Making Use of All the Networks Around Us: A Case Study in Android

Kok-Kiong Yap Te-Yuan Huang Masayoshi Kobayashi† Yiannis Yiakoumis
Nick McKeown Sachin Katti Guru Parulkar
Stanford University NEC†
{yapkke,huangty,mkobaya1,yiannisy,nickm,skatti,parulkar}@stanford.edu

ABSTRACT

Poor connectivity is common when we use wireless networks on the go. A natural way to tackle the problem is to take advantage of the multiple network interfaces on our mobile devices, and use all the networks around us. Using multiple networks at a time makes makes possible faster connections, seamless connectivity and potentially lower usage charges. The goal of this paper is to explore how to make use of all the networks with today's technology. Specifically, we prototyped a solution on an Android phone. Using our prototype, we demonstrate the benefits (and difficulties) of using multiple networks at the same time.

Categories and Subject Descriptors

C.2.0 [**Computer Systems Organization**]: Computer-Communication Networks—*General*

General Terms

Design, Experimentation, Performance

Keywords

Mobile Internet, Open vSwitch, Android

1. INTRODUCTION

Poor connectivity is common when using wireless networks on the go. Connectivity comes and goes, throughput varies, latencies can be extremely unpredictable, and failures are frequent. Industry reports that demand is growing faster than wireless capacity, and the wireless crunch will continue for some time to come [2, 10]. Yet users expect to run increasingly rich and demanding applications on their smart-phones, such as video streaming, anywhere-anytime access to their personal files, and online gaming; all of which depend on connectivity to the cloud over unpredictable wireless networks. Given the mismatch between user expectations and wireless network characteristics, users will continue to be frustrated with application performance on their mobile computing devices.

The problem is often attributed to a shortage of wireless capacity or spectrum. This is not entirely true. Today, if we stand in the middle of a city, we can likely "see" multiple cellular and WiFi networks. But, frustratingly, this capacity and infrastructure is not available to us. Our contracts with cellular companies restrict access to other networks; most private WiFi networks require authentication, and are effectively inaccessible to us. Although we are often surrounded by abundant wireless capacity, almost all is off-limits. This is not good for us, and it is not good for network owners: Their network might have lots of spare capacity, even though a paying customer is close-by.

We believe users should be able to travel in a rich field of wireless networks with access to all wireless infrastructure around them, leading to a competitive market-place with lower-cost connectivity and broader coverage. In the extreme, if all barriers to fluidity can be removed, users could connect to multiple networks at the same time, opening up enormous capacity and coverage.

The good news is that smart-phones will be armed with multiple radios capable of connecting to several networks at the same time. Whereas today's phones commonly have four or five radios (e.g. 3G, 4G, WiFi, Bluetooth), in future they will have more. Shrinking geometries and energy-efficient circuit design will lead to mobile devices with more radios/antennas; a mobile device will talk to multiple APs at the same time for improved capacity, coverage and seamless handover.

If a smart-phone can take advantage of multiple wireless networks at the same time, then the user can experience:

Seamless connectivity: by using the best current network, and allowing the client to choose which network to connect to dynamically,

Faster connections: by stitching together flows over multiple networks,

Lower usage charges: by choosing to use the most cost-effective network that meets the application needs,

Lower energy: by using the network with the current lowest energy-usage per byte.

In our vision, intelligent and autonomous mobile devices will hunt the vicinity to find the best radio networks, and will choose which one(s) to connect to so as to best meet the user's needs. Key to our vision is the notion that control rests with the client (the user and the smart-phone): The network and the mobile client software will provide information about the presence, performance and price of different

CellNet'12, August 13, 2012, Helsinki, Finland.

networks; the client and the applications decide which one(s) to use. Beyond barriers due to business reasons, there are technical challenges.

Our vision requires much more than just multiple radios and multiple networks—it requires that the mobile client (as well as the applications and user) can take advantage of them. Today's clients are ill-equipped to do so, having grown up in an era of TCP connections bound to a single physical network connection. This leads to several well-known shortcomings: (1) An ongoing connection oriented flow—like TCP—cannot easily be handed over to a new interface, without re-establishing state; (2) If multiple network interfaces are available, an application cannot take advantage of them to get higher throughput; at best it can use the fastest connection available; (3) A user cannot easily and dynamically choose interfaces at fine granularity so as to minimize loss, delay, power consumption, or usage charges.

The three limitations are not just consequences of TCP. They are manifestations of the way the network stack is implemented in the operating system of the mobile device today. Therefore, as a step towards our bigger vision, we want to understand what changes are needed in the mobile device networking stack, to overcome these three limitations. In this paper, we describe how we refactored and then modified the networking stack on Android and Linux devices to be able to use multiple network interfaces simultaneously, and then we measure the performance of several experiments where several network interfaces are used.

Our first prototype, reported here, is purely host-based. The sending host decides which interfaces to use, and then divides outgoing traffic over multiple interfaces. In some cases we assume the receiver is also equipped with the same networking stack, so it can reconstitute the original flow. However, in this paper we assume the rest of the network infrastructure is unchanged. We plan to explore the benefits of coordination between the end host and the infrastructure in future work.

2. SYSTEM DESCRIPTION

We will now describe how we modified the Android and Linux operating system to allow a mobile device to use multiple interfaces. For our prototype, we had four high-level requirements: (1) it should run on commercially available smartphone devices and laptops, (2) it should work with unmodified existing applications, (3) it should connect to existing production WiFi and cellular networks, and (4) wherever possible, it should reuse existing well-supported software components.

2.1 Prototype

Our prototype consists of the following components shown in Fig. 1:

2.1.1 Android/Linux

The first problem to solve is that, by default, Android only allows one network interface to be active at a time—clearly no good for us. Android chooses which interface to use according to a preference order: If the device is connected to a WiFi network, Android automatically disconnects from WiMAX. We therefore modified the Connectivity Service in Android to allow us to use multiple interfaces simultaneously.

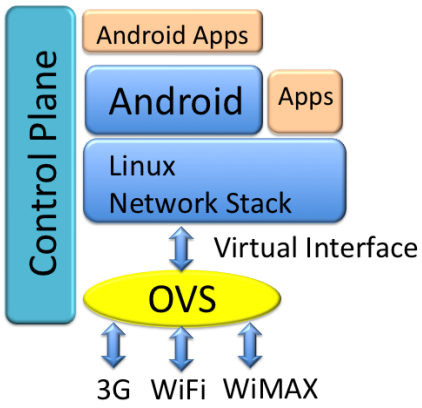


Figure 1: System diagram illustrating the main features of our prototype

Next, we need to spread traffic from one application over multiple interfaces. The application sends traffic using one IP source address; the networking stack takes care of spreading the traffic over several interfaces, each with its own IP address. We do this using a virtual Ethernet interface to connect the application, with its local IP address, to a special gateway inside the Linux kernel. The gateway stitches multiple interfaces together, without the application knowing. Essentially, the gateway is a traffic load-balancer that demultiplexes flows using Open vSwitch (see below), with appropriate changes made to the routing table and ARP tables. In this way, the application flow is decoupled from the IP addresses on each interface, which allows the set of interfaces to change dynamically as connectivity comes and goes. We further illustrate this setup using our experiment described in section 3.1.

Android is based on a minimal Linux kernel which is missing several tools and kernel modules we need (e.g. the kernel module for virtual Ethernet interfaces). We added the modules and cross-compiled common utilities such as `ifconfig`, `route` and `ip`.

2.1.2 Open vSwitch (OVS)

Open vSwitch (OVS) replaces the bridging code in Linux,[1] and lets us dynamically change how each flow is routed. OVS has an OpenFlow interface and therefore we can use `<match,action>` flow-table entries to easily route, re-route and handover existing connections.

We run OVS in kernel space, and ported it to Android by patching and and cross-compiling its kernel module and user-space control programs using Android Native Development Kit (NDK) for the ARM or OMAP processors.[2]

2.1.3 Control Plane

We control how flows are routed and re-routed using a small custom-built control plane, that interfaces to OVS using the OpenFlow protocol. In our prototype, the control plane is on the mobile device; but in principle the control

[1]OVS was recently upstreamed to Linux kernel 3.3 [6].

[2]Our patches and instructions are publicly available at `https://docs.google.com/document/pub?id=1k5jAkz_R475OhjOOaJdWwSpAw6mmR2Mp_Ggr8_yrXsY`.

(a) Android smartphones. (b) Laptop with ten interfaces.

Figure 2: Devices running our prototype network stack.

plane can be anywhere—for example, it could be run by the network operator, or outsourced to a third party provider.

Our control plane runs as an Android background service, and applications can interact with the control plane using Android IPCs [1]. This control plane controls OVS using the OpenFlow protocol running over a TCP socket. It controls the network interfaces (and other local resources) through system calls (e.g., Android Runtime Process). The control plane can also communicate with control planes on other hosts using JSON messages, allowing it to negotiate how flows are spread across interfaces.

2.2 Hardware

Our prototype runs on four common mobile devices (three smartphones running Android, and a laptop running Linux), shown in Fig. 2:

Smartphone: Motorola Droid with TI OMAP processor (600 MHz) and 256 MB of RAM, CDMA with Verizon 3G data plan, running Android Gingerbread 2.3.3.

Smartphone: Nexus One with Qualcomm ARM processor (1 GHz) and 512 MB of RAM, GSM, HSDPA with T-Mobile 3G data plan, running Android Gingerbread 2.3.3.

Smartphone: Nexus S 4G with Cortex ARM processor (1 GHz) and 512 MB of RAM, CDMA, WiMAX with Sprint 3G/WiMAX data plan, running Android Gingerbread 2.3.5.[3]

Laptop: Dell with AMD Phenom II P920 quad-core processor (3.2 GHz) and 4 GB memory, installed with Ubuntu 10.04.

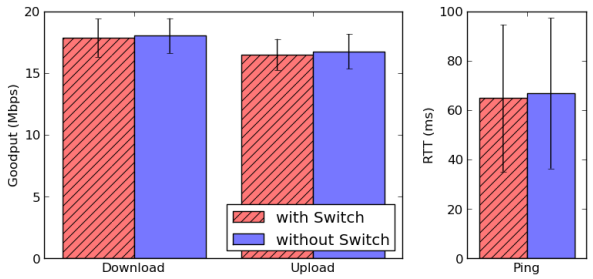
Where possible, we run experiments on the mobile phones, but sometimes it is infeasible (e.g. in one experiment we used ten interfaces, which is too many for current smart phones). Our experiments also communicate with peer servers and middleboxes, for which we used servers running Ubuntu 11.04.

2.3 Overhead Benchmarks

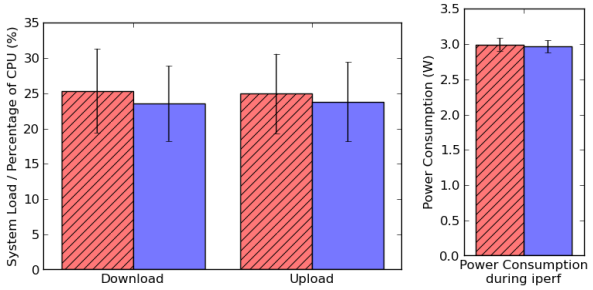
Our prototype adds functionality to Android, and inevitably consumes more power, more CPU cycles, and potentially reduces the maximum throughput. We designed the system to have minimal overhead, which is confirmed by our first set of experiments.

Throughput Reduction: We measured the goodput for ten `iperfs` with and without OVS. To maximize the cost of the overhead, we used the Motorola Droid, the least provisioned

[3]Android 2.3.5 includes an important fix to WiMAX driver.



(a) Throughput benchmark (b) RTT benchmark



(c) System Load benchmark (d) Power benchmark

Figure 3: Overhead of Switch Datapath

Android device we have. Fig. 3(a) shows that the goodput is reduced by no more than 2%.

RTT Increase: Similarly, we profiled the delay incurred by sending 300 `pings` with and without OVS. There is no observable increase (in Fig. 3(b)) in round trip time.

CPU Load: The CPU load is logged while running `iperf` on the Droid with and without OVS. Fig. 3(c) shows that the CPU load is increased by 1.8%.

Power Consumption: To measure our prototype's impact on power consumption, we removed the battery and powered the Droid via its USB port and a power monitor. Fig. 3(d) shows negligible power increase with OVS.

Several of the authors use the prototype daily, and it has proved robust so far. Going forward, we intend to support it for others to use, and enable others to build their research prototypes on top.

3. EXPERIMENTS

The goal of our prototype is to overcome the three problems listed in the introduction, namely (1) an ongoing connection cannot easily be handed over to a new interface without re-establishing state; (2) if multiple network interfaces are available, an application cannot take advantage of them to get higher throughput (3) a user cannot easily choose interfaces so as to minimize power consumption, or usage charges.

To evaluate how well our prototype solves these problems, we ran the three experiments described below. Our experiments assume we have no special control of the network (we use existing WiFi and cellular networks), and the clients communicate with unmodified peers (except other-

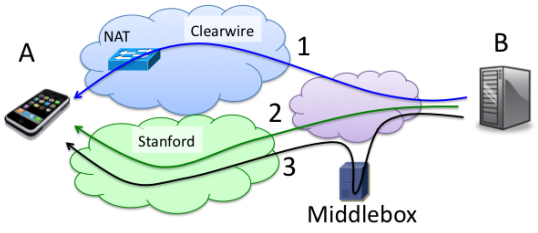


Figure 4: Diagram showing the routes of the flow at each stage of the experiment.

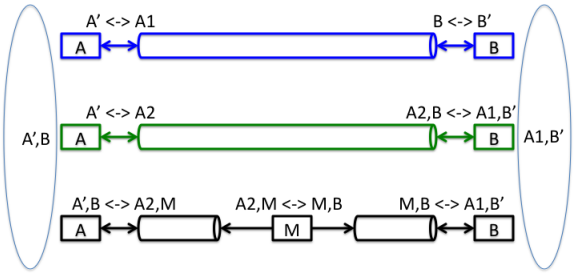


Figure 5: Diagram showing address translation happening along the routes of each flow at each stage of the experiment.

wise noted). Hence our experiments also validate the degree to which our prototype is backward compatible.

3.1 Seamless Connectivity

We begin with a simple testing to show how the system maintains a HTTP connection across a migration. Our model is a user arriving to work who wishes to migrate an ongoing video stream from a public WiMAX network to a corporate WiFi network.

In this experiment, both the client and peer are running our prototype stack (i.e., with OVS and a custom control plane). During the migration, the client's IP address will change. This change has to be coordinated with the peer for a seamless migration through control packets between the OpenFlow controllers. The control packet signals the impending migration of an ongoing flow to the peer, which can be done without aid from the network. The peer would then rewrite the addresses of the subsequently incoming packets such that the migration of the flow is transparent to the application above.

Several possibilities exists in this design space. In our implementation, we rewrote the source address to that of the initially established flow (as shown Fig. 5). At any point in time, the application in host A thinks that the communication is between addresses A' and B while the application in host B thinks that the communication is between addresses A1 and B'. The consistent views of the applications in the end hosts is maintained by the translations indicated in Fig. 5. Another possible implementation is to always rewrite the source address to one that is arbitrarily picked at the onset of the flow.

Fig. 6 shows the throughput of the session as we migrate the flow (as shown in Fig. 4). Initially the flow is routed through WiMAX; then after 30 seconds we migrate

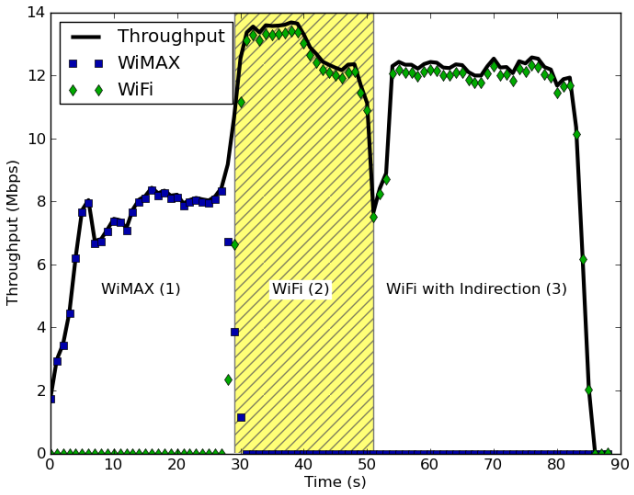


Figure 6: Throughput of mobile during the experiment.

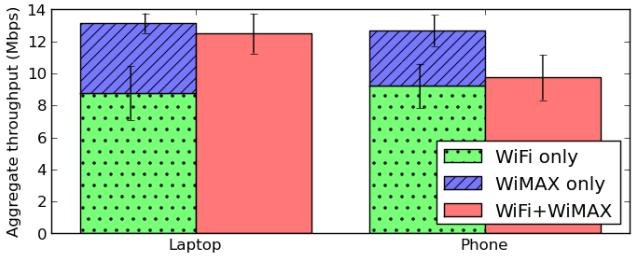


Figure 7: Stitching two networks: Steady state throughput.

it to WiFi. The control plane decides when to make the move, and reconfigures OVS to change the addresses, rewrite packet headers, and switch packets to/from the new interface. This change is coordinated with the control plane of the peer. The result is an uninterrupted TCP flow that has been migrated from one network to another without re-establishing state.

To show the flexibility of our system, we also tested a very different migration mechanism, as described by Stoica *et al* using *I3* [18]. The flow is routed through an off-path middlebox (or waypoint); each end communicates only with the middlebox. This could be used, for example, to insert a firewall or DPI box in a corporate environment. In our experiment, the migration takes place at 50 seconds, with a brief drop in data rate while packets reach the middlebox.

The experiment shows that our setup is quite powerful: Both migrations were done without changing the network. Usually, migration and mobility are considered fixed functions of the infrastructure [16, 18].

3.2 Stitching Networks for Throughput

Our prototype allows multiple networks to be used simultaneously. To test how well this works, we streamed data while varying the number of interfaces, and measured the throughput seen by the application.

In the experiment, we download a 100 megabyte file using five parallel TCP connections using `aria2c`. First, we ran all

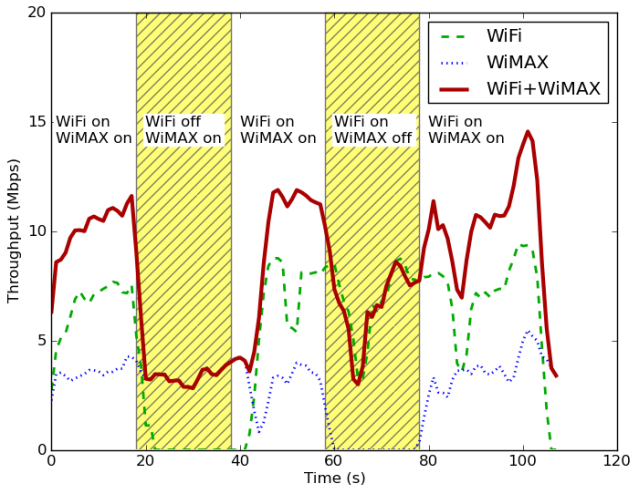


Figure 8: Stitching two networks: Throughput when downloading a 100 MB file. WiFi is turned off from 20s to 40s, and WiMAX from 60s to 80s.

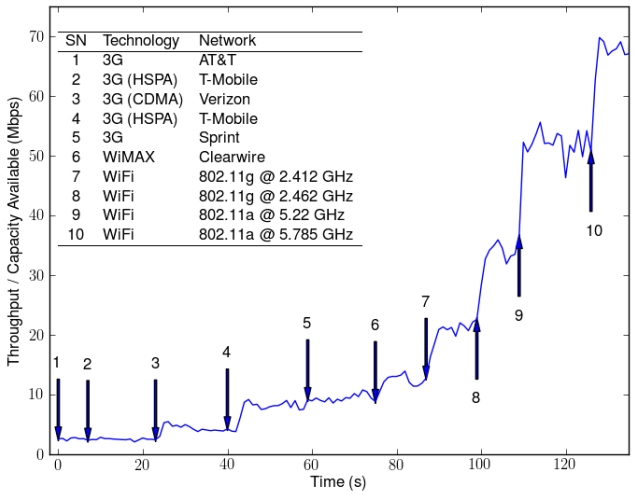


Figure 9: Connecting a laptop to ten wireless networks. The data-rate increases as more networks are added (in the order listed in the figure). The arrows show when each interface is turned on.

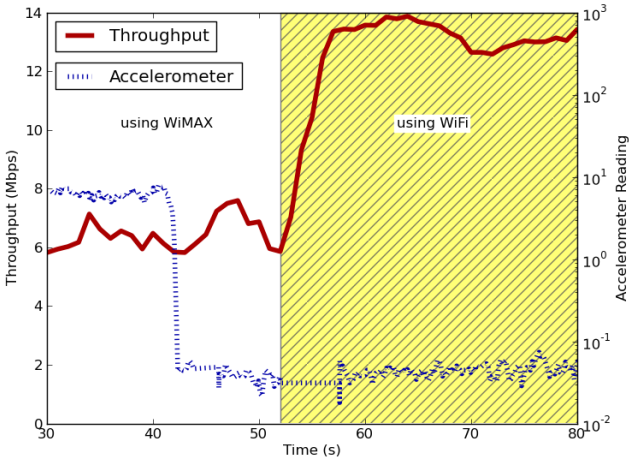


Figure 10: Moving an ongoing flow from WiMAX to WiFi when device stops moving.

five TCP connections over our campus WiFi network; then we used Clearwire's WiMAX network. Finally, the control plane stitched both networks together. We ran each test ten times on two clients (the laptop, and on the Nexus S 4G smartphone), and report the average throughput.

Fig. 7 shows the average aggregate throughput with and without stitching. The laptop achieves 95% of the aggregate data-rate, whereas the smartphone achieves 77%. Further investigation revealed that there is interference between the WiFi and WiMAX interface in the mobile phone, because the transceivers are close together. There is no fundamental reason why this can not be solved by better shielding—something we can expect if stitching becomes common.

Stitching interfaces together also helps maintain connectivity during times of packet loss or complete network outage, as Fig. 8 shows. Each interface was turned off for 20s during the experiment; connectivity was maintained because of the other interface.

Finally, to push the limits of stitching, we stitch ten networking interfaces together (!). The ten networks are listed in Fig. 9, and include four different wireless technologies: 3G, WiMAX, WiFi 802.11a (5 GHz), and WiFi 802.11g (2.4 GHz); and include six different production networks. We had to use the laptop, because there was no way to attach so many interfaces to a smartphone. To measure the capacity brought by each successive interface, we gradually bring up one interface at a time. The control plane stitches it to the others to increase the data-rate. Fig. 9 shows the throughput rising as each interface is added (in the same order as Fig. 9), up to a maximum of 70 Mbps (more than three times the fastest interface).

3.3 Dynamic Choice of Network

Our final experiment (inspired by [13]), shows how the user or application can choose which network to use. In our experiment, we use the phone's accelerometer to tell if the device is moving. When the user is moving, we connect it to WiMAX for greater coverage; when stationary, we connect it to the free and faster WiFi network (Fig. 10).

Because the decision is made by the user (or client), we can expect faster innovations to be designed and made available in the future, for example methods described in [8, 13, 14].

4. RELATED WORK

There is a large body of work that seeks to exploit the diversity of connectivity options in mobile wireless networks [4, 5, 12]. This work is also related to recent work on multipath transport protocols [9, 11, 15]. While MPTCP provides bandwidth aggregation, similar transport protocol optimizations such as TCP Migrate [17] provide the ability to handover a TCP connection to a new physical path without breaking the application. Our work is orthogonal to these techniques, our goal is to provide an implementation that can accommodate these (and other) extensions.

Our work is also related to a number of recent optimizations to improve wireless network performance, some of which leverage sensors [13], others exploit geolocation informa-

tion [8], while some leverage user-specified application policies [3]. Finally, recent work [7] has noted the prevalence of multiple-SIM phones in countries such as India and proposed modifications to the cellular network infrastructure to enables clients to make better network choices. Similarly techniques such as FatVap [12] aggregate bandwidth from multiple neighboring APs to build a faster connection. Our work compliments these techniques by providing flexibility at the client to take advantage of these innovations.

5. DISCUSSION

Our prototype using Android and Open vSwitch, is able to achieve the following: (1) handover an ongoing TCP connection without re-establishing state; (2) stitch multiple interfaces together for higher throughput; (3) dynamically choose interfaces to minimize loss, delay, power consumption or usage charges. This demonstrates that a refactored client network stack can achieve a lot of our goals without modifying the fixed infrastructure.

However, current networks and devices do not make it easy:

Address ambiguity : A client might have two interfaces connected to different networks that use identical private address spaces, for example they might both use addresses starting from `192.168.0.0`. While we can send packets via gateways on both networks, to reach hosts directly attached to either networks requires us to distinguish them by some means other than IP address; e.g. by forwarding packets based on the interface they are destined to (if we know). Otherwise, one set of hosts will be unreachable.

Discovering connectivity : Discovery protocols (e.g. DNS and DHCP) are typically tied to a particular network interface, and therefore if we want to use multiple networks, we must keep track of the DNS and DHCP settings for each interface. And to find which networks are available, we must proactively ARP hosts and routers on each interface.

Middleboxes : Wireless networks—particularly cellular networks—are riddled with middleboxes, which can interfere with flow migration. For example, a migrating flow might be blocked if the new network did not see a SYN packet which we observed during our experiments.

Interfaces : Sometimes, a single network requires different header formats depending on the physical device. For example, a 3G network requires Nexus One (using the Qualcomm MSM 3G chipset) to present a virtual Ethernet interface, whereas they are presented as IP interfaces on Google Nexus S and the Sierra 3G USB Dongle. Different interfaces also present different MTU to the network stack, e.g. 3G and Ethernet interfaces typically has MTU of 1400 and 1500 bytes respectively.[4] These are not limitations of the approach, because it is possible to rewrite the header format arbitrarily for each interface and fragment the packet accordingly.

To solve the problem of ambiguous private addresses will take more work. Hopefully cellular providers will, in time, fix the middlebox problem. The advantage of our refactored client networking stack—where changes can easily be added to OVS or to the control plane—it should be quite straightforward to add solutions to these, and as-yet unidentified problems down the road. Indeed, we believe our approach makes innovation and experimentation very straightforward

[4]We set the MTU to the minimum of all interfaces in our prototype to work around this problem.

6. ACKNOWLEDGEMENT

This research is supported in part by the NSF POMI (Programmable Open Mobile Internet) 2020 Expedition Grant 0832820, and ONRC (Open Networking Research Center). The authors also like to thank Monica Lam, Yongqiang Liu, and Adam Covington for their time, effort and advice in the early stages of this project.

7. REFERENCES

[1] Intents and Intent Filters. `http://developer.android.com/guide/topics/intents/intents-filters.html`.

[2] C. Albanesius. AT&T: Give us spectrum, not rules. `http://www.pcmag.com/article2/0,2817,2361708,00.asp`.

[3] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3G using WiFi. In *Proc. ACM MobiSys '10*, Jun. 2010.

[4] C. Carter, R. Kravets, and J. Tourrilhes. Contact networking: a localized mobility system. In *Proc. ACM MobiSys '03*, May 2003.

[5] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *Proc. IEEE INFOCOM 2004*.

[6] J. Corbet. Routing open vswitch into the mainline. `https://lwn.net/Articles/469775/`.

[7] S. Deb, K. Nagaraj, and V. Srinivasan. MOTA: engineering an operator agnostic mobile service. In *Proc. ACM MobiCom '11*, 2011.

[8] J. Eriksson, H. Balakrishnan, and S. Madden. Cabernet: Vehicular Content Delivery Using WiFi. In *Proc. ACM MOBICOM*, Sep. 2008.

[9] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. RFC 6182 (Informational).

[10] V. Godinez. Wireless carriers grapple with spectrum shortage. `http://www.thenewstribune.com/2011/04/24/1638191/wireless-carriers-grapple-with.html`.

[11] H.-Y. Hsieh and R. Sivakumar. pTCP: an end-to-end transport layer protocol for striped connections. In *Proc. IEEE ICNP 2002*, Nov. 2002.

[12] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: aggregating AP backhaul capacity to maximize throughput. In *Proc. USENIX NSDI '08*, Apr. 2008.

[13] H. B. Lenin Ravindranath, Calvin Newport and S. Madden. Improving wireless network performance using sensor hints. In *Proc. USENIX NSDI '11*.

[14] A. J. Nicholson and B. D. Noble. BreadCrumbs: forecasting mobile connectivity. In *Proc. ACM MobiCom '08*, Sep. 2008.

[15] L. Ong and J. Yoakum. RFC 3286 (Informational).

[16] C. Perkins. RFC 5944 (Proposed Standard).

[17] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *ACM MobiCom '00*.

[18] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. ACM SIGCOMM '02*, Aug. 2002.