Combining Heterogeneous Access Networks with Ad-Hoc Networks for Cost-Effective Connectivity

(Extended abstract of the MSc dissertation)

João Miguel Marques de Almeida da Cunha Mota
Mestrado em Engenharia de Redes de Comunicações
Instituto Superior Técnico

Advisor: Prof. Dr. Artur Miguel do Amaral Arsénio
Co-advisor: Prof. Dr. Helena Rute Esteves Carvalho Sofia

Abstract — Current networks comprise a multitude of different parameters, be it different transport technologies, protocols, or offered data rates. The multitude and variety of existing and emerging wireless and wired networking technologies continues to be the driving force towards convergence of networks. This is triggering the growing availability of communication devices with multiple networking capabilities. This work addresses the problem of using several different network interfaces, as a method to increase the throughput in wireless ad-hoc networks. It studies and compares different techniques that have been previously presented in the literature, and proposes an architecture applicable to a broad range of networks. To achieve such goal, we implemented an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc networks. The solution is based on a virtual interface (vi) approach, which allows the usage of all interfaces presented in a mobile device simultaneously, while hiding the heterogeneity from the applications. Additionally, it allows any number of interfaces to be added, increasing the total throughput. We further investigate methodologies to explore the availability of multiple interfaces in order to reduce energy consumption.

Keywords: *Multihoming, Heterogeneous wireless networks, Efficiency, Virtual Interface, Load-balancing.*

1. Introduction

The transparent support of a multitude and variety of existing and emerging wireless and wired networking technologies is a driving force towards convergence of networks. Moreover, it is commonplace nowadays to have electronic devices with multiple networking capabilities. Personal computing devices, e.g., laptops, PDAs, smartphones, are typically equipped with several networking interfaces ranging from different flavours of *Wireless Fidelity* (Wi-Fi) to Ethernet, GPRS, UMTS, and Bluetooth.

Adding to the diversity of network interfaces that end-user devices today include, the common Internet end-user has at his/her disposal a set of applications with significantly different bandwidth requirements and which comprise multimedia services, gaming, as well as collaboration, among others. However, most services provided today to the end-user simply take advantage of one network interface at a time.

This perspective is bound to change due to the fact that more and more, different *Service Providers* (SP) serve the same household or enterprise location. As an answer to this increasing complexity, several traffic-engineering techniques are being applied to take advantage of the different interfaces available on a single device. This is the case of multihoming and load-balancing, techniques which have been used to give networks some redundancy and redirect traffic flows based on the device necessities (power, signal strength, available bit rate, etc), thus assisting in making the network more robust. Hence, multihoming and load-balancing aspects are to be surveyed, analyzed and compared to the work developed in this paper, but as will be seen, the multiple and simultaneous use of different interfaces is still in an embryonic state, since it is not yet possible to make full use of all the physical interfaces present in mobile devices.

Our main objectives are two-fold. Firstly, to understand up to which point and for which cases it is relevant to consider a single interface (as a virtual container for all the potential network interfaces in an end-user device). Secondly, to analyze and evaluate up to which point is possible to achieve an efficient utilization of multiple network interfaces by devices via rate control and optimal assignment of traffic flows to available networks.

This document is organized as follows: the next section surveys previous work in this area, addressing several possible ways to improve the effectiveness of a heterogeneous ad-hoc network, as well as some problems that may arise from the implementation of such solutions. Section 3 presents our solution which is based on a virtual layer two device. In section 4 we evaluate the system and section 5 concludes the paper.

2. Related Work

This work addresses the efficient utilization of multiple network interfaces by devices, via rate control and optimal assignment of traffic flows to available networks, with a special emphasis on Ad-Hoc networks. This section introduces fundamental concepts, presenting a brief analysis of current related research.

2.1. Transparent Heterogeneous Mobile Ad-Hoc Networks

The authors' of this work [1] goal was to develop an end-to-end communication abstraction that supports MAC-switching[1], node mobility and multihoming[2]. Two issues to be solved are broadcast emulation and handover. Broadcast emulation because broadcast is not directly supported in Bluetooth (or on nodes comprising both Bluetooth and 802.11).

The authors define a Virtual Interface (*vi*) that is responsible for storing a MAC/Interface mapping, based on incoming packets. Like a Linux Ethernet bridge [3], the *vi* represents a regular layer-two-device and can be configured accordingly. The *vi* allows to plug in any 802.x compatible network device, like e.g a wireless LAN card or a BNEP/Bluetooth connection, while hiding the heterogeneity of the used devices from the upper layers. For every neighbouring node, the *vi* holds an array of possible outgoing interfaces in a so called neighbouring database. The author's solution is not bound to 802.11x or Bluetooth, but works together with any 802.x-compatible MAC Layer. The *vi* in combination with a MANET routing protocol supports multihoming, dynamic reconfiguration and node mobility.

If the *vi* receives a packet from the upper layer for delivery, it first checks the packet type. In case the packet is a broadcast packet, it will be sent through all available interfaces. Therefore, the *vi* also acts as a broadcast emulation layer for Bluetooth. However, if the packet is unicast, the *vi* looks for the corresponding entry in the neighborhood database mentioned above and retrieves the information about the interface the packet has to be sent to (entries are periodically checked for expiration). If there is more than one option, the *vi* makes use of another feature, the so called priority table. The priority table specifies a ranking among the interfaces, meaning that whenever a given neighbour can be reached through several interfaces, the interface with the lowest priority is taken. This means that the *vi* also acts as a load-balancing mechanism, capable of prioritizing interfaces based on different factors (e.g. energy consumption).

Even though this work presents an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc network, it does not make full use of the interfaces presented in mobile devices. Meaning, this solution does not offer the possibility to use both interfaces simultaneously, to send different traffic flows of information in order to increase the overall transmission rate.

[1] Refers to the fact that the used MAC technology may change along a source/destination path.

[2] A node having multiple network interfaces.

2.2. On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts

The authors of On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts, study if heterogeneous wireless interfaces can be aggregated with intelligent strategies to improve throughput beyond sum of the parts, as they call them super-aggregation principles. The authors propose three principles in the context of TCP that achieve super-aggregation benefits in Wi-Fi network when by adding a 3G interface [4]:

- S*elective offloading*: some of the interfaces may have a limited bandwidth, and by selectively offloading some portions of the data transferred it can cause a significant impact on the performance.
- *Proxying*: when an interface has only limited bandwidth but is up when the other interface is down, the limited bandwidth can be used for critical control information that in turn can serve to significantly improve the overall performance of the data transfer.
- *Mirroring:* for certain portions of the data being transferred intelligently mirroring the transfer on the interface with lower bandwidth can again have a profound impact on the perceived performance.

The *super-aggregation* principles presented can be implemented as a layer-3.5 software middleware in the mobile host. It can be implemented in the Linux kernel and uses NetFilter [5][6] to capture and process TCP packets traversing the network stack, or generate packets if necessary. The *super-aggregation* principles only require deployment at the mobile device and do not require any modification at the remote host or intermediate routers. The TCP implementations on the remote host and the mobile device are unaware of the *super-aggregation* principles that improve their performances transparently [4]. With this deployment model, *super-aggregation* can enhance end-to-end performance of mobile host with any legacy TCP-based server.

This solution although making possible the usage of two interfaces simultaneously (in this case Wi-Fi and 3G) and increasing the total throughput, does not escalate to more interfaces, does not take in consideration the use of two interfaces with similar bandwidth since it uses the interface with lower transmission rate to send certain small messages (e.g. ACK messages) and the other interface to send and receive the remaining data.

The tests prove that their solution in fact provides clear improvements in terms of throughput beyond the sum of the parts, which did not happen with other simple aggregation solutions [5][7][8], but unfortunately the authors only tested their solution with TCP data, neglecting the UDP data.

2.3. Linux Ethernet Bridge

The Linux Ethernet Bridge allows putting several real interfaces into a virtual bridging device. It is not only an in-kernel equivalent to a real Ethernet bridge but together with Netfilter a very sophisticated tool for packet filtering. Packets are forwarded based on Ethernet address, rather than IP address (like a router). Since forwarding is done at Layer 2, all protocols can go transparently through a bridge. The Linux bridge code implements a subset of the ANSI/IEEE 802.1d standard.

Bridging is supported in the 2.4 and 2.6 kernels from all the major distributors. The required administration utilities are in the bridge-utils [3] package in most distributions.

An Ethernet bridge distributes Ethernet frames coming in on one port to other ports associated to the bridge interface. Whenever the bridge knows on which port the MAC address to which the frame is to be delivered is located it forwards this frame only to this only port instead of polluting all ports together. Ethernet interfaces can be added to an existing bridge interface and become then (logical) ports of the bridge interface.
The advantage of this system is evident. Transparency alleviates the network administrator of the pain of restructuring the network topology.

3. The Virtual Network Interface System

We conceived a virtual interface that is able to, not only perform load-balancing, but also analyze each equipment needs using a priority and a neighboring database table. This virtual interface (vi) besides hiding the network heterogeneity from the application, aggregates transparently the physical interfaces under it, and selects the interfaces to be used. If necessary, it will perform as well the handover, in case an interface is no longer available. The architecture of the implemented vi will be explained in more detail during the next sections.

Although we are interested in a generic solution, we take a network that combines different flavors of 802.11 as a basis for this work, in particular to assist realistic experimentation.

3.1. Proposed Architecture

In terms of architecture, it is divided into 4 main blocks:

- Virtual Interface.
- Priority Table.
- Decider / Virtual Bandwidth Aggregation (VBA).
- RTT Estimator.

Figure 1 describes the path that the data coming from a certain application takes, until it reaches the physical interfaces, passing through our virtual interface.

The data coming from and to the application is intercepted by the virtual interface, which will check three parameters: the priority, availability and RTT of each interface. Then the Virtual Bandwidth Aggregation block (VBA) will decide how to distribute the intercepted data between the available physical interfaces.

The virtual interface is similar to the Linux Ethernet Bridge [10]. The *vi* represents a regular layer-two-device and can be configured accordingly. The *vi* supports any 802.x compatible network device, such as wireless LAN card or a Bluetooth connection, while hiding the heterogeneity of the used devices from the upper layers [9].

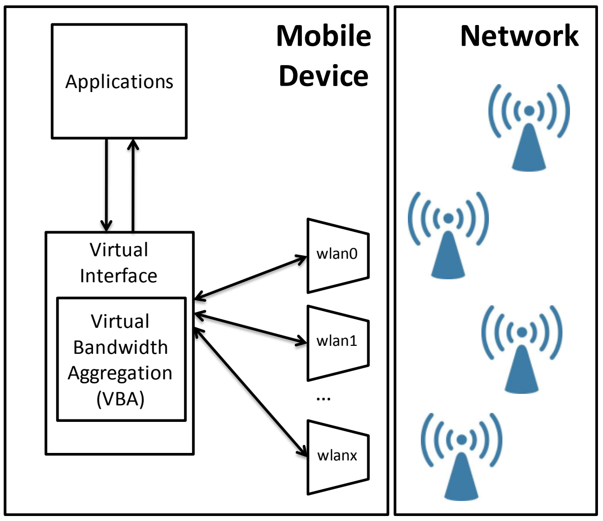


Figure 1: Interaction between the implemented blocks.

The *vi* also holds an array of possible outgoing interfaces in a neighboring database (NDB), similar to the Linux bridge’s forwarding database. An entry contains a timestamp and it is created upon receiving the first packet (i.e. a routing broadcast message or a route reply) of the associated neighbor/interface pair. Every consecutive incoming packet refreshes the timestamp. With this information the *vi* has a view of all neighbor nodes and the interfaces that are available to be used.

The priority table specifies a ranking among the interfaces, meaning that whenever a given neighbor can be reached through several interfaces, the interface with the highest priority is taken, being 0 de highest.

The default priority is also 0, which means if the user wants a specific interface to be used, he has to define the priority of each interface. If there are interfaces with the same highest priority, all of those are used simultaneously, since no limitations were set by the user (e.g. no preference between Wi-Fi over Bluetooth).

The *vi* collects information from the priority table to select a set of interfaces for communication, according to each device’s needs. It is responsible for deciding on handovers and to perform them, switching the traffic from one interface to another using a simple timer.

The VBA/decider, has information concerning the interfaces that can be used from the vi, and according to that information chooses how the data, we want to send, is divided between those interfaces. This is the mechanism that will increase the total throughput, in comparison with a simple solution, since we are dynamically allocating the data we want to send between the existing interfaces.

Based in the number of physical interfaces present in a mobile device, their priority and bandwidth, the VBA/decider, decides what to do, in this case which interfaces should be used. The implementation details, and how the decision is taken by the VBA, are explained afterwards, in section 3.2.

The RTT Estimator is responsible for measuring the average RTT for each physical network interface, which will then relay it to the VBA so it can decide which interfaces to use. The estimation is based on the traffic leaving the device, and it is done so we can have a clear image of the neighboring nodes, and if a certain path used by a physical network interface is congested or not. This estimation is made periodically and the values stored in a hash table, so that the VBA can easily access this information.

3.2. Implementation Aspects

This section is exclusively dedicated to the contributions of this work both in terms of concepts, implementation, and analysis.

The virtual network interface for transparent heterogeneous mobile ad-hoc networks in terms of implementation consists of three parts:

1. A kernel module providing the actual network interface.
2. A library providing programmatic access to the configurable options.
3. A userspace utility to manage virtual interfaces.

In terms of implementation, we first needed to rewrite the previous implementation of the virtual interface code [1], since it was limited to a very specific version of the Linux kernel, and only worked with *sysfsutils* v1.x [11]. In version 2.x *sysfsutils* suffered a number of changes to the way attributes were populated, another significant change was the removal of *struct sysfs_directory*, which rendered the previous module implementation non operational.

The second step was to improve the method used to intercept the data, since the previous one was too evasive [2]. The hook was placed in the general packet reception routine of a network device. Before passing the *sk_buff* to the upper layers it was checked if it has to be passed to a virtual interface. This previous solution added so much overhead to the vi, that the total throughput was significantly affected.

The solution we found was to insert Netlfilter hooks, removing the need to recompile the Kernel with the patch inserted into the *dev.c* file, substantially reducing the overhead added.

The next step was to add a new block to the virtual interface, named Decider / Virtual Bandwidth Aggregation (VBA). This block is responsible for choosing which physical interfaces to use from the ones behind the virtual interface. The VBA makes this decision based on three parameters: the priority and RTT of each interface and their availability according to the neighboring database, which contains the available neighbors and the path used to reach them.

Based on these three parameters, the VBA chooses how the data stored in the *dev_queue_xmit* buffer will be redirected to the available interfaces. For this purpose the physical interfaces are transparently aggregated under the virtual interface and a load balancing mechanism was implemented to distribute the data between the available interfaces. To do this we calculate the RTT of each interface, and use a simple function to calculate a value in the form of percentage, for each interface. This value defines the percentage of data intercepted by the vi a physical interface is responsible for. By doing this we are dynamically balancing the traffic between our physical interfaces, taking in consideration not only their RTT but also the paths actually being used.

The way we aggregate the interfaces under the virtual interface is the same used by the one implemented by the Linux bridge [3], where there is an aggregation of several interfaces, and the traffic is redirected between them. What was done was an adaptation of the mechanism used by Linux Bridge to our virtual interface, so that it would also work in an ad-hoc network environment.

Additionally, the VBA is also able to monitor the device's power levels (the amount of battery left and if the device is plugged in to any power adapter), and if needed it will reduce the energy consumption by dynamically choosing the interfaces, based on their power consumption and throughput, making certain that the device uses the minimum amount of power to send the data. This extra function was also created from scratch, allowing the virtual interface to balance the data in a different way according to the power level, in order to save some energy.

The detailed implementation of each block will be presented in the next sections.

3.2.1. Data Interception

The most promising method we found to intercept the data, was using a custom Netfilter target. Such a target can be loaded and unloaded from kernel at any time. A well-understood architecture in the kernel and a userspace utility makes Netfilter a powerful tool. The Netfilter target for the virtual interface and other known Netfilter targets can also be combined in any favored way [5].

Packets will pass through hooking points sequentially. On each hooking point, it is possible to configure some filtering rules via the *iptables* command. After packets pass through NF_IP_PRE_ROUTING, the Linux kernel makes the routing decision to decide whether packets should enter the local processes or be routed to the next hop through the virtual interface and then redirected to a certain physical interface (this is decided by the decider).

The Netfilter hook is created by the *net_hook* function, and registered using *nf_register_hook*. By adding the hook, we are intercepting and storing each data flow in a temporary buffer (*dev_queue_xmit*), while the virtual interface decides to which network interface(s) it should be redirected to.

3.2.2. The Neighbor Database (NDB)

The neighbor database is a hash table with the hash function calculated on the mac address. A linked list for each hash value contains the entries corresponding to neighbors (cf. Figure 2).

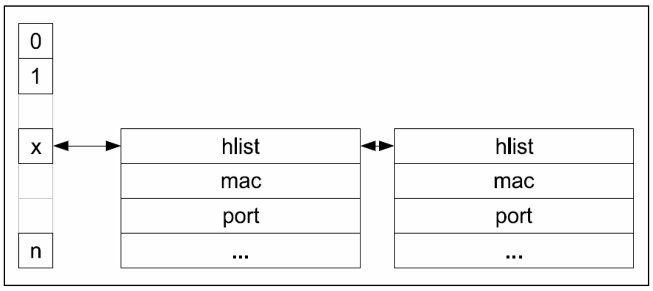


Figure 2: The neighbor database (simplified).

At the very beginning the neighboring database contains no entries but the transmission of a broadcast packet does not need any neighborhood information anyway. After the route request has passed several hops, a route reply eventually returns back to the origin. The route reply not only establishes the route but also creates an entry within the neighboring database, providing the *vi* with information on the interface to which the packets to the given neighbor have to be transmitted.

In the case of a pro-active routing protocol, things are slightly different. Here nodes periodically broadcast their neighboring information and therefore are also creating entries within neighborhood databases. In both cases (proactive and reactive) the NDB entry is established in combination with the new route, regardless of whether the MAC technology changes or not.

Insertion

The function to insert and update entries into the neighbor database is the same. First, the hash table is searched for a matching entry. If one is found, it is updated; otherwise a new entry is created. The update sets the timestamp to the kernel time *jiffies*[3].

Outgoing link selection

Outgoing links are selected according to the available neighboring nodes, present in the neighboring database. First we check if there is any available neighbor, if not, then the network interface cannot be used. After knowing which network interfaces can be used, the VBA decides which ones to use, based on their priorities and RTT estimation.

This structure is used to store the information of the available neighbors in an ad-hoc network, particularly the available nodes and which interface should be used to establish a connection with a certain node. We also added the possibility to use the virtual interface in a non ad-hoc scenario, which widens the possible scenarios the vi can be used in.

3.2.3. Decider / Virtual Bandwidth Aggregation (VBA)

The VBA is responsible for choosing how the data, we want to send, is divided between the available interfaces. This is the mechanism, within our solution, that was created from scratch and shall increase the total throughput, in comparison with a basic setup, without the virtual interface, since we are transparently aggregating the available interfaces under the vi and dynamically allocating the data we want to send between the existing interfaces.

As mentioned before, there are several steps the vi must complete before choosing how to divide the data between the physical interfaces. First it is necessary to check three parameters: the priority and RTT estimation of each interface and their availability. Based on these three parameters the VBA chooses how the data stored in the *dev_queue_xmit* buffer will be redirected to the available interfaces.

To store the priority of each physical interface, we created a simple hash table that stores the names of each physical interface within a certain virtual interface, and their corresponding priorities. The access to the priority table is done in order do find all the physical interfaces with the highest priority, being 0 the highest.

After knowing which interfaces to use and their availability, it is necessary to calculate the *Round-Trip Time* (RTT) of each physical interface.

For this purpose and since TCP continuously estimates the current RTT of every active connection in order to find a suitable value for the retransmission time-out, we implemented a mechanism capable of calculating the RTT using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. For every data stream sent using TCP there is an acknowledge response that reaches the mobile device, these packets are intercepted by the Virtual Interface, which will then extract the RTT estimation.

After extracting the RTT information out of the TCP ACK packets every 5 seconds, we use formula (1) to calculate the *smoothed RTT* (SRTT), which give us a more correct estimation of the actual average RTT for each

[3] A jiffy is the duration of one tick of the system timer interrupt. It is not an absolute time interval unit, since its duration depends on the clock interrupt frequency of the particular hardware platform.

network interface. With each new sample Si, the new SRTT is computed as [12][13]:

$$SRTT(i+1) = \alpha \times SRTT(i) + (1-\alpha) \times S(i) \quad (1)$$

Where SRTT(i) is the current estimate of the round-trip time, SRTT(i+1) is the new computed value, and α is a constant between 0 and 1 that control how rapidly the SRTT adapts to changes (usually α=1/8).

By applying formula (1) with the information extracted from the ACK packets constantly arriving, we are capable of estimating the average RTT values for each physical interface, without producing additional data.

Function (2) is used to balance the data in a proportionate way through the several physical interfaces, and is executed for each data flow that is intercepted by the virtual interface.

$$P_{I_a} = \frac{1}{RTT_a \times \sum_{j=1}^{n}\left(\frac{1}{RTT_j}\right)} \quad (2)$$

The P_{Ia} is the percentage a certain interface should be used to transfer the data intercepted by the vi and its value is between [0, 1]. The sum of all PI's must be one and it is calculated for every single available interface with the highest priority. The RTT is the Round-Trip Time of a certain interface and the summation interval is between 1 and n, being n the total number of available physical interfaces with the highest priority.

When a virtual interface is first created, and several interfaces are added, the table containing the results from formula (2) is empty. For this matter we use function (3), which uses the bandwidth from each interface, as a metric, to calculate the necessary proportions that will be used to calculate the amount of data each physical interface is responsible for, within a certain data flow.

$$P_{I_a} = \frac{Bandwidht_a}{\sum_{j=1}^{n}(Bandwidht_j)} \quad (3)$$

Again, the P_{Ia} is the percentage a certain interface should be used to transfer the data intercepted by the vi and its value is between [0, 1]. The Bandwidth values are acquired via *SysFS* and the summation interval is between 1 and n, being n the total number of available physical interfaces.

After calculating all the PI's, the VBA will now redirect the data to the physical interfaces, taking into consideration the obtained values.

In comparison with the previous versions [1][2], where the authors only used priorities to divide the data between the physical interfaces, we are now using a dynamic load-balancing mechanism since we are dynamically allocating the data through the existing interfaces.

3.2.4. Power Saving Mode

If the battery level is below 10% and if the mobile device is not plugged in to any power adapter then the power saving mode is activated. We take into consideration the RTT values, in order to extrapolate the throughput and the energy consumption of each interface. Based on these two parameters we find the solution that consumes the least amount of energy to send the data.

The Throughput is measured in bits per second, it is estimated based on the RTT measurements and it is calculated using formula (4). Note that by default the TCP Buffer size >= TCP Window size. Typical TCP window size is equal to 64 Kbyte, and the RTT is measured in seconds.

The value we obtain in formula (4) is a theoretical value of the throughput. It is calculated in order to estimate the energy consumption of a certain interface and is used in formula (5). To simplify the calculation we are assuming a packet loss of 0%, since the obtained values are merely for comparison reasons, so we can understand which interfaces use the most amount of energy to send a certain data flow.

$$Throughput = \frac{TCP\ buffer\ size}{RTT} \quad (4)$$

When a node sends or receives a packet, the associated network interface, decrements the available energy according to the following parameters: (a) the specific network interface controller (NIC) characteristics, (b) the size of the packets and (c) the bandwidth used. The following formula represents the energy used (in Joules) when a packet is transmitted or received (Formula 5) and the packet size is represented in bits.

$$Energy_{tx,rx} = \left(\frac{Energy\ Consumpt.*Energy\ Supply*PacketSize}{Throughput}\right) \quad (5)$$

The energy consumption is measured in miliamperes (mA), varies with the interface being used and if a packet is being transmitted or received. The energy supply also varies with the device being used and is measured in Volts (V).

Although the equipment consumes energy, not only when sending and receiving but also when listening, we have assumed in our model that the listen operation is energy free, since all the evaluated ad-hoc routing protocols will have similar energy consumption due to the node idle time.

After knowing how much energy a network interface requires for sending a packet, we can now calculate if the current set up, defined by the VBA is consuming the least amount of energy to send a certain data flow. For that we use formula (6), representing the energy consumed during the transmission of the data present in the output buffer (in Joules). The BufferSize and PacketSize are both represented in bits and the summation interval is between 1 and n, being n the total number of available physical interfaces with the highest priority. The PI represents the value calculated in either formula (2) or formula (3) and Energy represents the energy used (in Joules) when a packet is transmitted, and it is calculated in formula (6).

$$Energy\ Consumed_{VBA} = \frac{BufferSize}{PacketSize} \times \sum_{j=1}^{n}\left(Energy_j \times PI_j\right) \quad (6)$$

We then compare the acquired energy consumed value with the energy the interface with the lowest energy consumption would require for sending the same amount of data. For that we use formula (7). The parameters are the same as the ones in formula (6), but now we are only taking in consideration one interface, not all the interfaces present in the mobile device.

$$Energy\ Consumed_I = \frac{BufferSize}{PacketSize} \times Energy_I \quad (7)$$

After acquiring this second value we compare both energy results, and verify if EnergyConsumedVBA ≥ EnergyConsumedI. If this is the case, then the VBA will only use the interface with the lowest energy consumption to transmit the data flow, since it will consume less energy.

4. System Evaluation

This chapter is dedicated to the performance evaluation of the main building blocks of this work, attempting to answer the questions that lead to this work and that can be aggregated into three main aspects:

- Is the overhead added by the virtual interface excessive?
- Are the implemented mechanisms improving the total throughput?
- Is the handover time affected by such end-to-end abstraction?

The traffic used in the simulations was generated by relying on *iperf*[4], since it is supported by both Linux and Windows operated systems, via its graphical component *jperf*[5]. It is more focused on measuring the network available bandwidth, capable of measuring bandwidth and datagram loss and also presents the results of jitter and RTT. To evaluate the overhead and handover time, we used the command ping.

For each test, in terms of measuring the available bandwidth we also tested different packet sizes, which are important to determine the per-packet overhead. The values are averaged over 20 readings and each reading takes 100 seconds to acquire (except the handover time that requires only the number of packets lost while handover is occurring).

All readings were taken on a laptop Sony Vaio PCG-7N2M. A Tsunami desktop computer using Windows Vista operating system was also used as server, to receive the data sent from the laptop running the virtual interface module. The access points used in the experiments were routers Fonera+[6], flashed with the Linux based firmware DD-WRT[7]. Our solution was tested with AODV-UU [14] routing protocol and different types of data (TCP and UDP). The obtained results were of course compared with a simple scenario with no virtual interface, no bandwidth aggregation and no load-balancing mechanisms, referred as RAW.

The experiments run considered three different topologies as basis for the different developed scenarios. In each topology we test both the load-balancing mechanism as well as the power saving mode, in terms of throughput and delay, to understand the actual impact of the virtual interface.

The first topology considered (Topology I) is illustrated in Figure 3. For the mentioned topology, node A is connected by means of an ad-hoc network. The purpose of this configuration is related to the need to get data that serves as control, relating to the situation where we are not using the virtual interface, which will then be compared with more complex scenarios where we use the vi. We also use this topology to estimate the delay added by the virtual interface, in order to understand the impact the vi is causing in terms of throughput in a situation where only one single node is available. The virtual interface will be situated behind the physical interface, as presented in Figure 3.

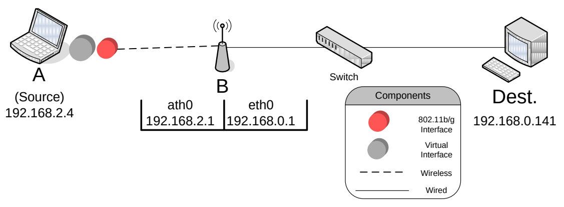


Figure 3: Topology I, one interface and one AP.

Topology II is in fact similar to I, being the only difference the number of physical network interfaces present and the number of access points, which was now increased to 2. Such increment will assist in trying to understand the impact caused by the number of interfaces in the performance evaluation of the virtual interface. Two tests will be made using this topology. In the first one, both access points will only have one user connected and transmitting data, while in the second test, one of the access points will be saturated with data from a third party computer.

Finally, topology III still consists of paths that are one hop long, but now there is just one AP and three network interfaces. In this experiment the AP is saturated with data from a secondary source. Our expectations were to observe what would be the reaction of the vi when there is an increment in the number of interfaces, in a worst case scenario, where there is just one AP and multiple network interfaces. For this scenario we also used different network interfaces, two IEEE 802.11g and one IEEE 802.11b.

[4] Iperf, http://sourceforge.net/projects/iperf/
[5] Jperf, http://sourceforge.net/projects/jperf/
[6] Fonera, http://www.fon.com/
[7] DD-WRT, http://www.dd-wrt.com

	1KB	STD (σ)	2KB	STD (σ)	4KB	STD (σ)	8KB	STD (σ)	16KB	STD (σ)	32KB	STD (σ)
RAW (Mbps)	21,914	0,381	22,139	0,343	22,814	0,421	23,108	0,396	23,363	0,414	23,743	0,442
Virtual interface (Mbps)	21,047	0,511	21,243	0,547	21,984	0,650	22,455	0,682	23,022	0,715	23,482	0,788
Virtual Interface w/ Power saving mode on (Mbps)	21,001	0,523	21,109	0,581	21,807	0,694	22,393	0,702	22,900	0,763	23,374	0,802

Table 3: Wlan throughput in Mbps, different packet sizes.

4.1. Experiment 1

Starting with the analysis of the delay added by the vi, Table 1 shows the round-trip time in milliseconds and zero percentage of packet loss for the two same situations. This information allows us to have a very clear image of the delay added by the virtual interface since ping was used with the default packet size (64bytes). The smaller the packets, the more of them there are, which makes it easier to calculate the difference in terms of RTT between the control, designed as RAW, and our solution, since the differences are most likely caused by the vi module The values are taken during 100 seconds and averaged over 20 readings.

	Average RTT (ms)	Std. Deviation (σ)
RAW	1,730	0,009
Virtual Interface	1,799	0,016
Virtual Interface w/ Power saving mode on	1,808	0,014

Table 1: Ping results, 1 interface and 1 access point.

As we can see by the results presented in Table 1 the delay added by the vi, while the power saving mode is off, is around 4.0%, and roughly 4.2% with the power saving mode on, which is almost irrelevant (less than 0,1ms). The standard deviation also increases, while using the vi, since it is periodically estimating the RTT values for the available interfaces, which causes some fluctuations in terms of the total throughput. Also, there was zero percent packet loss for all three test situations.

Still in Experiment 1, we calculated the handover time, using the command ping. The number of missing packets were counted and multiplied by the ping frequency. The type of handover that was measured was the horizontal handover, where the MAC level protocol remains the same, but the route changes. We trigger a route change by physically detaching the interface of a node. The results are presented in Table 2. Handover times are averages over 20 readings with standard deviation σ. Entries of the form "VI/x" must be understood as "Virtual Interface with a *maxdiff* value of x".

Under our setup, when the vi was not being used, we measured a packet loss of roughly 1.5 packets when the route changed from one hop to another. Each packet that is lost corresponds to roughly one second passed by, since the ping frequency that was used was one second.

From the results presented in Table 2 we see that packet loss increases with increasing *maxdiff* threshold. The former is reasonable because the bigger the *maxdiff* value, the more the priority policy gets enforced, and a pure priority driven MAC switching would not lead to any switching at all. As expected, the smaller *maxdiff* gets the less stable the handover becomes. However, in our scenario a *maxdiff* value of 10 was sufficient to guarantee stable handover while changing interface priorities.

Type	Interface	Time (s)	Standard Deviation (σ)
Horizontal	RAW	1.5	0,5
	Vi/10	1.8	0.92
	Vi/100	2.3	0.73
	Vi/1000	2.5	0.67

Table 2: Handover time in seconds, using Wi-Fi interfaces.

Regarding the total throughput using TCP data, Table 3 shows the throughput and corresponding standard deviation (σ) when using the virtual interface, and the raw measurements taken without it.

The results we obtained for the first scenario show that the difference in terms of throughput between using or not the virtual interface for TCP, with just one interface, is very small. Around 3.1% less in average with the power saving mechanism off and 3.5% while on. Which is a good indicator, since with this topology the vi is simply relaying packets to the available network interface.

Finally in Experiment 1, we calculated the amount of energy the mobile consumed during 600 seconds, with and without the vi module, so we could understand this abstraction impact on the energy being consumed by the mobile device.

	Energy Consumption (mWh) per second	Energy Consumed (mWh) in 600s	Standard Deviation (σ)
RAW	5,294	3123,333	1,035
Virtual Interface	5,316	3136,667	1,209

Table 4: Energy consumption in milliwatt hour.

The values in table 4 are presented in *miliwatt hour* (mWh). They were measured during 600s and averaged over 10 readings. To measure the consumed energy during this period we used the bash script. It extracts the energy the device has, for each 10 seconds. The following table presents both the average energy consumption in mWh per second, and the total energy consumed during the 600s period.

As we can see in Table 4, the difference in terms of energy consumed during 600 seconds, between both scenarios, was only 13,33 mWh, just 0,427% more. This result proves that the module consumes just a minor added amount of energy when compared with a control scenario, designed as RAW in Table 4.

4.2. Experiment 2

The experiment 2 is divided in two scenarios. Both rely on Topology II. While in the first test, the two existing APs are only being used by one node, in the second test, one of the APs is also being used by a second node that is constantly sending data, to simulate a saturated AP. The two physical interfaces used for this experiment were IEEE 802.11g.

In this experiment we investigate wherever the virtual interface is capable of detecting a saturated AP and reducing the amount of data a certain physical interface is sending to it, by diverting part of the traffic to a second physical interface that is using a less saturated AP.

Two Access Points, no saturation

For this scenario we used one data flow coming from a single application and multiple data flows coming from different applications, so the available interfaces could be used simultaneously. The throughput values were measured during 100 seconds and averaged over 20 readings. The packet size used for this experiment was 32 Kbytes.

The total throughput when using the virtual interface with TCP data (while sending a single data flow) is very similar in comparison with RAW measurements taken without the virtual interface. On the other hand the values obtained when sending different data flows, are nearly 79% higher when comparing with the RAW measurements from Experiment 1. This increment results from the fact that we are simultaneously using several physical interfaces to send the different data flows.

	Average Throughput (Mbit/s)	Standard Deviation (σ)	Packet loss (%)
One Data Flow	23,627	0,610	1%
Multiple Data Flows	42,615	5,161	3%
RAW	28,879	0.410	0%

Table 5: Throughput in Mbps, 2 network interfaces and 2 APs.

The packet loss is in average 1% and 3% (cf. Table 5) for one data flow and multiple data flows respectively, due to the fact that we are constantly switching the physical interfaces used to transmit the data, which causes some packets to be lost and some fluctuations in terms of throughput, raising also the standard deviation.

Two Access Points, AP C is saturated

For this scenario we saturated one of the access points, to verify how would the vi adapt to a sudden increase in terms of the RTT value measured for a certain physical interface, connected to that AP. As in the previous test experiment, single and multiple data flows were tested. The values obtained were taken during a time frame of 100 seconds, and averaged over 20 readings. For a better comparison, we also measured the throughput of a single interface, without using the vi module, designated as RAW, connected to a single saturated AP. The packet size used for this experiment was 32 Kbytes.

In this specific scenario, with a single data flow, the usage of the vi module with two different physical interfaces is in average, increasing 63% the total throughput, when comparing with a situation where a single interface is connected to a saturated AP (11,713 Mbps).

When sending different data flows, we obtained a very similar result to the one presented in the previous scenario (2 APs, no saturation). It takes around 6s for the throughput to reach a maximum of 35Mbps, when the vi starts using both physical interfaces, and consequently the two APs.

	Average Throughput (Mbit/s)	Standard Deviation (σ)	Packet loss (%)
One Data Flow (vi)	19,125	5,491	2%
Multiple Data Flows (vi)	33,575	5,551	3%
RAW	11,713	0,428	1%

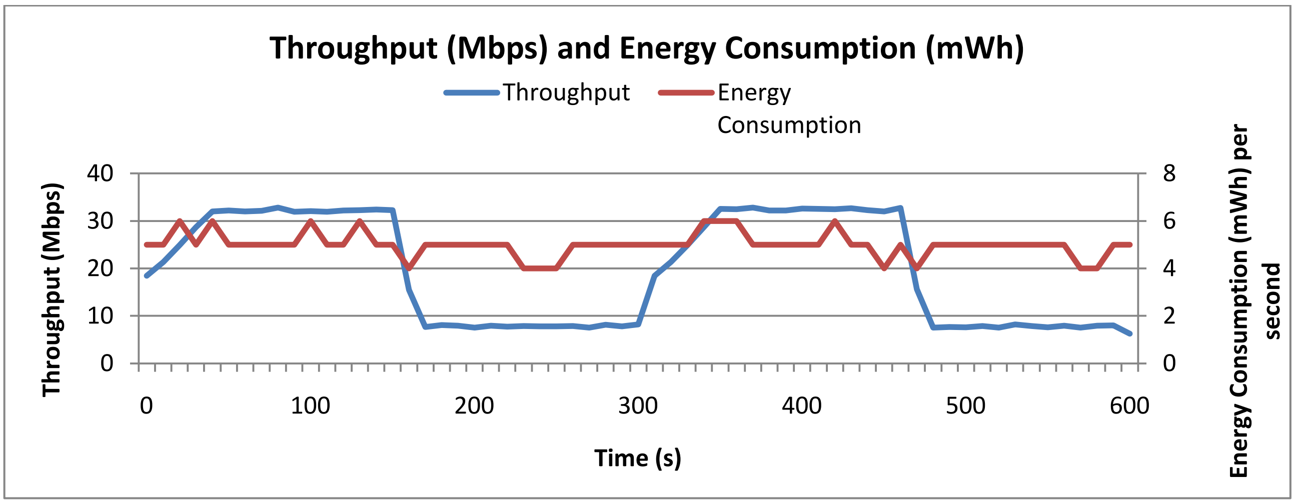
Table 6: Throughput in Mbps, using two interfaces and two APs (one saturated).

The load-balancing mechanism presented in the vi module slightly increases the packet loss and standard deviation as seen in Table 6. If we compare both the standard deviation results, of this and the previous scenario, when sending a single and continuous data flow, there is a slight increment, which is caused by the difference in throughput values obtained for the two access points (11.5 Mbps and 23.5 Mbps).

4.3. Experiment 3

To understand and find the potential limitations of the vi module, we decided to create a worst case scenario. It relies on Topology III, where we have three physical network interfaces (two IEEE 802.11g and one IEEE 802.11b) connected to just one saturated access points.

Starting with the analysis of the total throughput using TCP data, Table 7 shows the throughput when using the virtual interface with one and several different data flows during 100 seconds, averaged over 20 readings. The packet size used for this experiment was 32 Kbytes.



Graph 1: Correlation between the total Throughput in Mbps and the Consumed Energy in miliwatt hour per second (mWh/s).

We also added the throughput of a single interface, calculated without using the vi module, designated as RAW, connected to a single saturated AP (values calculated in experiment 2), for a better comparison.

	Average Throughput (Mbit/s)	Standard Deviation (σ)	Packet loss (%)
One Data Flow (vi)	9,915	2,664	2%
Multiple Data Flows (vi)	14,756	1,021	4%
RAW	11,713	0,428	1%

Table 7: Throughput in Mbps, using three interfaces and one saturated AP.

As seen in Table 7 both the packet loss and standard deviation, for the two test experiments are above the values obtained for the control scenario, described as RAW. This difference can be easily explained due to the fact that for this experiment we have three network interfaces, one of them has a lower bandwidth than the other two, which causes a bigger fluctuation in terms of total throughput. The one percent increment in the packet loss may result from the usage of multiple interfaces in a saturated environment.

4.4. Experiment 4

Finally in Experiment 4 we tested the power consumption mechanism, to try to understand if by using fewer interfaces to send the data we are able to diminish the amount of energy being spent and if the difference is significant. This experiment relies on Topology II, the two network interfaces (1x IEEE 802.11g, 1x IEEE 802.11b) are connected to two access points. Real traffic (different data flows, from different applications) was used, to simulate a real-life environment. The data for both measurements was collected over 600 seconds (10 minutes) and is presented in Graph 1. Each measurement is displayed according to its own axis, so we can then verify if there is a correlation between the throughput and the energy values. The energy consumption is presented in *miliwatt hour* (mWh), while the throughput is displayed in Mbps.

We can clearly verify from Graph 1, that when there is an increment is terms of throughput, there is also a slight increment is terms of the energy being consumed by the device. The opposite is also true. This happens because the power consumption mechanism, depending on the size of the data flow, decides if the device should use the network interfaces picked by the VBA or the interface with the lowest energy consumption to send the data.

When a data flow has a significant size, it is usually better to maintain the VBA's policy, since the device will transmit the data at a higher rate, which will result in a lower amount of energy being consumed to send that amount of data. When data flows have a relatively small size, it takes roughly the same time to send them when using one or several network interfaces, resulting in a noticeable energy saving when using just one of those interfaces (5.4%). This is exactly what we see in Graph 1.

During this experiment, the mechanism saved roughly 190 mWh of battery in our device, which for a laptop is not very significant, but in a smaller and more economic device, such as a mobile phone or a sensor, this difference can have a very big impact, since it gives extra time of battery.

5. Conclusions and Future Work

We implemented an end-to-end communication abstraction that can be used in heterogeneous mobile ad-hoc networks. Such networks are characterized by different MAC technologies used among the nodes. The solution is based on a virtual interface (vi) approach, which allows the usage of all interfaces presented in a mobile device simultaneously, while hiding the heterogeneity from the network and allow any number of interfaces to be added, increasing the total throughput.

Implementing a virtual interface for transparent heterogeneous mobile ad-hoc networks has proven to be a good approach.

Overall there is a slight overhead when relying on the *vi*, be it from a throughput, power consumption, or from a delay perspective. This is expected, as by adding a layer of abstraction, we are also adding overhead in computation, with the expectations to introduce significant advantages. Reasonable handover times can be achieved when using any routing protocols. The throughput rates when using the vi module, with several network interfaces, are significantly higher, reaching in some situations an increase of 79%. In terms of the power consumption mechanism, the experimental values are also very promising; by using this mechanism we were able to optimize the amount of energy being consumed.

As future work, we would like to extend the virtual interface to work in different access networks, others than ad-hoc networks, with for example several 3G and 802.11x interfaces. Moreover, it would be interesting to extend the module to smaller devices, such as mobile phones, to see how it would impair their performance in terms of their total throughput and energy consumption.

References

[1] P. Stuedi and G. Alonso, "Transparent Heterogeneous Mobile Ad-Hoc Networks". In *Proceedings of the Second Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, USA, July 2005.

[2] S. Graf, "Implementing a virtual network interface for heterogeneous mobile ad-hoc networks (802.11 and Bluetooth)", Swiss Federal Institute of Technology, Zurich, August 2006.

[3] James T. Yu, "Performance Evaluation of Linux Bridge", DePaul University, 2004.

[4] C. Tsao and R. Sivakumar, "On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts", Georgia Institute of Technology, December 2009.

[5] Netfilter - Firewalling, NAT and packet mangling for Linux [Online] Available: http://www.netfilter.org/ [Accessed: Aug. 2010].

[6] The Netfilter project team, "Linux Netfilter/Iptables frameworks", Nov 1999. [Online]. Available: http://www.netfilter.org/. [Accessed: Aug. 2010].

[7] H.-Y. Hsieh and R. Sivakumar. "A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts". In *MobiCom '02 Proceedings of the 8th annual international conference on Mobile computing and networking*, Atlanta, September 2002.

[8] K.-H. Kim, Y. Zhu, R. Sivakumar, and H.Y. Hsieh, "A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces". *Wireless Networks*, 2005.

[9] J. Mota, A. Arsénio and R. Sofia, "Combining Heterogeneous Access Networks with Ad-Hoc Networks for Cost-Effective Connectivity". In *API Review 2010, 1º volume, Edições Lusófonas*, February 2011.

[10] Mirzaie, S. Elyato, A.K. Sarram, M.A., "Preventing of SYN Flood Attack with Iptables Firewall", in *Communication Software and Networks, 2010. ICCSN '10. Second International Conference*, Singapore, February 2010.

[11] Linux Diagnostic Tools – System Utilities based on Sysfs. [Online] Available: http://linux-diag.sourceforge.net/Sysfsutils.html. [Accessed: June 2010].

[12] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In Proceedings of the SIGCOMM '87 Conference, Stowe, Vermont, August 1987.

[13] V. Jacobson. "Congestion avoidance and control". In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[14] Uppsala University CoRe Group. Aodv-uu. [Online] Available: http://core.it.uu.se/AdHoc/AodvUUImpl. [Accessed: Aug. 2010].