# Network Interfaces Flying over IP Networks

João Paulo Barraca, Alexandre Brito, Rui L. Aguiar

*Abstract*— **We propose a novel method to export both interface control and data planes across different hosts, effectively enabling hardware specific control of network interfaces over the Internet, or from a host to its virtualized guests. Our solution is a major step towards distributed environments of heterogeneous communication systems, particularly relevant in the scope of custom system composition, remote development and testing, and is especially relevant when considering embedded or geographically constrained devices. Results obtained by our prototype implementation validate the effectiveness of the solution. We present a preliminary characterization of its impact, when considering traffic generation applications, when applied over an IEEE 802.11g communication medium.**

*Index Terms*—**Network Interface, Remote Operation, Experimentation**

## I. Introduction

Network devices are the core components of current communication infrastructures. Regardless of considering clients, routers, servers - or any other equipment connected to a network - all have at least one network device. This piece of equipment makes it possible to interconnect different devices, conveying the digital information to a specific medium such as ATM, Ethernet or DSL.

Network devices operate in an almost transparent manner and provide two specific planes for applications and operating systems: a data plane and a control plane. Applications provide bit sequences to be sent, (data plane) and the interfaces will send information correctly to the next hop, adapting the medium independent information to the actual medium in use. Some medium specific headers are added, and bits are encoded and transmitted according to the proper medium access protocol. After traveling across the transmission medium, packets arrive at the next network device, and are decoded. Some headers are removed, and the remaining bit stream is sent upwards in the network stack. A control plane, parallel to the data plane allows managing network devices, specifying operational parameters such as medium bitrate, frequency, packet size and even security profiles. Also, this control plane can provide statistics about the communication medium in the form of Signal-to-Noise-Ratio (SNR), amount of packets sent and received, amount of collisions, Bit Error Rate (BER). One particular aspect about these two planes is that (with the exception of vendor specific extensions or performance counters), their scope is local to the current equipment. That is: it is not trivial to inspect a remote network interface and interact with it. Even if this is not much of a problem for most uses, there are an increasingly large number of situations where such a feature would be desirable.

Our work targets those situations where a network device control and data planes should be accessed remotely. In particular where it is required to send and receive packets, or control operational parameters of a specific, non-local network device. One of such scenario is the development for embedded devices. Applications for such devices are usually developed inside sandboxes on common workstations. Sandboxes emulate a particular environment and are used to improve development and debugging times, due to higher availability of computational, memory and storage resources as well as debugging tools. One limitation of sandboxes is that hardware is not mimicked, and access to low level functions of network devices (or other hardware) are not always possible, or presents limitations. The same problem arises when virtualizing hosts using hypervisors. The hardware composing the host is not always exported to its guests, as only some classes of devices are supported (e.g. USB). While 802.3 devices are popular in these environments, where a virtual 802.3 interface is provided to guests, wireless interfaces such as 802.11 or 802.16 (as well as many others) aren't always available.

In our work, we aim to provide a solution for this issue by enabling hosts to export both data and control planes of selected network interfaces over an IP based network. With our solution is becomes possible to: have real 802.11 interfaces inside virtualized hosts, independent of the bus they use (e.g. USB, PCI, or PCMCIA); to compose hosts with network interfaces, which are not collocated; and to better trace networking problems.

This paper is organized as follows: In section II we present other work relevant for the field related to our solution; In section III we describe the architecture of the FlyingInterface solution, which we developed to address problems with 802.11 devices; Section IV focus on presenting and discussing the results obtained by evaluating our solution. Finally we present the conclusions obtained by developing and evaluating this work.

## II. Related Work

The solution we devise presents some similarities with the

This work was supported in part by the European Union Framework Program 7 under grant agreement n°224263-OneLab2.

João Paulo Barraca is with the Instituto de Telecomunicações – Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; e-mail: jpbarraca@av.it.pt.

Alexandre Brito is with the Instituto de Telecomunicações – Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; e-mail: abrito@av.it.pt.

Rui L. Aguiar is with the Instituto de Telecomunicações – Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; e-mail: ruilaa@ua.pt.

concept of the Network Block Device [6]. NBD aims at exporting block devices (e.g. hard disks) and we similarly aim at exporting network interfaces. Also, EtherPuppet [10] provides basic state export functionalities but with several limitations: control plane not kept in sync; limited to 802.3 network devices; limited to well known attributes such as MTU and IPv4 address.

Virtualization solutions for testbeds such as in [7] propose the use of virtual machines in order to increase scalability, and most importantly, to re-use hardware. CoreLab assumes that virtual machines have network interfaces that are connected to TAP interfaces. Furthermore, standard DNAT mechanisms allow applications to communicate with the outside world. In this case, no MAC or PHY information is sent to nodes, and packets are routed normally. UML-Wifi as presented in [8], proposes the addition of a new bus (netbus) to User Mode Linux so that wireless state can be sent to a guest running UML. This proposal enables a guest to have access to some aspects of the underlying wireless card (registers, memory, interrupts). The focus of UML-Wifi is to emulate the entire wireless card as a virtual hardware device. While powerful, this solution is restricted to cards supporting hostap and to guests running under UML on hosts with wireless hardware. The Orbit testbed [2] uses the UML-Wifi solution over the network like shown in [3]; this is a good example of the need for virtual wireless interfaces. Virtualization solutions such as VMWare and Virtualbox, designed to virtualize generic hosts, already provide the means to expose existing devices to guests (connected through USB). However, only devices directly attached to the host can be exported, and more importantly, exporting a device to a guest will make it unavailable to the host.

In comparison with these approaches, our solution is completely generic and flexible: it is usable in any Linux system, with either real or virtualized resources, attached to the system using any communication interface (e.g. USB, PCI), supporting any private set of IOCTLs, and located on the same machine or over different hosts connected by an IP Network.

## III. Flying Interfaces Approach

The FlyingInterface concept was developed after we identified some aspects in wireless network research methods, which could be improved: a) create research environments capable of combining the communication capabilities of an equipment (often a small device) to the computational, and storage capabilities of a more powerful host; b) compose testbeds for research and demonstration purposes, while overcoming the dependency on geographic location; c) simplify the development and debugging of network centric solutions when accessed by remote teams; d) simplify fast prototyping of solutions targeted to embedded systems (e.g. Android platform), on more powerful development systems; e) improve overall security of remote access to wireless R&D infrastructures.

Our solution fulfills these goals by providing a tun/tap interface on the local host, which clones the state of a real (or virtual) interface, located on a remote host (see Figure 1).

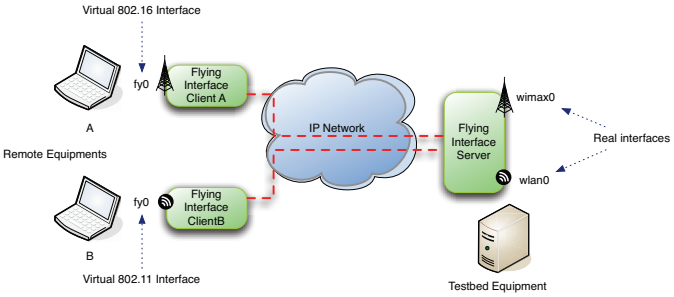


**Figure 1 - Overview of remote interface export using Flying Interfaces**

Both the Control and Data planes of the networking infrastructure are exported over the IP network. Packets sent to the local interface are injected in the destination, and packets received by the real interface are tunneled back to its clone. IOCTL invocations on the local interface are sent to the real interface, and the resulting response is returned. Our solution doesn't need to be aware of the actual call being invoked. To the local host, the tap interface will seem to support the same capabilities as the real device interface, such as Wireless Extensions (and private IOCTLs) or any other type of IOCTL. The local tap interface will also have the same media type, and provide the same traffic counters and statuses flags. In all aspects the local interface will appear as much as possible as a copy of the real (remote) device.

The FlyingInterface device driver (see Figure 2) is based on the TUN/TAP driver that was extended to support remote replication of the device state. Therefore, all support for TUNelling (which only allows to tunnel application above IP, and not arbitrary protocols) was removed and only the TAP mechanisms were kept.

During the remaining of this document we assume that the Client application runs in the local host, while the Server runs in a remote host.

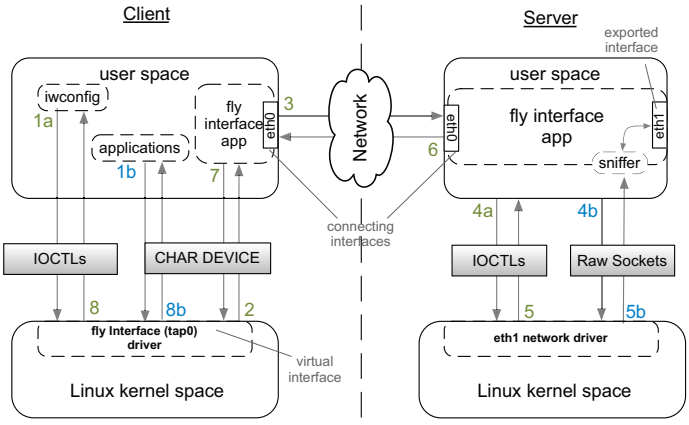


**Figure 2 - FlyingInterface architecture, and the modules composing the solution: Network Device module, Client and Server.**

This kernel driver creates a cloned interface that accepts all types of IOCTLs and packets. Also, it creates a char device to communicate with the userland client. The client application

interacts with this char device and connects to a server application, which is in a remote machine and has the actual communication hardware. The Server both injects packets into the real device, listen to packets arriving to the device, and issues IOCTLs.

Packets sent to the *flyX* network interface (the clone) are delivered to the Client application, running locally. The FlyingInterface Client application reads these packets from the kernel char device and forwards them to the FlyingInterface Server application, located at the remote host, and attached to the real network interface. Every message sent to the network is encapsulated accordingly with an extra header, which distinguishes control messages (IOCTLs), data packets and internal commands.

When IOCTLs requests are generated at the local host, by invoking applications such as *ifconfig* or *iwconfig*, they are intercepted by the developed kernel module (1a), encapsulated with the corresponding header (2), and sent to the Server application through the IP network (3) to be executed remotely (4b). The response follows the reverse path and is delivered to the calling application. If the IOCTL is not supported by the remote interface, returns additional data, or produces an error, this information is returned as-is to the calling application, completely emulating the remote interface. If communication is lost and the response to the IOCTL doesn't arrive after a configurable delay (default is 30s) the module will return a generic error to the calling application.

Data packets flows between client and server in a similar manner as IOCTLs, i.e., read (1b), sent to the Client application and encapsulated with the corresponding header (2), sent to Server (3) and injected at the remote interface (4b) through a RAW Socket.

Finally, Command messages are used for establishing and configuring the connection between FlyingInterface Client and Server. One example is when connecting a client to a Server. In this case, the Server application immediately sends the real interface stats and addresses (MAC, IPv4 and/or IPv6) to the Client. Also, if the server is shutting down, the client application must also quit and release the cloned interface.

Having a cloned network interface implies to virtually have two network interfaces sharing the same IP and MAC address (the remote and the local). When packets arrive at the remove interface, the result will be duplicated responses (e.g. ICMP Echo Request). One response is issued by the network stack at the remote host, the other response from the network stack at the local host. Since the Server application installs a RAW socket, actually sniffing packets at the real interface, every packet that crosses the networking stack is replicated but not halted (due to performance issues, the real interface is only set to promiscuous mode if required by the user). To solve this problem the FlyingInterface Server application also installs a proper set of iptables rules to silently drop all packets at the OUTPUT and INPUT hooks of the remote interface. A RAW socket assures that the Server application is able to maintain its communication: injected packets are sent after the OUTPUT hook (and not blocked) while incoming packets are captured before the INPUT hook (also not blocked).

Another expected issue is related to the network MTU and the resulting packet fragmentation. Since some packets can be built with a size, which is close to (or equal to) the network MTU, and these same packets need to be encapsulated and sent to the destination, the total size of messages exchanged by Client and Server can exceed the MTU. Therefore they require reassembling before being injected to the real interface (on the remote host) or being dispatched to the FlyingInterface driver (on the local host). FlyingInterface Client and Server applications implement reassembling of received packets to the max size of 65326 bytes (64KB minus 4 bytes from the size of the message header) effectively supporting Jumbo Frames to the limitations imposed by the underlying hardware.

The described architecture and mechanisms aims to create a fast and reliable remote communication approach without any special configuration requirements, in an almost transparent way to the user. We aim to create a solution with minimal overhead and with small impact on the (network) measured relevant metrics.

### *A. Limitations*

Our solution provides a transparent and efficient way of exporting network interfaces over an IP network. Still, there are some inherent limitations either associated to the concept, or caused by the way the actual prototype was devised.

Related to our prototype implementation, we identify one major drawback: compatibility between local and remote hosts. IOCTL calls impose limitations on the underlying hardware architecture of client and server: both hosts must run a Linux Kernel, and both must run similar versions of it. Moreover, because we are agnostic on the parameters of the IOCTL calls, and due to memory alignments, variable integer size, and endianness, the same hardware architecture must also be used at both endpoints. Data packets are not affected by this limitation and are forwarded freely by our solution.

Conceptually, all solutions aiming to export devices (like NBD for block devices) suffer from the fact that additional delay, jitter and eventually loss, is introduced to the system. Such changes result from the fact that several network hops may separate an application from the real network interface. Eliminating this is difficult or even impossible (in non simulated environments), although can be partially minimized. We minimize loss by using a TCP connection between Client and Server. The drawback is that TCP congestion avoidance algorithm will shape jitter and latency. Minimizing such effects can be achieved by guaranteeing that there is enough bandwidth, and minimal latency between Server and Client. In particular, the network medium interconnecting Server and Client must have higher bandwidth and lower latency than the destination medium, so that accurate results can be obtained. This limitation is not relevant if the purpose of the use is other than accurate measurement of behavior of the destination network medium.

## IV. Results and Discussion

The AMazING testbed [4], which we host in the rooftop of

IT Aveiro premises, was used to evaluate the performance of the prototype developed. As depicted in Figure 3, we used two neighbor wireless nodes and a fixed node (as client). The wireless nodes are powered by a 1Ghz Via Eden CPU, 1GB Ram, 1 Gigabit Ethernet Network Interface Card (NIC), and have two wireless NICs: a 802.11abg and a 802.11abgn. We used the later in our evaluation. Node B acted as an Access Point while Node A acted as a client connected to B. The Local Node only had one Ethernet connection to reach Node A, and was powered by an Intel Celeron 2.66Ghz with 1GB Ram. All nodes were running Debian Linux with kernel 2.6.30 and no services running besides syslog (system logger), ntpd (Network Time Protocol Daemon) and SSH (Secure Shell). An Ethernet based IP network connected Node A to the Local Node, and demonstrated to have an average round trip time of 0.177ms +/-5.1%, when no other traffic was present. Nodes had their date synchronized with an error less than 1ms using NTP (typical error was less than 80µs). The Local Node had a FlyingInterface interface replicating the wireless interface of Remote Node B. During the timeframe of each run, no other wireless traffic, external to the experiment, was generated in the surrounding environment.

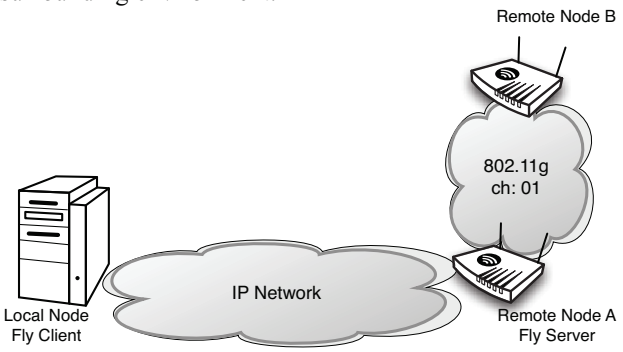


**Figure 3 - Evaluation scenario. Remote Node B is the destination of traffic sent by Remote Node A or Local Node**

The purpose of the tests was to evaluate whether the FlyingInterface approach was effective, and in addition verify that the results obtained were similar to the ones obtained when using the node directly. In our case the Local Node had more powerful processing capabilities than the Remote Nodes. Still, we do not make use of any computationally hungry application, only the MGEN traffic generator, since this would make testing in the wireless nodes unviable. More demanding applications would justify using our approach simply by the fact that the Remote Nodes would have no computational capacity.

Tests consisted of using the MGEN tool [5] to generate a CBR UDP flow to Remote Node B with packets of 1024 bytes (including IP and UDP headers). Traffic bitrate started at 32kb/s and doubled at each 10s of experiment time until it reached 1Mb. After this point, bitrate increased by 1Mb at each 10s. The same run was executed directly on Remote Node A and on the Local Node while using Node A replicated interface. Each run was repeated at least 5 times, with the exception of the latency results, which were obtained from 20 runs. All results have an associated confidence of at least 95%. All traffic is received at Remote Node B and stored to a RAM disk, together with a timestamp. We followed this approach in order to reduce jitter in the final data: Input-Output delay is much higher when using the existent CompactFlash card or the also existing NFS storage, than when using a RAM Disk.

Figure 4 depicts how throughput varied when sending the described flow from Remote Node A to Remote Node B and from Local Node (using the virtual interface) to Remote Node B. The difference between both results is presented as a percentage of variation from the mean of both values. As it is shown, throughput values are closely related with differences of less than 10% during the course of the entire experiment. Maximum differences between approaches occur in the 256kb/s - 1Mb/s range and at the 20 - 24Mb/s range. The later corresponds to a highly saturated wireless medium, and high input/output interrupt rate. System dynamics seem to imply higher variation at the 256kb-1Mb mark. This seems to be a result of synchronization of the multiple queues and schedulers (both at interface, network stack, and kernel level), in particular because of the TCP session used to connect Client and Server. Similar scheduling synchronization issues are presented and discussed in [9].

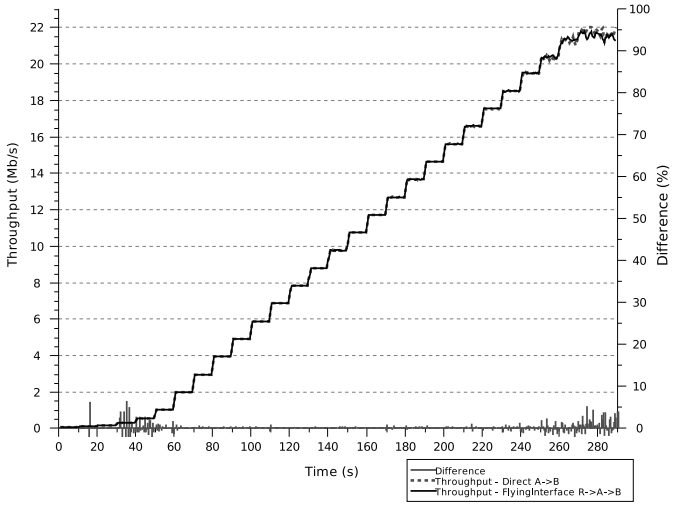


**Figure 4 - Throughput measured at the received (Remote Node B)**

Similar to the results obtained for throughput, results of end-to-end latency show to be closely related and following the same trends (see Figure 5). When using FlyingInterface, latency is higher by an amount that is directly related to the network delay between Local Node and Remote Node A. This is expected, as packets have an additional network hop to reach the destination node. As a result from the higher variation detected in the throughput charts, latency values also presents a maximum variation in the same intervals as before. The 256kb-1Mb interval presents a maximum difference of 16ms, which seems to be directly related to the bandwidth injected. This can be concluded due to the staircase appearance of the solid line in Figure 5. When the experiment was close to its end, large latency variation also occurs due to

delay in accessing the wireless medium because maximum capacity is being reached, and the hosts are under very high IOPS (Input/Output Operations Per Second) load. In particular Remote Host A that will have twice as much interrupts as the other nodes (receiving and sending at the same time). As stated before, these results are averages of 20 runs, and the differences observed are not due to noise.

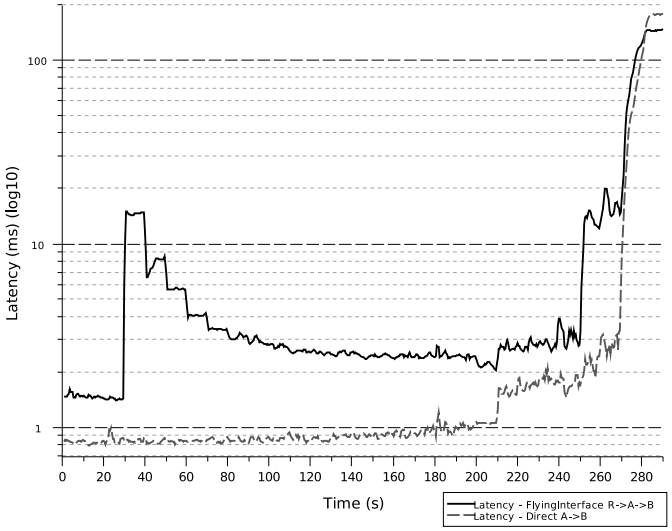


**Figure 5 - Variation of end-to-end latency as generated traffic increases. A $\log_{10}$ scale was used to better identify the differences between both experiments.**

During the experiment demonstrated previously, we also monitored the amount of Idle CPU at Remote Node A in order to better characterize the performance of our prototype. In Figure 6, and for better visualization, CPU Idleness curves are superimposed to network throughput. CPU Idleness was chosen instead of application CPU usage because the Idle CPU includes the amount of time the kernel spends waiting for IO or processing packets, and better reflects overall system impact.

Once again, results show that using the FlyingInterface Server module (solid line) implies a CPU usage similar to the direct usage for a packet generation tool such as MGEN (dashed line). When the wireless medium is saturated, results differ by a large amount. Further inspection showed this difference to be due to high CPU usage by the kernel itself (sys) and high IO wait. Such results imply some added overhead in the kernel (or the ath9k driver used) when injecting high amounts of packets through RAW sockets, or simply due to high IOPS. During ours tests, Remote Node A showed a maximum interrupt load of 6500 interrupts per seconds, when generating traffic locally. When traffic was being forwarded by using our solution, the number of interrupts per second reached almost 9000.

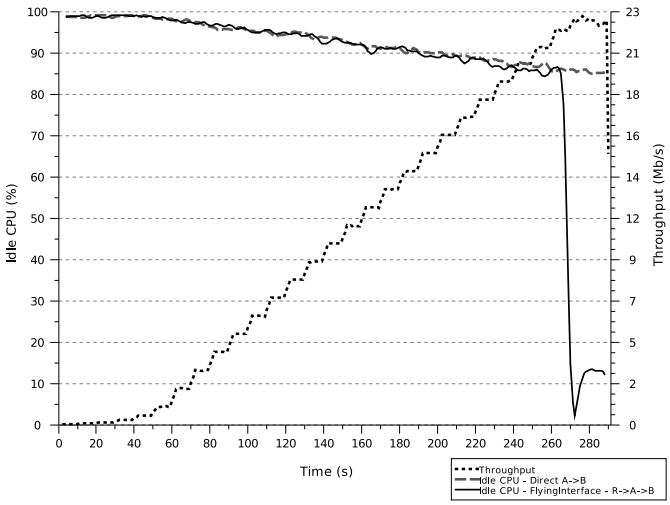


**Figure 6 - Amount of IDLE CPU as a function of traffic generated**

Besides data packets, also IOCTL requests are passed between the local and remote network interfaces. Figure 7 depicts the average time of two different IOCTLs calculated as an average of performing $10^6$ calls. For evaluating IOCTL performance we used the SIOCGIFMTU (Get Interface Maximum Transfer Unit) and the SIOGIWNAME (Get Wireless Interface Name). The first is a standard call supported by all network devices, while the last is a call supported only by 802.11 devices. As the solution proposed is agnostic of the actual IOCTL being invoked, any other combination of IOCTLs could be used.

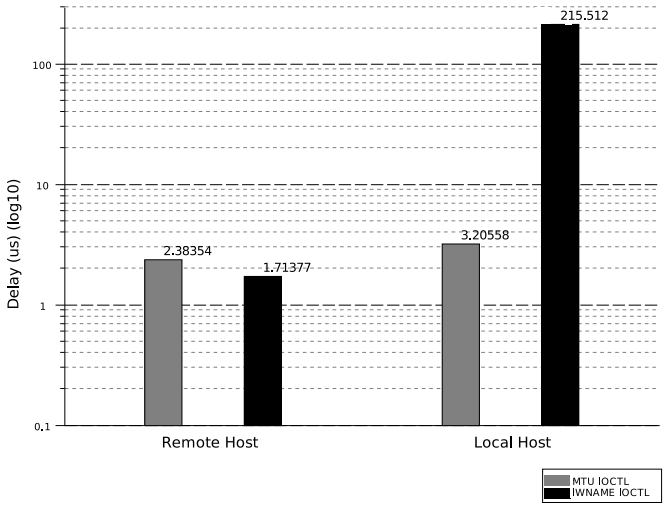


**Figure 7 - Comparison of IOCTL call duration between local host and using the FlyingInterface device. A $\log_{10}$ scale was used to better identify the differences between both experiments.**

In the Remote Node A (Server) each SIOCGIFMTU takes 2.38μs, while in the Local Node the same IOCTL takes 1.7μs, mainly due to a higher frequency CPU. In both cases, the IOCTLs are handled locally and serve as basis for comparison of the impact to both machines. The actual values reflect the

underlying hardware and computational capabilities of each host, and the absolute values are irrelevant for our evaluation.

The SIOCGIWNAME IOCTL is handled by the FlyingInterface driver, exported to the Client application and then forwarded to the Server application, which applies the IOCTL to the remote network device. The response follows a reverse path. In this case, while Remote Node A presents an average delay of 3.2μs for this IOCTL, at the Local Node, the IOCTL call takes much longer. As concluded, total time is the result of adding the delay of the transport network (177μs), plus some extra overhead, thus averaging at 215.5μs.

## V. CONCLUSION

The FlyingInterface approach accomplished all the requirements necessary to enable remote network communications in an almost seamless manner. Control and Data planes are linked between remote and local interfaces, providing new means to manage resources. Latency issues are inherited from the network interconnecting Client and Server, except unique cases that need to be studied upon, but which show low relative significance. This solution proved an invaluable technique for fast protocol prototyping in 802.11 wireless networks. In future work we intent to evaluate the impact of exporting other wireless interface technologies such as 802.16. While this should be rather simple as our concept is highly modular, and abstract in relation to the underlying technology, performance aspects still need to be assessed.

## REFERENCES

[1] Guffens, V. and Bastin, G. 2005, “Running virtualized native drivers in user mode Linux”, in proceedings of the Annual Conference on USENIX Annual Technical Conference (Anaheim, CA, April 10 - 15, 2005). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 40-40.

[2] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu and M. Singh, “Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols”, in proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2005)

[3] Singhal, S.; Hadjichristofi, G.; Seskar, I.; Raychaudhri, D., “Evaluation of UML Based Wireless Network Virtualization”, in proceedings of Next Generation Internet Networks, 2008. NGI 2008 , vol., no., pp.223-230, 28-30 April 2008

[4] João Paulo Barraca, Diogo Gomes, Rui L. Aguiar, “AMazING - Advanced Mobile wIreless Network playground”, in proceedings of ICST TridentCom 2010 - International Conference on Testbed and Research Infrastructures for the Development of Networks and Communities. Berlin, Germany, 18-20 May 2010.

[5] Multi-Generator http://cs.itd.nrl.navy.mil/work/mgen/index.php

[6] Network Block Device (TCP version) - http://nbd.sourceforge.net

[7] Nakao, A., Ozaki, R., and Nishida, Y.. “CoreLab: an emerging network testbed employing hosted virtual machine monitor”, in proceedings of the 2008 ACM CoNEXT Conference (Madrid, Spain, December 09 - 12, 2008). CONEXT '08. ACM, New York, NY, 1-6.

[8] Guffens, V. and Bastin, G. “Running virtualized native drivers in user mode Linux”, in proceedings of the Annual Conference on USENIX Annual Technical Conference (Anaheim, CA, April 10 - 15, 2005). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 40-40.

[9] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, Junichi Murayama, “Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency”, in proceedings of the SPIE, Volume 6011, pp. 138-146, 2005

[10] Philippe Biondi, EtherPuppet - http://www.secdev.org/projects/etherpuppet