MIC: Enabling Efficient Concurrent Use of Multiple Network Interfaces on Mobile Systems

Varun Anand, Taeyeon Ki, Karthik Dantu, Steven Y. Ko, Dimitrios Koutsonikolas
University at Buffalo, The State University of New York
{varunana, tki, kdantu, stevko, dimitrio}@buffalo.edu

***Abstract*—Mobile devices such as smartphones and tablets are provisioned with multiple network interfaces capable of data connectivity such as the cellular radio, WiFi, bluetooth, and others. However, restrictions from the carrier side allow us to use only one of the interfaces at any given time, and not allow us to fully utilize all the data links in a mobile device's arsenal. We propose MIC, an architecture to enable concurrent, efficient, *multi-interface connectivity*. MIC begins with the observation that a large fraction of mobile traffic is HTTP [3], [4]. By modifying the HTTP support libraries in the Android Framework, MIC provides the ability to utilize multiple interfaces for the same connection while maintaining the application as well as OS semantics. This allows for users to re-flash the system software on their phones to enable multi-interface connectivity and use it in a seamless manner without changing the apps or need any modifications to the server.**

I. Introduction

Technology has enabled seamless access to information through the Internet. Recently, the proliferation of mobile devices such as smartphones and tablets has extended this access outside our homes and offices. Such ubiquitous access to information has enabled many applications such as turn-by-turn navigation, location-centric information such as local restaurants and coffee shops, anytime access to email and social media, as well as numerous others. Nowadays, such access is assumed to be a given, and users expect a smooth and seamless experience. Such access could be simple text queries to fairly bandwidth-intensive applications such as video and audio streaming. Most of today's mobile devices also have multiple network interfaces to for such connectivity on the go such as Wi-Fi, cellular radio, Bluetooth, NFC, and others. One way to improve information access is by enabling and using multiple available network interfaces concurrently.

Let us look at a concrete example. Imagine Alice leaving her office after work. Her daily routine involves listening to her favorite podcasts using her mobile device on her drive home. However, she always forgets to download them in advance. On her way out, she starts downloading few of these podcasts. However, the moment she steps out of the building, her Wi-Fi signal is lost. This disrupts the previous download, requiring her to restart the download on her mobile (3G/4G) data connection, re-download the portions already downloaded through Wi-Fi. This is both a waste of her cellular data connection, and delays the download of Alice's podcasts significantly. She might swing by the local coffee shop for a quick coffee on the way home. Her mobile device might connect to the Wi-Fi available at that coffee shop causing another disruption and potential restart of downloads. The ideal solution for Alice's problem would be to make use of both network interfaces, or a subset of the available network resources as per Alice's choice. That way, when one of the connections is lost, the download could continue in a seamless manner using the other available ones.

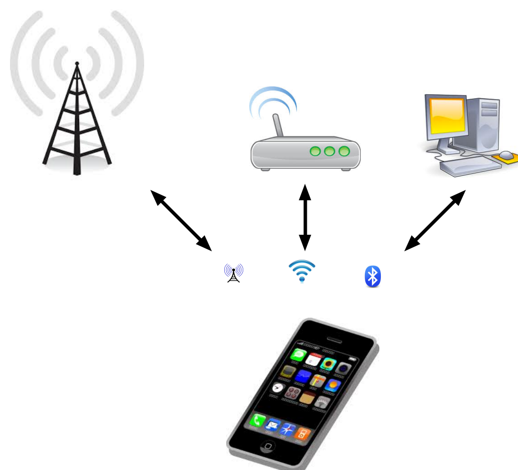

Fig. 1. Modern mobile systems have multiple network interfaces capable of having concurrent data connections

Modern smartphones and tablets allow the use of only one interface at any given time. This might simplify the implementation of network functionality but affects user experience significantly. Current mobile devices utilize the available network interfaces in a prioritized order. If Wi-Fi is available, then the cellular radio data connection is disabled. This is what Alice observes when she enters the local coffee shop. Such a transition requires enabling data connectivity on the new interface as well as updating of network properties all of which could take several seconds to minutes. This delay is especially disruptive when downloading large files, and for latency-sensitive flows such as streaming media. During such a download, any interruption due to change in network conditions will cause it to fail, which then requires the download to be restarted and loss of content previously downloaded. This is highly undesirable and can be avoided by intelligent use of all available network interfaces.

The primary goal of this work is to enable concurrent connectivity across multiple available network interfaces in a seamless and efficient manner on mobile devices. For better adaption, we want to restrict our modifications to the mobile

device only. Further, we would like to restrict our changes to the system software on the mobile device in order to seamlessly support existing apps. Our implementation is on the Android platform. However, our modifications are solely based on the HTTP protocol, and we believe that they can be seamlessly transferred to other platforms as well.

The rest of the paper is as follows. In Section II we will describe other efforts to achieve concurrent use of multiple network interfaces. Section III will provide an overview of our design, some statistics on what Android apps use today, and how our design affects that landscape. Section IV describes our modifications in Android to accomplish concurrent multi-interface connectivity. Section V will describe our results along with our design modifications to improve the performance of using multiple network interfaces concurrently. We conclude with thoughts on future work, and extensions to our system.

II. Related Work

There has been a substantial amount of work done in concurrent connectivity over multiple interfaces. We can classify previous approaches based on the layer of network stack at which the modifications are made.

A. Link Layer

Yap et al. [14] make use of Open vSwitch [1] to control the TCP flows over multiple interfaces. Open vSwitch acts as a bridge that spreads traffic from one application over multiple interfaces. In their approach, an application sends traffic by connecting to the virtual ethernet interface that stitches multiple interfaces together. The networking stack takes care of spreading the traffic over multiple interfaces. This approach differs from our approach in terms of multiplexing that is carried out at layer 2, whereas we do it at the HTTP level by making use of its implicit byte-range support feature.

B. Transport Layer

Multipath TCP uses multiple interfaces for a single connection by modifying the TCP protocol [10]. An MPTCP (Multipath-TCP) connection contains one or more subflows that appear(s) like a regular TCP connection. MPTCP options are included in the SYN packet to verify that the end-server is MPTCP capable. An MPTCP-capable server splits the data stream among all the established subflows. An MPTCP connection thus makes use of multiple interfaces and provides seamless usage of different network interfaces. However, MPTCP requires explicit support from both the client and the server. While it is gaining popularity, only a small fraction of modern servers provide this support making its applicability limited.

Delphi [2] is a transport module chooses the best network interface based on certain policies, active probing, and information exchange between peers. It comprises of components to predict properties of each network connection by learning from easy-to-measure observations, property sharing between different nodes and a selector that makes the best network choice at each node. The application specifies its intent to Delphi, which then selects an interface at the beginning and does not change it throughout the transfer. In our implementation, we make use of more than one interface for a single connection, which in turn provides seamlessness and flow switch between multiple interfaces during data transfer. This work could be thought to complementary to our effort, and can be utilized in deriving efficient policies for using our multi-interface connectivity support.

Nika et al. [9] study the potential gains of using multiple interfaces such as LTE and WiFi along with MPTCP. Their finding is that MPTCP achieves only a fraction of the maximum possible gain and at the same time, it consumes much energy. They study this by performing measurements at 63 outdoor locations in 5 US cities.

C. Socket Layer

Socket Intents [12] modify the Socket API to enable the application express its communication preferences such as high bandwidth, low delay, connection resilience etc. It uses this information to select the appropriate interface, tune the network parameters, or even combine multiple interfaces. They do not make use of multiple interfaces as they choose one of the interfaces based on the application's preferences before establishing the connection. Their work relies on the application to provide initial information about the kind of work it is going to perform. Therefore, this implementation does not make use of all the available interfaces for a single download and hence cannot be used to provide seamlessness. Moreover, it requires existing applications to modify their implementation as per the modified interface.

D. Application Layer

mHTTP [8] is multi-source, multi-path HTTP that makes use of more than one network interfaces and end-servers to request data in chunks by leveraging byte-range support provided by HTTP protocol. This implementation makes use of multiple source servers for requesting data. Our implementation differs from theirs in terms of efficient use of the available bandwidth. They use fixed-size chunking for requesting data, which esentially causes under-utilization of the pipe capacity between the client and the server. We overcome these difficulties by introducing new concepts such as Dynamic-chunking combined with request pipelining to remove the round-trip delays.

Rahmati et al. [11] target seamlessly migrating TCP flows from one network interface to the other. To achieve this, they enable multiple interfaces and monitor those interfaces, based on which they select a primary interface and switch all the flows through this primary interface by using the routing tables. Their implementation selects a primary interface that is used by all the applications system-wide and thus does not make use of all the available interfaces.

Intentional networking [6] relies on the application's intent that provides information about the data to request. Applications declare labels which defines the importance and

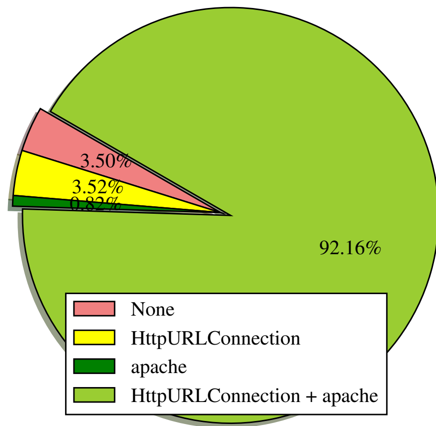

Fig. 2. **Http support library usage through 1134 applications downloaded on Jan 29, 2016.**

Library Name	Http Libraries		Applications
comscore	HttpURLConnection apache	&	112 (11.47%)
freewheel	apache		35 (3.59%)
conviva	apache		21 (2.15%)
adobe pass	apache		19 (1.95%)
analytics	HttpURLConnection apache	&	86 (8.81%)
volley	HttpURLConnection apache	&	215 (22.03%)
retrofit	HttpURLConnection apache	&	192 (19.67%)
crashlytics	HttpURLConnection		283 (29%)
not used			13 (1.33%)

TABLE I

THIRD PARTY LIBRARY USAGE OF POPULAR APPLICATIONS. THIS REULT IS MEASURED THROUGH 976 APPS THAT USE BOTH HTTPURLCONNECTION AND APACHE LIBRARIES.

the urgency of the requested data. For instance, one such label differentiates between foreground and background traffic. Based on these labels appropriate network interface is selected. They also define certain ordering contraints (based on the urgency) which requires sending additional data from the client end. This require server changes to understand the additional data. In contrast, our implementation makes changes at the client end.

FloMiS [5] is a middlebox approach to migrating HTTP connections. It is designed to be used in a multi-homed gateway router for public transit vehicles such as city buses, and seamlessly migrates HTTP flows depending on the conditions of various links. It enables the flow migration by rewriting packets and leveraging HTTP range requests.

III. DESIGN OVERVIEW

As described in Section II, splitting a single logical connection across multiple interfaces is a problem of great interest. Fundamentally, it requires the two ends of that connection to agree that two separate physical connections are part of the same logical connection. Solutions such as Multipath-TCP achieve this by requiring modifications to the server as well as the client. One of our chief design goals is to restrict all modifications to the client. This make implementing our solution hard at the transport layer. We choose to enable multi-interface functionality at the application layer, with the use of the HTTP protocol in particular.

This design has several implications. Apps that use other forms of connectivity are not able to utilize our modifications. We have a detailed study on how many applications we can and cannot impact later. Restricting ourselves to a protocol allows us to exploit the features of that protocol. As will be described later, we use the byte-range request feature as well as the pipelining feature to improve connectivity. Another reason to use the HTTP protocol is because many apps use one of the available http libraries from Android for their connectivity.

Shown in Figure 2 is a distribution of library usage in apps in Android. Using Google Play API, we have downloaded the top 50 popular apps in each of the 26 categories available on Google Playstore. Our analysis is done through a Java optimization framework called *Soot* [13], and we are able to analyze 1134 free apps of the possible 1300 apps. Figure 2 shows that 3.52% applications use `HttpURLConnection` library exclusively, and 92.18% apps use both `HttpURLConnection` and Apache `HttpClient`. To verify analyze http usage, we analyze the http library used by third party libraries. Table I shows that many popular applications take advantage of third party libraries to manage network requests, bug reporting, and others. For example, Volley library helps to make networking for Android apps faster and easier. It is an open-source library written by Google and used in Android Open Source Platform. Volley library operates with both apache and HttpURLConnection libraries.

Given these observations, we choose to enable networking on multiple interfaces using the `HttpURLConnection` library. The rest of the paper will describe our design and modifications to this library. However, our changes can easily be implemented using any other http library as they use well-known http features from the standard, and are in no way specific to this library.

IV. ANDROID IMPLEMENTATION

We describe the components we implemented to enable concurrent connectivity across multiple interfaces in Android.

A. Network Interface Management

Android uses the Linux OS. Linux uses one route table by default, with rules for specific subnetworks, and a default gateway. This is true even when multiple interfaces are enabled. We are interested controlling the interface being used instead of using the default gateway. For this, we create a separate routing table per interface. We employ linux iptables for this. By doing this, and binding connections to a particular interface, we are able to ensure that the packets on that connection use that particular interface. Mobility tends to see large dynamics in connectivity. For this, we modified the Android

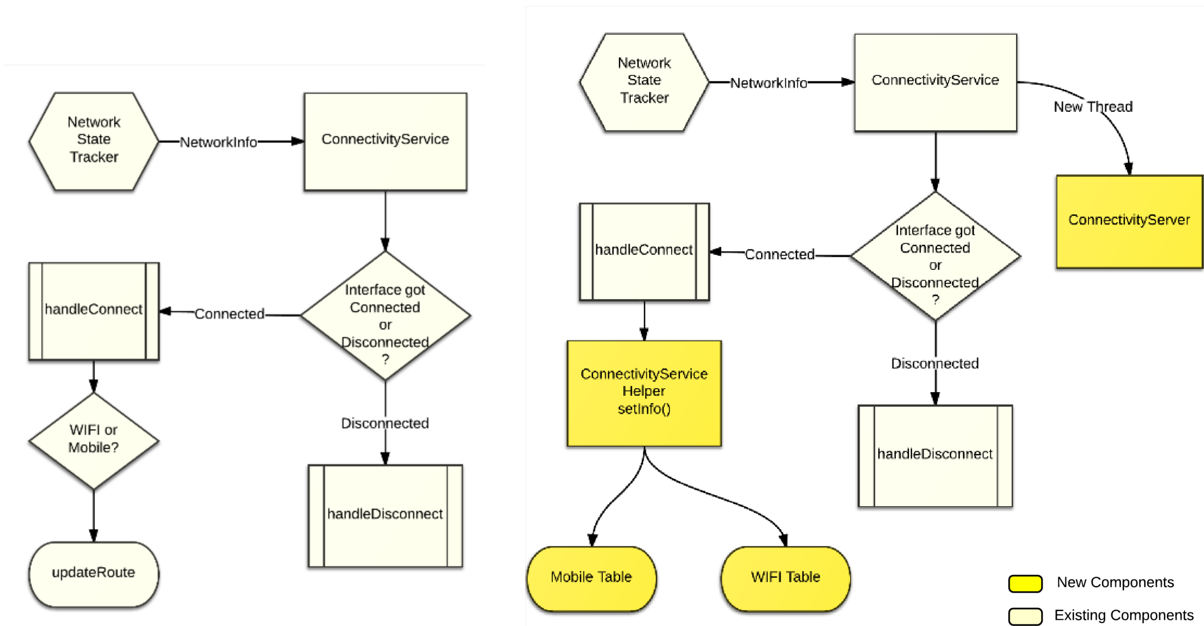

Fig. 3. Connectivity Service: Original (left) and Modified (right)

framework to constantly track the network interfaces that are currently available, and ensure that the appropriate routing tables are updated accordingly to ensure proper functionality. These modifications are described below.

B. Connectivity Management in Android Framework

`NetworkStateTracker` and `ConnectivityService` work together closely to manage the network connectivity of a device, as depicted in Figure 3(left). `NetworkStateTracker` is responsible for tracking any changes in the network state from the underlying kernel. This component maintains the state of an interface. Whenever an interface goes up or down, it notifies `ConnectivityService` module which then takes appropriate actions based on whether the interface is connected or disconnected. For example, when an interface gets connected, `ConnectivityService` updates DNS server entries and the kernel routing table.

We first disabled the priority-based activation/deactivation of network interfaces in the `handleConnect` module. This allows both the WiFi and the mobile interface to be enabled, and the routes for both interfaces are added to the main routing table. Since connectivity state changes frequently on a mobile device, we created a helper module (`ConnectivityServiceHelper`) in the Android framework to update the linux routing tables (Section IV-A) at runtime as well as when the corresponding interface is connected/disconnected. In addition to the new routing tables, we also update the main routing table. We maintain a single `default` route in the main routing table when more than one interface is connected. This is needed to maintain connectivity for additional services such as DNS name resolution.

C. Modifications to `HttpURLConnection`

`HttpUrlConnection` library internally maintains a connection pool that manages up to five open connections per URL. When an application wants to fetch data from the server, it creates an instance of this library and requests for an `InputStream`. The application then reads from this stream for the actual content. Figure 4(left) shows the control flow from the application to each component in this library.

This library takes care of preparing a request header and decoding the response type from a response header. This is taken care of by `HttpEngine` component of the library. There is one instance of `HttpEngine` per request-response pair. `HttpEngine` prepares the request header and then sends the request using the `HttpConnection` component. This component is an abstraction used to represent a http connection between client and server. Its underlying implementation holds the actual socket that is used to send and receive data.

Figure 4(right) shows our modifications. Firstly, we use multiple `HttpWorkers`, one per active network interface. This allows us to use one or more network interfaces concurrently. Secondly, we have a `ConnectionStatus` module that queries the `ConnectivityServer` regularly, and enables/disables the corresponding `HttpWorkers` based on the network interfaces available. Finally, the `HttpHelper` provides data structures to manage the download such as assembling out-of-order packets received, tracking outstanding requests etc.

V. Design Optimization

A. Request Decomposition

Our design allows a single request to be sent over different interfaces. In the current version of our implementation, we

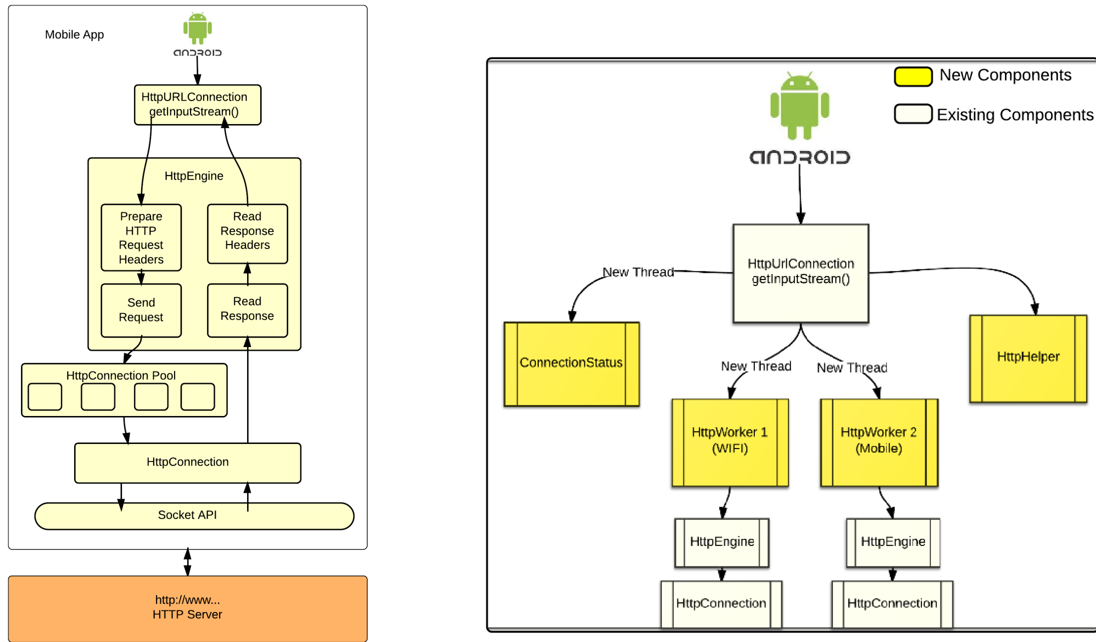

Fig. 4. HttpUrlConnection Library: Original Control Flow (left) and Modified Flow (right)

make use of two interfaces: WiFi and cellular (3G or 4G, whichever is available). Each interface is associated with a worker which represents that interface at an abstract level. These workers collaboratively download the entire file based on their availability. We maintain one connection (socket) per interface.

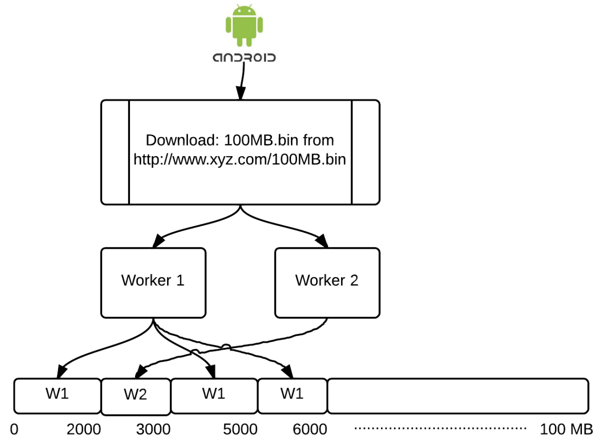

Fig. 5. Request Decomposition and Work Stealing

B. Download Scheduling

Figure 5 shows the pictorial representation of request decomposition and scheduling used in our implementation to handle requests. Instead of statically assigning different chunks of the file to each worker, we let the workers decide the chunks they want to download based on their availability. Requests are handled on a best-effort basis.

Buffering: The request occurs in parts, and can be downloaded out-of-order. This is because different interfaces may have different network parameters such as bandwidth, latency etc. We cannot return the chunks directly to the application, and need to maintain a buffer that returns the data to the application in order.

Adapting Chunk Size: The primary parameter to vary when requesting data in chunks is the chunk size. This parameter decides how much data each interface will request in a single transaction. The simple approach of using a fixed-size chunk suffers from an inherent problem. Slower interfaces become a bottleneck during the downloads since the faster interface downloads the same number of bytes faster. Consequently, even though the remaining portions of the file have been downloaded, the application has to wait for the slower interface to complete the download of its chunk. Our empirical observations show that network bandwidth is highly variable, especially on the cellular interface and the optimal chunk size is dependent on the current bandwidth and the timeliness expected by the application.

Based on this observation, we adapt the chunk size for each interface at run-time, based on the current state of the network. Specifically, the chunk size for the next request is calculated as:

$$Chunk_Size = Download_Speed * Wait_time \quad (1)$$

In Equation 1, $Download_Speed$ is the average speed of the last k downloaded packets and $Wait_time$ denotes the maximum amount of time we are willing to wait per request. We find that instantaneous bandwidth tends to vary quite a bit, and averaging over k packets helps smooth our estimate. For example, a $Wait_time$ of one second makes the chunk size equivalent to the current throughput experienced by that interface. With this modification, the slower interface is no longer the bottleneck as the download time is similar for all interfaces.

Figure 6 shows the effect of smoothing on the chunk size with the wait time of 1 second. We can clearly see that the chunk size variation is same as the variation in speed without the smoothing, whereas the variation is reduced with the smoothing.

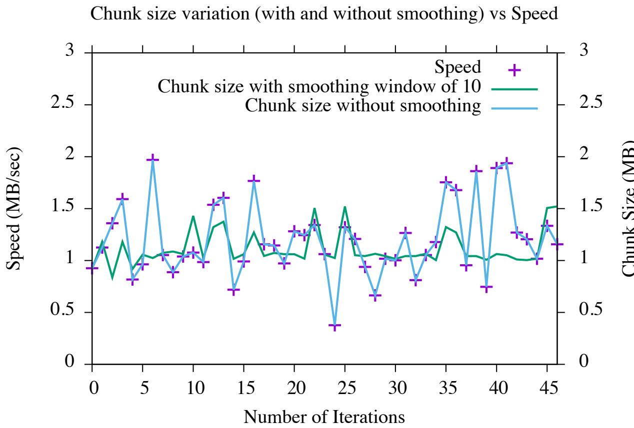

Fig. 6. Effect of smoothing on chunk-size with Wait-time = 1 second

C. Dynamic Chunking Performance

In this section, we compare the performance of our implementation with dynamic chunking against the default implementation over one interface. Figure 7 shows the download time with our implementation and the default implementation for a file of 100MB. For the default implementation, we ran over 100 trials. For our implementation, we ran 10 trials for a given value of $Wait_Time$, and varied $Wait_Time$ from 100 ms to 10 s. In both cases, we plot the average download time along with the variance.

In Figure 7, we observe that, although dynamic chunking allows us to estimate the observed latency on slower interfaces, there is still a performance overhead of about 20-30% in comparison to the default implementation. This problem is independent of the chunk size, and is inherent in the way we send requests. Specifically, there is a round-trip delay for every request made when the connection between the server and client is being under-utilized. For the default implementation, this happens once at the start of the download whereas our implementation incurs this overhead for every chunk downloaded.

Another interesting observation is the shape of the curve in Figure 7. We notice that the average download time

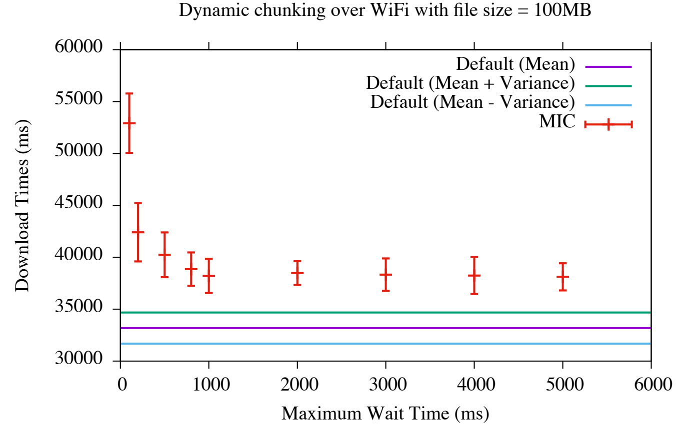

Fig. 7. Comparison of default implementation with our implementation with dynamic chunking

decreases with the increasing $Wait_Time$. As we saw in equation 1, the chunk size depends on the $Wait_Time$. If the network conditions remain constant, the chunk size becomes directly proportional to the $Wait_Time$. Therefore, given two $Wait_Time$ values `W1` and `W2`, such that `W1` is less than `W2`, we will have requests with smaller chunk size with $Wait_Time$ `W1` compared to the requests made with $Wait_Time$ W2. Smaller chunk size implies more requests and thus more delay caused by the round-trip time between two consecutive requests.

1) Pipelining: We saw in the previous section that requesting one chunk at a time introduces the round-trip delay between two consecutive requests sent over an interface. To overcome this, we added HTTP request pipelining to our implementation. Instead of one chunk at a time, we request for n chunks in parallel, thus maintaining atmost n outstanding requests at any given point of time. For each worker thread (per interface), we maintain a sender and a receiver which share a common socket and maintain a request queue. The sender sends the requests until the queue becomes full and then waits for the request queue size to decrease which happens when receiver reads the data corresponding to those requests. This way, we make use of HTTP pipelining to asynchronously send requests and receive corresponding responses over a single socket or connection. HTTP pipelining mandates that server must send responses in the same order of the requests received [7]. This enhancement overcomes the round trip delay by utilizing the connection between client and server. By issuing multiple requests in parallel, the client allows the server to utilize the connection efficiently.

2) Performance evaluation with request pipelining: Figure 8 shows our results for download over WiFi with request pipelining for 3 outstanding requests. We observe that our implementation now performs as well as the default implementation over a single interface thereby demonstrating that we have bridged the overhead that observed earlier (Figure 7). We also observe that the download time decreases with increase

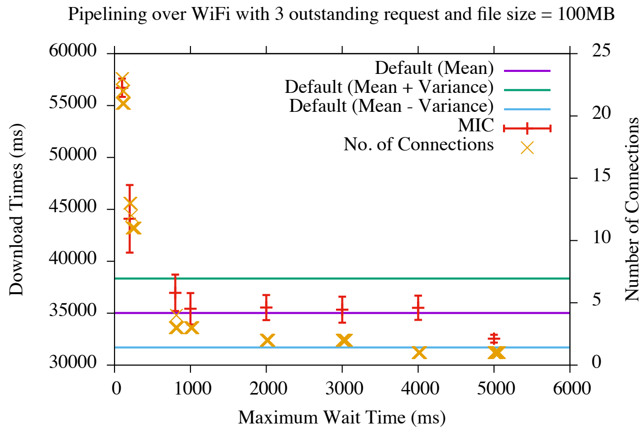

Fig. 8. Comparison of default implementation and our implementation with dynamic chunking, and request pipelining for 3 outstanding requests

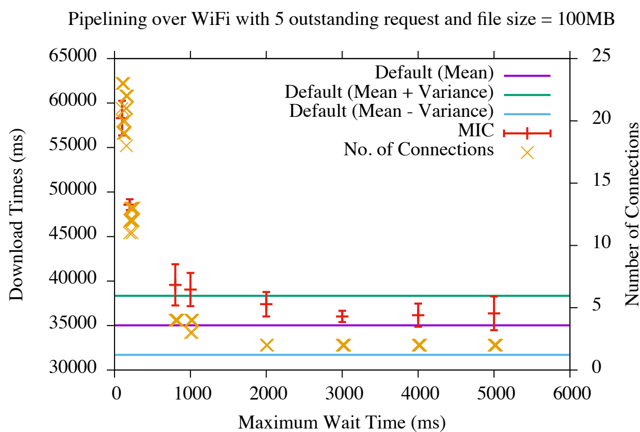

Fig. 9. Comparison of default implementation and our implementation with dynamic chunking, and request pipelining for 5 outstanding requests

in wait-time similar to what we saw earlier (Figure 7). This is due to the number of *connection-close* events we receive from the server. We see that the number of connection-close events decrease with the wait-time. Smaller wait-time imply smaller chunks which in turn mean more requests. We observe that the server a larger number of connection-close events in this case. Our empirical observation is that the connection-close events (and therefore observed overhead) is now proportional to the number of chunks we download.

Figure 9 shows the same results over WiFi with 5 outstanding requests. We observe the results are about the same, with minor variations in the download time in case of 5 outstanding requests. They are slightly higher than the 3 outstanding request case due to the fact that with increasing number of outstanding requests, we would have more number of failed requests that need to be resent for every connection-close we receive from the server.

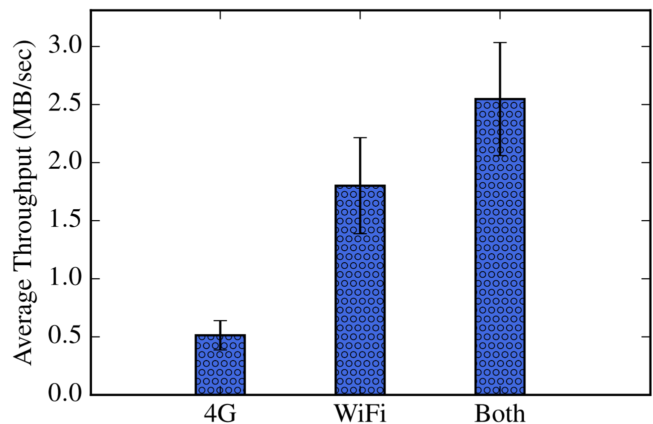

Fig. 10. Average throughput with 4G, WiFi, and both configurations over 20 trials

D. Performance evaluation over multiple interfaces

In this sub-section, we show our results for the combined throughput and seamlessness when using more than one interface in our implementation. Figure 10 shows the results as a bar graph. The results are averaged over 20 trials, where each trial consisted of 3 experiments, one downloading a file using the 4G interface, second performing the same download using the WiFi interface and third performing the same download with both the interfaces enabled concurrently. We set `wait-time` as 1 second for all the experiments. We calculate the average throughput for each run within a set, by dividing the total file size (100 megabytes) with the total time taken. Figure 10 plots the mean and variance for the three configurations (WiFi, 4G and both). The 4G interface observes an average throughput of 500kB/s. The WiFi interface observes a throughput of about 1.75 MB/s. Our implementation that uses both the interfaces concurrently observes an average throughput of 2.5 MB/s which is greater than the average throughputs of the individual interfaces combined. We believe that the increase in throughput is due to variance in the WiFi throughput over the course of our experiments. This result demonstrates that the we have *managed to eliminate the overhead* observed by our implementation and make full use of the bandwidth available on each interface.

A second goal of our design was to provide seamlessness to the user. Figure 11 shows the seamlessness and resumability provided by our implementation. In this figure, we plot the instantaneous throughput for each chunk over 4G and WiFi for a file of size 500 megabytes. We have segmented the plot into different sections based on the availability of network interfaces. There are four possible combinations given two network interfaces i.e. both on, both off and either of the two on while the other is off. We start with both the interfaces enabled (case 1) and then turn off the WiFi interface (case 2) which essentially reflects the scenario of moving out of the office WiFi when you step on the street. In this case, the download continues over the 4G interface. Next we turn off

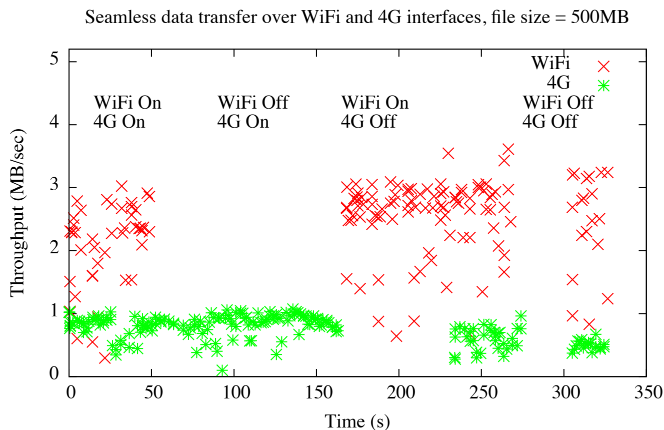

Fig. 11. Proof of seamlessness and resumability provided by our implementation

the 4G interface and turn on the WiFi interface (case 3) which reflects the scenario of entering in a WiFi only zone such as a coffee shop. The download in this case continues over the WiFi interface. We see that the data transfer continues in the absence of availability of one interface, thus supporting the seamless data transfer. There is short duration during which none of the interfaces are available (case 4). The download essentially pauses during this interval and continues once either of the interface(s) is available. This demonstrates the resumability provided by our implementation.

This experiment highlights the power of our system, and achieves the goals we set out to do. Our focus was to *use multiple network interfaces concurrently* in an *efficient, seamless* manner on modern mobile devices. Through various platform changes described in this chapter, and extensive experimentation, we've achieved this goal.

VI. Conclusions

This paper is a guide to achieve multi-interface connectivity on Android at the HTTP level for a single download. We describe all the steps that are necessary to be taken in order to get the best performance over multiple interfaces. Future work includes policy-based networking wherein a user can select from one of the available policies, that would result in enabling one or more interfaces. For example, we could have a policy to reduce the use of mobile interface (3G or 4G) for cost reduction.

So far, we have focused on single download cases. When the number of downloads increases, we have other questions that come into picture such as load balancing, as we do not want to overwhelm one particular interface taking care of most of the requests. In such cases, we would want to have a well devised algorithm that takes into account the number of requests in the queue per interface as well as network parameters such as bandwidth and latency associated with it. Based on these parameters, we could make smart decisions regarding request assignment.

In addition, we would also like to study the interference effect of having multiple interfaces running at the same time. Furthermore, using multiple interfaces simultaneously might have an effect on the battery drain of the phone. We would like to devise intelligent power management policies that would make best use of the available interfaces.

We believe that this work has much potential in making the best use of available network around us. People would not only be able to select the network of their choice, but also the underlying platform (or the OS) would help them make these decisions.

Acknowledgment: This work has been supported in part by an NSF CAREER award, CNS-1350883.

References

[1] Open vSwitch: An Open Virtual Switch. http://openvswitch.org/.

[2] Deng, S., Sivaraman, A., and Balakrishnan, H. All your network are belong to us: A transport framework for mobile network selection. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2014), HotMobile '14, ACM, pp. 19:1–19:6.

[3] Falaki, H., Lymberopoulos, D., Mahajan, R., Kandula, S., and Estrin, D. A First Look at Traffic on Smartphones. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 281–287.

[4] Gember, A., Anand, A., and Akella, A. A Comparative Study of Handheld and Non-handheld Traffic in Campus Wi-Fi Networks. In *Proceedings of the 12th International Conference on Passive and Active Measurement* (Berlin, Heidelberg, 2011), PAM'11, Springer-Verlag, pp. 173–183.

[5] Hare, J., Hartung, L., and Banerjee, S. Transparent Flow Migration through Splicing for Multi-Homed Vehicular Internet Gateways. In *2013 IEEE Vehicular Networking Conference, Boston, MA, USA, December 16-18, 2013* (2013), pp. 150–157.

[6] Higgins, B. D., Reda, A., Alperovich, T., Flinn, J., Giuli, T. J., Noble, B., and Watson, D. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2010), MobiCom '10, ACM, pp. 73–84.

[7] HTTP/1.1. Pipelining.

[8] Kim, J., Khalili, R., Feldmann, A., Chen, Y.-C., and Towsley, D. Multi-source multi-path http (mhttp): A proposal. *CoRR abs/1310.2748* (2013).

[9] Nika, A., Zhu, Y., Ding, N., Jindal, A., Hu, Y. C., Zhou, X., Zhao, B. Y., and Zheng, H. Energy and Performance of Smartphone Radio Bundling in Outdoor Environments. In *Proceedings of the 24th International Conference on World Wide Web* (2015), WWW '15.

[10] Paasch, C., Detal, G., Duchene, F., Raiciu, C., and Bonaventure, O. Exploring Mobile/WiFi Handover with Multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design* (New York, NY, USA, 2012), CellNet '12, ACM, pp. 31–36.

[11] Rahmati, A., Shepard, C., Tossell, C., Zhong, L., Kortum, P., Nicoara, A., and Singh, J. Seamless tcp migration on smartphones without network support. *Mobile Computing, IEEE Transactions on 13*, 3 (March 2014), 678–692.

[12] Schmidt, P. S., Enghardt, T., Khalili, R., and Feldmann, A. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2013), CoNEXT '13, ACM, pp. 295–300.

[13] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a Java bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (1999), CASCON '99.

[14] Yap, K.-K., Huang, T.-Y., Kobayashi, M., Yiakoumis, Y., McKeown, N., Katti, S., and Parulkar, G. Making Use of All the Networks Around Us: A Case Study in Android. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design* (2012), CellNet '12.