the tool's measurement errors [9]. Similarly Ali et al. [10], used differential calculus to estimate the errors introduced by timestamps inaccuracies in end-to end bandwidth measurements by four different tools: *pathload*, *pathChirp*, *spruce* and *IGI*. The *spruce* tool estimates bandwidth with 23% error, while IGI with 29% error for timestamp accuracy of the order of 10 μs. Moreover, Ali et al. estimate a system clock access delay of 1 to 6 μs using *gettimeofday()* function, and protocol stack delay of 5 to 65 μs per packet.

The most practical, i.e. measurements-based evaluation of selected application-level tools like *ping* and *J-OWAMP* [11] has been conducted indicating measurements inaccuracies under Windows OS [12]. Our study follows this approach.

Section 2 of this paper provides setup for evaluation of tools, presented then in Section 3. Section 4 explains analysis method for the collected data and Section 5 presents measurements results. Finally, Section 6 concludes on our findings.

2 Setup

Figure 1 presents the physical setup used in the experiments. The evaluated tools A and B were installed on the source/destination hosts: *H1* and *H2*. The tools were instructed to generate over the period of 25 minutes link-layer PDUs of 576 Bytes, corresponding to 526 Bytes for TCP and 536 Bytes for UDP, with the nominal IPT

(IPT_{nom}) of (a) 125 ms and (b) 90 ms. This *load* corresponds to a mobile healthcare application scenario when sending eight channels of patient's vital signs data from a mobile patient to the application server in a hospital [1].

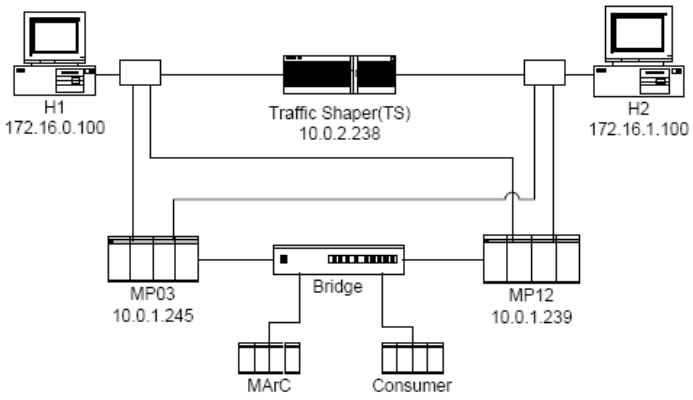

Fig. 1. Experimental setup.

The hosts H1 and H2 were identical with respect to hardware (Dell Optiplex), with 667 MHz Pentium-3 CPU, 256MB RAM and built-in 100Mb/s full-duplex Ethernet cards. The OS was Microsoft Windows XP with Service Pack 2 (updated on 26.06.2006) with Java v1.5.0. We have chosen this OS because in a mobile application scenario, a mobile device is likely to be a Windows OS-based device, serving as a user's extension of mostly Windows OS-based desktop.

Both hosts ran Tardis v1.6 software [13] to synchronize their clocks to a local Network Time Protocol (NTP) server (*time.bth.se*). Access to this server was obtained via the TS, which also acted as a *traffic shaper* on the traffic sent *between* the hosts (thus not influencing the NTP traffic). The TS introduced constant delay of 180 ms on traffic sent between H1 and H2. *Wiretaps* [14] tapped the traffic between H1 and TS, and between H2 and TS, and sent it to the Measurement Points (MP), MP03 and MP12, that are a part of a DPMI setup [4]. Both MPs were equipped with Endace DAG 3.5E cards [5], synchronized using a TDS-2 connected to an Acutime Global Positioning System (GPS) antenna, yielding a timestamp accuracy of 60 ns [12]. The MPs and wiretaps were wired such that a MP monitored traffic in one direction on both links, i.e., MP03 captured traffic from H1 to TS and from TS to H2, while MP12 from H2 to TS and TS to H1. This way, for each direction, the link-level PDU's timestamps were synchronized at the MP. A dedicated Gigabit Ethernet (GE) switch connected the MPs, a *Measurement Area Controller* (MArC) and a *Consumer*. The MArC managed the MPs. The Consumer analyzed (non-intrusively) the streamed measurement trace from the MPs in order to derive the setup parameters (e.g. link utilization), to catch possible problems or data discrepancies, as well as to store measurement data into files after each measurement session.

3 Tools

Both tools have separate sender and receiver programs. A sender has a configurable load generator with respect to a PDU length, IPTs and number of PDUs. Both sender and receiver have dedicated measurement function for collecting PDUs timestamps at the ingress (sender) and egress (receiver) points of the network. The receiver at both tools is implemented such that it continuously attempts to receive data from the network. During the measurement session, the tools collect the measurements data in-memory (static vector), which is then dumped into a file after the session, this to minimize the influence of the data collection process on the ongoing measurements.

Tool A has been developed and used in the European (FP6) MobiHealth project to evaluate the end-to-end performance of a heterogeneous networking environment (with e.g. 3G as wireless technologies) supporting time-critical mobile healthcare services [15]. The tool calculates (offline) per- PDU OWTT, its variation and an application-level throughput. The critical requirement for this tool is to have sender and receiver's time clocks precisely synchronized (by means of e.g. NTP) in order to get usable PDU timestamps.

The tool was implemented in Java v.1.3.0, to comply with e.g. IBM J9 JVM used on mobile devices. The tool uses TCP as the transport system interface, with explicit data *flush* after a PDU is send to the socket. The tool uses Java *Thread.sleep()* functions to send PDUs at a given IPTs. The sender, based on the required IPT, calculates and then tries to keep the required number of PDUs per time-window of 1 second. That implies change of the IPT of the last PDU send in the window, such that the sum of all IPTs in the window equals to 1s. The tool uses *System.currentTimeMillis()* to obtain a PDU timestamp just after each sent or received PDU. During a measurement session, tool's Text User Interface is disabled and the Java thread has the priority set to *HIGH* in the OS.

Tool B has been developed and applied in the Swedish PIITSA project to evaluate performance of networks supporting mobile services for intelligent data transport systems [2], [3]. The tool calculates (offline) observation-window-based application-level throughput statistics (at sender and receiver separately) and data losses. There is no explicit requirement for the nodes' clocks to be time-synchronized.

Figure 2 presents the load generation and measurement function at the sender side. At the start of PDUs stream generation, sender acquires a reference timestamp (T0) and uses it further as an absolute value to calculate all upcoming PDUs' transmission times $T2 = T0 + IPT \times PDU_{seqnr}$. Each created PDU, containing its sequence number (for data loss ratio calculation), is timestamped (T1) and enters an active waiting IPT loop, which is released only if measured $T2b \geq T2$. In this way, if a previous IPT was not fulfilled, e.g. due to PDU send function delays, the waiting time for an actual PDU will be shorter, to fulfill its required IPT. Finally, the sender take timestamps (T3,T4) before and after each PDU send function. Ideally, T2==T3, hence timestamp T3 is used to derive PDUs' IPTs from tool's measurements traces.

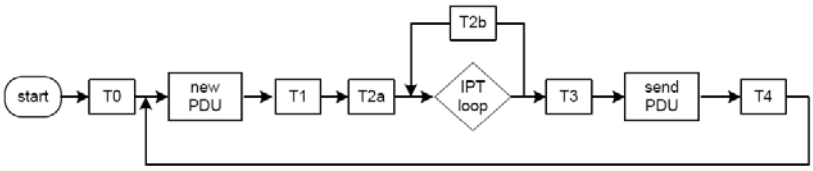

Fig. 2. Tool B load generator and measurement function at the sender.

The tool has been developed in C#, using the .NET framework. The tool uses UDP as the transport system interface. To get a μs timestamp resolution, the tool uses *performance counters*, the kernel32.dll QueryPerformanceCounter and QueryPerformanceFrequency functions, in conjunction with the system time. A timestamp is calculated by dividing the counter value by the frequency value. The latter one is evaluated only at the measurements initialization phase, and assumed to be constant during a measurements session. Moreover, during a session, tool's Graphical User Interface is disabled and tool's priority is set to *realtime* in the OS.

4 Measurement Data Analysis

The experiments were done by having host H1 (the sender) and H2 (the receiver) running the tools (one at a time) and collecting the application-level traces, while the DPMI was collecting the link-level traces. The measurement data was analyzed offline using Matlab 7.

Given $T_{x,y}(k)$ as a PDU timestamp obtained at party x (sender (s) or receiver (r)) at level y (application (a) or link (l) level) for a PDU k in (1...n − 1), we calculated an IPT for a PDU pair (k, k + 1) as

$$IPT_{x,y}(k, k + 1) = T_{x,y}(k + 1) - T_{x,y}(k).$$

Moreover, we calculated a *timestamp accuracy*, TΔ [12] as observed at the receiver's application-level over the whole measurements session, assuming the receiver's link-level IPT values as *reference* values. Therefore, we obtained

$$T\Delta = |\max(\varepsilon_{k,k+1})| + |\min(\varepsilon_{k,k+1})| \text{ for k in } (1...n - 1)$$

given a timestamp accuracy error for a PDU pair as

$$\varepsilon_{k,k+1} = IPT_{a,r}(k, k + 1) - IPT_{l,r}(k, k + 1).$$

The TΔ value represents an extreme timestamp accuracy error as it combines all error sources, e.g. these due to the clock resolution, skew and drift, due to the clock access time or due to the clock synchronization events and their associated errors.

5 Results

In the following subsections we present the calculated (from the measurements) IPTs at the application- and link-level for the different IPTs. As we saw from the calculations, the direction, in which PDUs were send (H1 to H2 or H2 to H1) had no influence on obtained measurements, therefore we present only the results in

one direction (H1 to H2). In Figure 3 (tool A) and Figure 4 (tool B) we plot the IPTs as a function of PDU sequence number for a nominal IPT_{nom}=125 ms. Tables I and II presents the corresponding statistics together with the estimated TΔ values. Both tool A and B display an average IPT close to IPT_{nom}, but the standard deviations differ significantly. Tool A displays a standard deviation of 5.11 ms, as confirmed in Figure 3. Obviously, the sender IPT alternates between 120 and 130 ms, which probably stems from the 10 ms timestamp resolution of Java *System.currentTimeMillis()* function used under Windows OS. The corresponding sender's link-level behavior is more stable as seen from a smaller standard deviation of 1.13 ms. The latter raises slightly on the way through the network due to the impact of the traffic shaper. At the receiver, the observed IPT again alternates between 120 and 130 ms (cf. Figure 3) with a similar effect on the standard deviation. One interesting observation at the receiver's application level is the minimum IPT of 20 ms (Tables I and II) occurring in the trace just after a large IPT of 230 ms. Probably we face a PDU delayed in the receiver's stack, resulting from the scheduling of the Java thread in the OS. Moreover, there is one sample of a maximum IPT of 241 ms after which occurs an IPT of expected 120 ms. A closer look at the sender's trace reveals that this value corresponds to the extraordinary sender IPT of 231 ms.

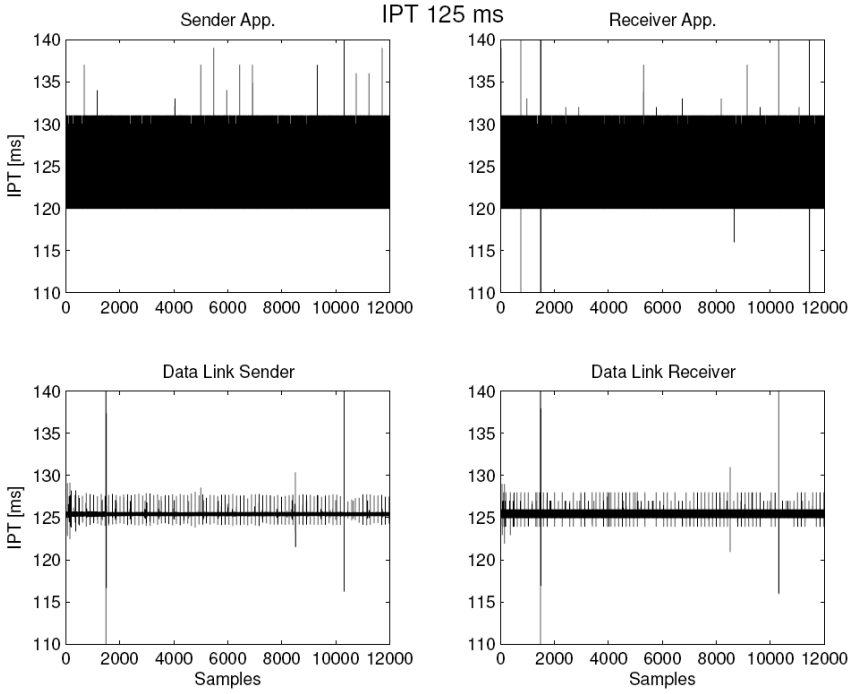

Fig. 3. Tool A: measured IPT at sender and receiver for IPT_{nom}=125 ms.

Turning our attention to tool B, cf. Figure 4, we observe that (1) the application and link-level graphs are more or less identical at sender and receiver side, and (2) the sender and receiver application-level traces are also very similar. These

observations are underlined by Table II, from which we see that the statistical parameters (mean, median and standard deviation) and the extreme values are almost identical, regardless of the host or level. The average of minimal and maximal IPT matches IPT_{nom}; this reveals the effort of the sender tool to ”keep up” in case of extraordinary delays in *PDU send* function. Looking closer at the sender’s application-level trace, we observe this effect to exhibit some periodic behavior, with a first large IPT value around sequence number of 1000, then around 5000, followed by another deviation around sequence number 9000. This behavior needs further investigation. For this measurement, the priority of the receiver process was set to *realtime* in the OS.

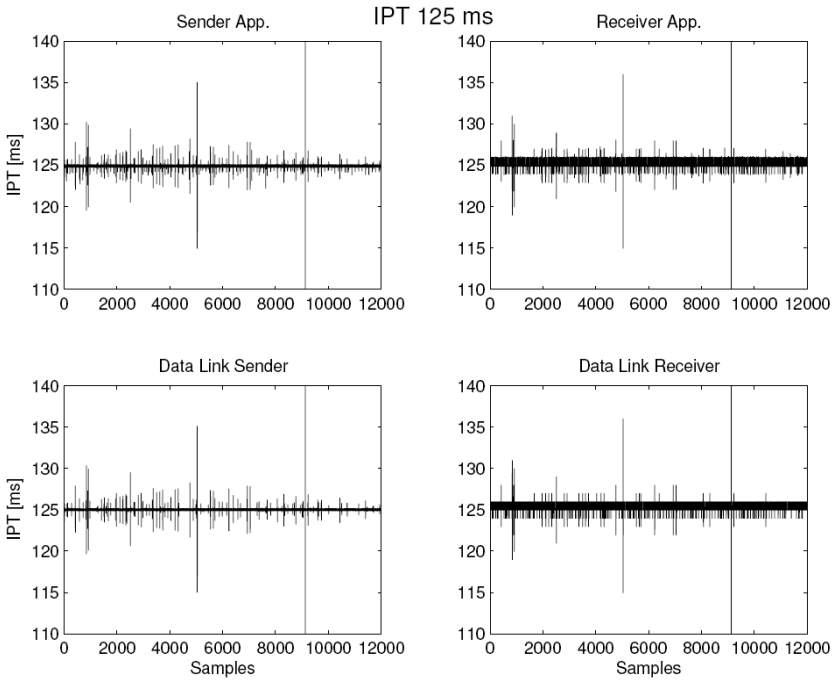

Fig. 4. Tool B: measured IPT at sender and receiver for IPT_{nom}=125 ms

Table 1. Tool A: IPT statistics at sender and receiver for IPT_{nom}=125 ms

Parameter	Sender		Receiver	
[ms]	Appl.	Link	Link	Appl.
min	120.00	109.88	109.96	20.00
max	231.00	236.76	236.92	241.00
mean	125.43	125.43	125.43	125.43
median	130.00	125.41	124.96	130.00
std.dev.	5.11	1.13	1.23	5.33
T_{Δ}[ms]	N/A		209.00	

Table 2. Tool B: IPT statistics at sender and receiver for IPT_{nom}=125 ms

Parameter	Sender		Receiver	
[ms]	Appl.	Link	Link	Appl.
min	65.86	65.97	65.98	65.97
max	184.78	184.86	184.94	184.94
mean	124.91	125.00	125.00	125.00
median	124.93	125.00	124.96	124.95
std.dev.	0.86	0.81	0.85	0.85
T_Δ[ms]	N/A		3.45	

Table 3. Tool A: IPT statistics at sender and receiver for IPT_{nom}=90 ms

Parameter	Sender		Receiver	
[ms]	Appl.	Link	Link	Appl.
min	90.00	82.17	81.97	0.00
max	200.00	205.12	204.92	210.00
mean	91.05	91.05	91.05	91.05
median	90.00	90.13	89.98	90.00
std.dev.	3.02	2.98	3.00	4.01
T_Δ[ms]	N/A		201.00	

Table 4. Tool B: IPT statistics at sender and receiver for IPT_{nom}=90 ms

Parameter	Sender		Receiver	
[ms]	Appl.	Link	Link	Appl.
min	55.98	69.84	69.98	69.96
max	109.96	110.07	109.96	109.91
mean	89.91	90.00	90.00	90.00
median	89.93	90.00	89.97	89.97
std.dev.	0.42	0.32	0.39	0.67
T_Δ[ms]	N/A		30.75	

If we now look at the estimated TΔ's for IPT_{nom}=125 ms, we see that the tool A has a TΔ of 209 ms and tool B a TΔ of 3.45 ms. Obviously, tool A does not only suffer from the inherit 10 ms timestamp resolution, but also from PDU queuing at the receiver, stemming from the Java-typical thread scheduling. In addition to this, the tool uses the system time which is conditioned by the Tardis time synchronization tool. In contrast, the tool B uses C# and its built-in performance counters, which in theory gives a timestamp resolution of 1.5 ns (1/667MHz). Furthermore, the tool assumes a stationary behavior of the CPU during a measurement, and synchronizes only at the measurement session startup.

In Tables III and IV we show the statistics for measurements with IPT_{nom}=90ms. The fact that tool A is parameterized by choosing the number of PDUs to be sent (here 11 PDUs per second equivalent to IPT~90.9 ms) explains the slight deviation of the average from the IPT_{nom}. However, we also see that tool A observed a minimum IPT of zero, which indicates that at least two PDUs were received within the same time interval corresponding to the timestamp resolution.

Looking at the statistics for tool B, we see that the tool's sender reports a quite small minimum IPT of 55.98 ms, cf. the IPT_{nom}=90ms. The other minimal values on link-level and on the receiver's application level of around 70 ms and the

maximal values of ~110 ms average to IPT_{nom}. As opposed to the IPT_{nom}=125 ms - case, the priority of the receiver process was set to *normal* in the OS, resulting in significant rise in IPT standard deviation from link to the application level. Looking at the TΔ estimations for IPT_{nom}=90 ms, the tool A maintains estimate close to the estimate for IPT_{nom}=125 ms, while the tool B has a estimate that is ten times larger than the first one. We explain this behavior by the fact that during this test, tool's priority at the receiver was set to *normal* in the OS, while it was set *realtime* in the first test. As we observe based on TΔ, the tool's priority in the OS has a major impact on the PDU-receiving process, being in competition with the other processes.

6 Conclusions

In this paper we have analyzed and quantified the accuracy of the application-level active-measurements using passive link-level measurements as the *reference* measurements. We have observed that the application-level behavior can be different from the behavior observed at the link-level due to the influence of the OS at the sender and receiver as well as their hosts' protocol stacks on the packets' generation and acquisition processes, and the system clock resolution under the Windows OS influencing application timestamps accuracy.

From the results we conclude that the tool A, using Java thread sleeping functions to generate PDUs at a given IPT_{nom}, indeed generates PDUs at the required IPT, as seen from the link-level data, but the application-level timestamps suffer from the *System.currentTimeMillis()* method resolution of 10 ms under Windows OS. We also notice that the tool cannot be configured to generate a constant IPT, but instead the tool tries to generate a certain number of PDUs within a 1 second interval. Moreover, comparing the link-level with the application-level statistics at the sender and receiver, we see that the mean IPT is the same while the extreme vales differ, which could be caused by e.g. the OS thread scheduling mechanism for Java threads or PDU delays along the protocol stack (especially at the receiver). We also see that for a given load, the receiver has problems keeping up, and we observe a IPT of zero value, meaning that some PDU was delayed and delivered together with the next one. All these observed behaviors are reflected in the tool's poor timestamp accuracy of around 200 ms translating into the fact, that for the tools A, practically only the *second* part of the timestamps can be considered to be true.

The tool B, implemented in C# using active waiting loop for PDU generation and *performance counters* for PDU timestamping, can generate PDUs with a constant IPT, and timestamp them quite accurately. Under our test conditions, the sender and receiver application-level statistics are quite identical to the link-level data. However, the sender needs further evaluation of the possible periodic behavior causing extreme values. We should also note that the tool is not synchronized and it does not adjust for the CPU frequency oscillations, which however does not seem to affect the tool accuracy in the observed interval. When looking at the estimated TΔ for the tool B, values increase up to ten times depending on if tests were executed with tool's priority set to *realtime* in the OS.

We shall notice that the best-case TΔ value for tool A is in the order of 3.45 ms and indicate high accuracy of this tool.

The main conclusion on our findings is that pure theoretical *estimations* of measurements accuracy may not sufficiently reflect the real measurements quality. Instead, such estimations may lead to too optimistic conclusions. Our strong recommendation is that one should always quantitatively and precisely evaluate the accuracy of an application-level measurement tool in its operational state. This should be done before using the results obtained with the tool by for adapting the application to the behavior of underlying networks as observed by the tool. Regarding the behavior of the underlying networks, application-level measurements can be misleading and not at all reflecting this behavior, neither an impact of the networks on the application. As we have proved in this paper, application-level measurements may rather reflect a random and unpredictable behavior of the end hosts' systems and their protocol stacks. Measurement errors and observation discrepancies may result in the (unnecessary) application adaptation, which in critical cases may result in application crash (e.g. due to buffers overflow). This should be avoided at all cost, especially in mission-critical mobile services like for example those in the healthcare domain.

References

[1] A. van Halteren, R. Bults, K. Wac, and et al. Mobile patient monitoring: The MobiHealth system. *Journal on Information Technology in Healthcare*, 2(5), 2004.

[2] M. Fiedler, L. Isaksson, S. Chevul, J. Karlsson, and P. Lindberg. Measurement and analysis of application-perceived throughput via mobile links, Tutorial. In *Performance modeling and evaluation of Heterogeneous Networks*, UK, 2005.

[3] M. Fiedler, S. Chevul, L. Isaksson, P. Lindberg, and J. Karlsson. Generic Communication Requirements of ITS-Related Mobile Services as Basis for Seamless Communications. In *1st EuroNGI Conference on Traffic Engineering*, Italy, 2005.

[4] P. Arlos, M. Fiedler, and A. Nilsson. A Distributed Passive Measurement Infrastructure. In *Passive and Active Measurement Workshop (PAM05)*, US, 2005.

[5] Endace Measurement Systems. URL: http://www.endace.com (verified Sept. 2006).

[6] F. Michaut and F. Lepage. Application-oriented network metrology: Metrics and active measurement tools. *IEEE Comms. Surveys and Tutorials*, 7(2):2–24, 2005.

[7] B. Mah. Pchar: A Tool for Measuring Internet Path Characteristics. URL: http://www.kitchenlab.org/www/bmah/Software/pchar/ (verified Aug. 2006).

[8] P. Beyssac. Bing: A point-to-point bandwidth measurement tool based on PING, 1995. URL: http://spengler.econ.duke.edu/ ferizs/bing.txt (verified Aug. 2006).

[9] B. Downey. Using pathchar to estimate Internet link characteristics. In *Conference on Applications, technologies, architectures, and protocols for computer communication*, US, 1999.

[10] A. Ali, F. Michaut, and F. Lepage. End-to-End Available Bandwidth Measurement Tools: A Comparative Evaluation of Performances. In *4th Intl Workshop on Internet Performance, Simulation, Monitoring and Measurement*, Austria, 2006.

[11] H. Veiga, T. Pinho, and Oliveira. J-OWAMP: Java implementation of the One-Way Active Measurement Protocol. URL: www.av.it.pt/jowamp (verified Sept. 2006).

[12] P. Arlos. *On the Quality of Computer Network Measurements*. PhD thesis, Blekinge Institute of Technology, Karlskrona, Sweden, 05:2005.

[13] Tardis 2000 software. URL: http://www.kaska.demon.co.uk (verified Aug. 2006).

[14] Finisar Corporation. URL: http://www.finisar.com (verified Sept. 2006).

[15] R. Bults, K. Wac, A. van Halteren, and et al. Goodput Analysis of 3G wireless networks supporting m-health services. In *8th International Conference on Telecommunications (ConTEL05)*, Croatia, 2005.