Socket Intents: Leveraging Application Awareness for Multi-Access Connectivity

Philipp S. Schmidt[1], Theresa Enghardt[1], Ramin Khalili[1,2], Anja Feldmann[1]
{philipp, theresa, ramin, anja} @inet.tu-berlin.de
TU Berlin[1] / Telekom Innovation Laboratories[2]

ABSTRACT

In today's Internet, almost all end devices have multiple interfaces built in. This enables users to seamlessly switch between different access networks or even use them simultaneously; to better use the resources available to them and to better satisfy their needs. This is referred to as mobile data offloading and has received lots of attention recently in both the research community and in the industry. However, all the proposed data solutions either rely on static configuration policies or are reactive rather than proactive with regards to the application needs.

In this paper, we propose a proactive, application informed approach, *Socket Intents*. Socket Intents augment the socket interface to enable the application to express what it *knows* about its communication patterns and preferences. This information can then be used by our proactive policies to choose the appropriate interface, tune the network parameters, or even combine multiple interfaces. We provide a prototype implementation of our Socket Intents and present a first evaluation of the Intents and its benefits.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; D.4.4 [**Operating Systems**]: Communications Management—*Network communication*

Keywords

Socket Intents; Socket API; Network Programming Interfaces; Application Awareness; Multi-Access Connectivity; Network Properties

1. INTRODUCTION

Ten years ago, most clients had just a single way to connect to the Internet (typically WiFi for laptops, GPRS for smartphones, Ethernet for workstations). Today, almost all devices have multiple interfaces built in. For example, almost all smartphones have built-in 3G/4G as well as WiFi interfaces, see Figure 1, while most laptops have Ethernet interfaces in addition. Moreover, they can seamlessly switch between using one or the other interface or even use multiple of them at the same time. This is often referred to as *Multiple Access Connectivity*.

CoNEXT'13, December 9–12, 2013, Santa Barbara, California, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535405.

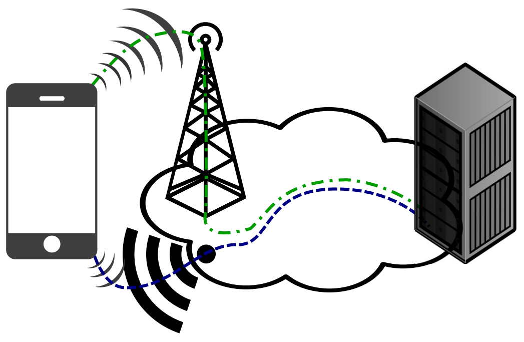


Figure 1: Multi-Access Host with built-in 3G/4G/WiFi.

Multiple access connectivity enables us to ask the question of mobile data offloading which has gotten a lot of attention recently in the research community, e.g., [8, 2], as well as in the industry, e.g. [4]. The motivation is that the performance that each individual network technology provides differs, e.g., in terms of bandwidth, delay, availability, congestion, cost per byte, and energy cost. For example, while WiFi if uncongested provides higher bandwidth than 3G and therefore may be preferable, the 3G network may be preferable if the WiFi network is congested. Moreover, 3G and 4G work even if the user is no longer in reach of her WiFi access points. Of course 4G currently offers higher peak bandwidth than even most DSL lines but it is a shared medium and thus susceptible to congestion. Thus, there is a lot of optimization potential on the devices with respect to choosing or combining the appropriate interfaces.

Earlier work in this area focused on either the offloading case, choosing an appropriate interface, or using multiple interfaces in parallel. For example for the offloading case we refer, e.g., to Lee et al. [8] who demonstrate via a quantitative study the performance benefit of offloading 3G mobile data to WiFi networks, and Balasubramanian et al. [2] who propose Wiffler to augment mobile 3G capacity with WiFi. For choosing an interface we, e.g., point to the work in the IPv6 multihoming context [9, 14], DNS related approaches, e.g., RFC6419 [15], or locator/identifier approaches, e.g., [10]. The most recent work for using multiple interfaces in parallel is Multipath TCP [5, 11].

However, all of the above solutions either rely on static configuration policies or are reactive rather than proactive with regards to the application needs. While reactive solutions like Multipath TCP can efficiently distribute traffic over multiple links, they can only react to effects like congestion, RTT and others observed over time throughout the transfer and therefore only long living flows can benefit from them.

In this paper, we propose a proactive, application informed approach: *Socket Intents*. For example, if the user of the smart phone of Figure 1 wants to download the newest football scores or stock market quotes, most objects will be small and delay is critical. However, if she accesses the newest e-book, bandwidth is more critical. One of the key problems though is that the transport and network layer typically have no idea of the size of the downloaded data nor the relevance to the user. However, this information is often available to the application. Therefore, Socket Intents augment the socket interface to enable the application to express its communication preferences. Here, preferences can refer to high bandwidth, low delay, connection resilience, etc. This information can then be used by our dynamic proactive policies to choose the appropriate interface, tune the network parameters, or even combine multiple interfaces.

The principle observation that the application has critical information is not novel, but has been made before in the context of Quality of Service. The main difference between QoS and our Socket Intents approach is, as smartly summarized by the CoNEXT TPC, that with the Socket Intents approach, "(...) the application tells what it *knows* as opposed to what it *wants*, as in prior work on QoS". Therefore, the application expresses what it knows about the communication, what the traffic might look like and what the application can tolerate. Based on that knowledge, our policies can take advantage of interface QoS, if it is available, but does not rely on it as the different interface technologies by default offer different QoS characteristics.

While Socket Intents are inspired by Intentional Networking [6], our system focuses on enabling elaborate policies instead of having a fixed one. We extend the Socket API to the Internet protocol suite instead of building an overlay layer on multiple protocol suites. Thus, the use of our system only requires minimal changes to the socket interface as well as to the applications and is incrementally deployable.

In summary, our system consist of (i) an augmented socket library to communicate the needs of the application, (ii) a set of policies to select or combine appropriate network interfaces, and (iii) mechanisms to combine or select interfaces.

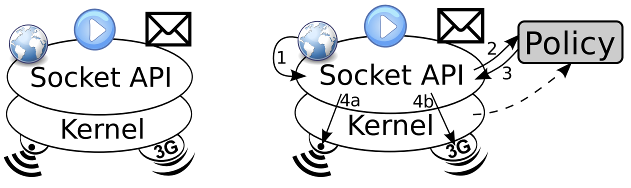
Our key contribution is the introduction of the concept of Socket Intents together with a prototype implementation and a first evaluation. Socket Intents provide:

- A generic scheme which enables applications to express their knowledge and needs for its communication on a per-connection basis.
- An interface for fulfilling applications' needs on a best-effort basis without requiring QoS.

The remainder of this paper is structured as follows: First we provide an overview over Socket Intents from a system point of view and present a set of Socket Intents for applications to express their needs. Then we describe our prototype implementation in more detail and present a first evaluation that illustrates a possible use of the Intents and its benefits. Finally, we conclude our paper with an outlook and possible next steps.

2. SOCKET INTENTS

The goal of Socket Intents is to enable the applications to express their communication preferences in order to take advantage of the various network interfaces. We assume that applications specify their preferences in a selfish, but not malicious way and that it is up to the policy to find a compromise between demands. Therefore, selfish behavior does not hurt other applications as it would with



(a) Traditional. (b) With Socket Intents.

Figure 2: Within host: Applications/network interfaces.

prioritization. In the following we discuss where the interface to specify intents should be placed and what the challenges are.

2.1 Concept

Let's review how communication works within today's OSes, see Figure 2(a). To start, the application request a socket, either TCP or UDP, via the Socket API, today's almost universal interface. The OS then maps the socket to a specific network interface using a fixed internal policy and then uses it for the communication. The fixed internal policy currently gets minimal information from the application.

To benefit from the application's knowledge about the upcoming communication, the programmer has to pass the information available within the application along with the request for communication. If no information is provided, the OS can only rely and react on information it can gather from the communication itself.

There are two principle options to add more information: either modify the kernel or extend the Socket API. We choose the latter as it is simpler and easier to port. More specifically, we decided to stick to user-space modification. Accordingly, we augment the Socket library of the Socket API with additional *socket options* and add an explicit user space policy manager, Figure 2(b). Upon (1) recognizing a Socket Intent option, the Socket library (2) calls the user space policy manager which can then (3) choose either a single interface or multiple interfaces. This is then used to (4) override the kernel interface policy.

Having stated this system architecture, the key design questions are:

- Which Intents?
- What policy?
- Do the policies need additional information?

With regards to realizing Socket Intents the system questions are:

- How to add Socket Intents to the Socket library?
- How to enable the interactions with the policy?
- How to bias the interface selection, e.g., by choosing source and destination IP addresses?

2.2 Applications Intents

As mentioned before, the goal of Socket Intents is to enable the applications to express their communication preferences. Here communication preference refers to desired characteristics, e.g., low delay or high throughput and is optional information. Socket Intents are purely advisory. They are not meant to specify hard requirements or imply QoS guarantees. Rather they are accounted for on a best-effort basis. Still, Socket Intents are inspired by DiffServ as well as IntServ in the sense that they specify traffic classes on a per connection basis.

Intent	Type	Value
Category	Enum	Query, bulk transfer, control traffic, stream
File size	Int	Number of bytes transferred by the application
Duration	Int	Time between first and last packet in seconds
Bitrate	Int	Size divided by duration in bytes per second
Burstiness	Enum	Random bursts, regular bursts, no bursts or bulk (congestion window limited)
Timeliness	Enum	Stream (low delay, low jitter), interactive (low delay), transfer (completes eventually) or background traffic (only loose time constraint)
Resilience	Enum	Sensitive to connection loss, undesirable (loss can be handled) or resilient (loss is tolerable)

Table 1: List of proposed Socket Intents.

Let's start with some examples: (i) If the antivirus software needs an update this usually implies downloading a large file. This is a bulk transfer for which the application may already know the file size. Timing is typically noncritical and the data can be downloaded as background traffic. It would not hurt if the TCP connection was closed during the transfer as the download can be continued. For this connection the application can set the general category bulk transfer. Additional information provided can be file size, timeliness, and resilience. (ii) If you want to watch a video it usually means using a streaming application. This is a stream transfer for which the application may know the bitrate and the duration. It does rather not want the TCP connection to be disconnected because that might have an effect that is visible to the user. It can put this connection into the general category stream transfer with additional information about duration, bitrate and resilience.

Based on this philosophy, we propose a set of Socket Intent options, see Table 1. Socket Intents are optional in the sense that they are not required but any number of them can be specified. They are structured hierarchically, starting with the "category" option with possible values of query, bulk transfer, control traffic, stream which are realized as enum. Then we have more specific options which include file size, duration of the flow, expected bitrate, whether the traffic is bursty, whether the timeliness of the flow completion matters and how resilient the application is against connection loss. Each of these can either be enums or integers. Note that these are extensible and Table 1 is only a first proposal.

2.3 Policies

Our system design places the hardest problem, the actual decision which interface(s) to use for which communication, into the policy component. It is important to note that the specific policy that is most beneficial will likely depend on the configuration of the host, the current location, the current network availability, etc. Because this is a very hard problem, we decided to not focus on a specific policy but rather provide a generic framework in which one can use and evaluate different policies. Accordingly, the system components of our design are summarized in Figure 3.

We do not address the problem of how to find a reasonably good policy within this paper, however for our prototype implementation we need some policies to start with. Among the most obvious policy is the following one: use high bandwidth interfaces for application requests of the category bulk transfer and low latency interfaces for application requests of the category query. With our Socket Intents, we enable the policy to decide what to optimize for, which would be undecidable otherwise. Yet, even this simple policy points out certain limits to realizing policies without additional information. How can the policy infer that an interface is high bandwidth or low delay? This is information that can only be derived via configuration or measurements. Accordingly, we decided to add both a configuration interface as well as a statistics interface to our prototype.

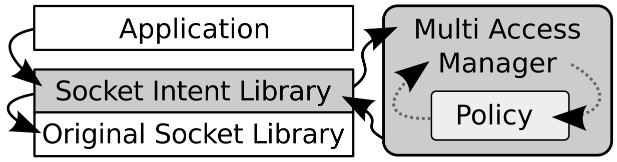


Figure 3: Components of the framework.

Finally, policies are not limited to the use of a single interface if the transport protocol supports the simultaneous use of multiple interfaces. The polices in our current prototype use socket options to control use of MPTCP [11] or multiple paths in SCTP [13]. Future versions will incorporate interactions between our policies and the path selection of MPTCP and SCTP.

3. IMPLEMENTATION

Our Socket Intents implementation consists of three components: the Multi Access Manager (MAM), the policies, and the Socket Intent library, see Figure 3, each implemented in C and compatible with Linux as well as MacOS.

We realize the selection of the network interface by choosing the source address of the new connection. While this approach seems to be a hack in the first place, it just overrides a decision which is usually made by the kernel based on a very simple policy and the routing table with a more informed decision.

Furthermore we can optimize by choosing among the possible alternatives for the destination address. This is possible because a large fraction of the content, especially almost all content delivered through Content Distribution Networks (CDNs), is served by multiple servers that can be resolved from a single host name. In addition, for a single host name, there can be different translations if requested over different interfaces or from different name servers as a result of optimizations by a CDN.

The functions as well as the control transfers realizing the mechanisms mentioned above are shown in Figure 4. The latter figure highlights that it is not sufficient to just modify the socket and the connect calls. Rather we also needed to modify the interface to the resolver which handles the host name to IP address translation.

3.1 Multiple Access Manager

The Multi Access Manager (MAM) is a daemon for hosting the policy modules which can be exchanged. As such it provides the policy framework with initialization and request processing.

Upon startup, the MAM scans for available network interfaces. Then, it reads the configuration file which includes the list of interfaces to include/exclude for source or destination address selection. Moreover, it is possible to specify interface and policy specific information via key/value pairs there. This information is stored within the MAM and later made available to the policy. Then it instantiates the policy from a dynamically loaded library and initializes it. This finishes the initialization phase of the MAM and it then moves into operation. This means that the MAM can now process requests by the application via the Socket Intent API. These requests are communicated via Unix domain sockets and realized asynchronously using *libevent*.

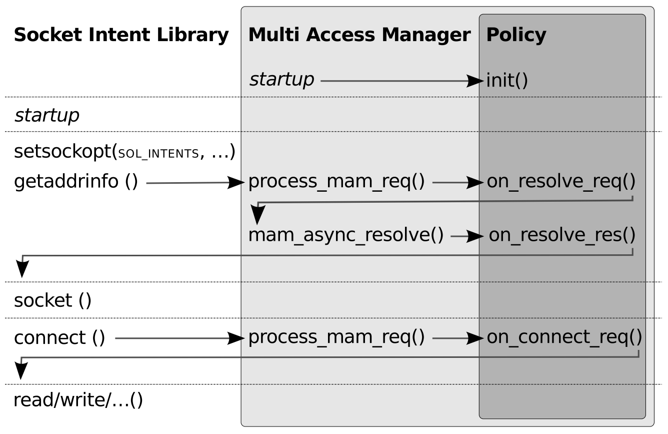


Figure 4: Interactions of the Socket Intent components.

3.2 Policy

The policy module is invoked by the MAM for each `getaddrinfo` or `connect` call of the Socket Intent API, see Figure 4. The MAM also provides the request context which includes the socket options including the Socket Intents as well as the state of the connection. The latter includes information about the network interfaces as well as if the application already executed a "bind" call. This is necessary as layering within TCP/IP is not strict and thus the application may be involved in address resolution and selection.

Within the `getaddrinfo` call the policy typically uses the name resolution framework provided by the MAM and discussed later, to compile an ordered list of possible destination addresses which is then returned to via the MAM to the application. For the `connect` call the policy selects the desired source and destination addresses from the request context information provided by MAM. Typically though the destination IP address is a parameter for the connect call and picked by the application based on the results of the `getaddrinfo` call. Moreover, the policy suggests the appropriate socket options to be set by the Socket API.

3.3 Augmented Socket Library

The modifications to the Socket API are threefold: we introduce the Socket Intent options, we enable the MAM to select the source IP and the destination IP address.

Socket Intent Options:
We decided to use socket options for similar reasons as in RFC5014 [9] or RFC6724 [14]. For this we introduce a new socket option level, called `SOL_INTENTS`, and within that we use names similar to those of Table 1. This involves fewer changes to the Socket API than needed in alternative approaches like [12, 1, 16] or passing labels with each message as proposed by Intentional Networking [6]. The intents are stored in a per-socket context and passed to the MAM.

Source Address Selection:
A policy selects a network interface by choosing the source address. Here we need to consider two cases: either the application calls `bind` itself or it does not. In the first case the application picks the source IP and includes it in the bind call. In this case the policy has the option of overwriting the source IP but typically should not. If the application does not call `bind` then the policy chooses the source IP address and our modified `connect` binds the socket to the selected source IP address.

Destination Address Selection:
For the policy to select the destination address, it is not sufficient to just modify the socket library calls as it involves name resolution. Therefore, we include a resolver library based on *libevent* with MAM. The resolver performs name resolution asynchronously from within the policy over all network interfaces specified in the MAM configuration file. Among the per interface parameters are which DNS servers to use with which parameters. Once the policy deems that it has sufficient information the resolving step can be aborted.

Realizing the address selection highlights another OS limitation. Typically sockets and resolver calls are not directly associated with each other. But this association is necessary for destination address selection within Socket Intents. We realize this by adding the socket context as an additional parameter to all socket and resolver calls.

4. EVALUATION

To understand which benefits Socket Intents provide to the end-user we, in this section, use illustrative examples to highlight the potential of even very simplistic policies.

Client Setup:
For this purpose we modified one of the "simplest" HTTP client, namely *wget*, to enable it to set Socket Intents. More specifically, we added support for two different Socket Intents: "category" and "filesize".

The goal of the "category" intent is to broadly classify upcoming transfers. It can be explicitly set by an end-user as command line parameter given to *wget*. We use it in Scenario 1 by setting it to be either "bulk" or "query" based on prior knowledge about the evaluation setup.

The goal of the intent "filesize" is to enable the Socket Intent API to distinguish between large and small downloads automatically. The latter are more sensitive to delay while the former are more sensitive to bandwidth availability. However, for Web we do not necessarily know the object size up front. To determine the value of the "filesize" intent, we further modified *wget* to perform its downloads in two phases taking advantage of the HTTP range query capabilities. More specifically, our modified *wget* first issues a range request for the first m bytes of each object[1] in order to acquire the size of the object from the HTTP header. It then opens a new connection for which it specifies the "filesize" n to download the remaining $n - m$ Bytes. We use the two-phase download to determine the "filesize" intent in Scenario 2.

One unfortunate limitation of *wget* is that it neither supports HTTP pipelining nor multiple TCP connections. Thus, in order to take advantage of two independent network interfaces one may have to run multiple *wget* instances.

Evaluation Setup:
We opt for clients with two network interfaces of opposite properties: one with relatively low delay but also only limited bandwidth and one with relatively large delay but higher bandwidth. More specifically: `i1` resembles a relatively low bandwidth DSL line with fast-path, i.e., 10ms RTT, 6Mbits downstream bandwidth and 768Kbits upstream bandwidth, as this is the type of DSL line in most parts of Germany [3]; `i2` resembles a reasonable LTE network access, i.e., 70ms RTT, 12Mbits downstream and 6Mbits upstream [7]. In all scenarios, we assume that these characteristics remain stable and are known to the policy.

[1] m for the initial request should be chosen in order to address the trade-off when a TCP download is dominated by the round trip time and when it is dominated by the network bandwidth. We suggest to use values for m that fit within the initial TCP congestion window.

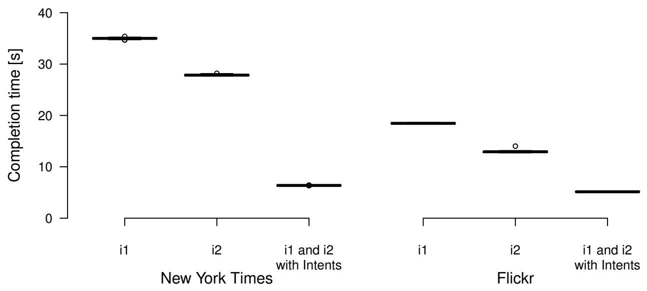


Figure 5: Scenario 1 — Boxplot of website loading times.

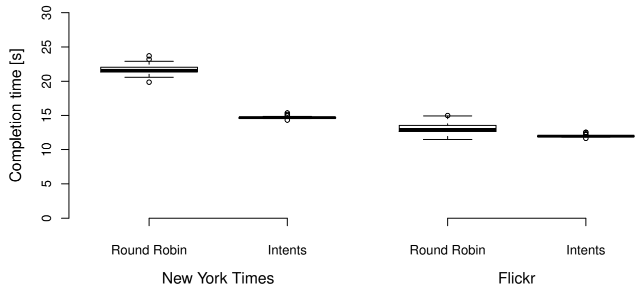


Figure 6: Scenario 2 — Boxplot of website loading times.

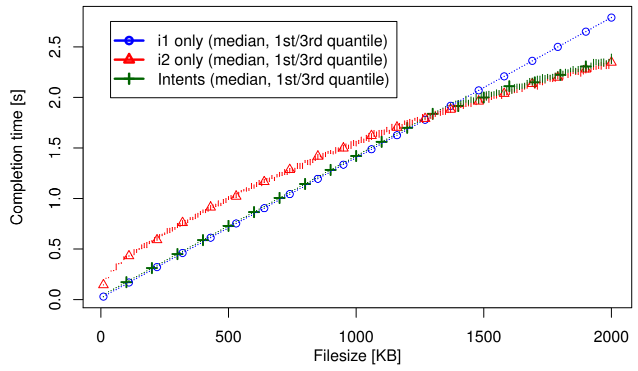


Figure 7: Scenario 2 — Completion times per object size.

We realize this scenario in an emulated environment where we have explicit control over all components. As server and client we use two Linux machines (Intel Xeon L5420, 2x4 Cores, 16GB RAM) interconnected via a 1Gbit/s Ethernet and routed via a machine of the same type on which we run a traffic shaper. More specifically, for shaping we use the *TC hierarchical token bucket (HTB)* scheduler and for delay the *TC Network emulator* scheduler. The client is equipped with two independent Ethernet interfaces which are configured as `i1` and `i2`.

With regards to the workload for both scenarios, we use snapshots of Web pages: more specifically the landing page of a popular newspaper, New York Times (`www.newyorktimes.com`), and a sub-page of a popular photo sharing site, Flickr (`www.flickr.com/explore`) from Jun 10th 2013. Both pages have a size of 2.8 MBytes. The first one has roughly 130 embedded objects which are relatively small with a range from 48 bytes to 263 KBytes with a median of 6 KBytes and a mean of 21 KBytes. The second one consist of only 30 embedded objects with size between 43 bytes to 572 KBytes a median of 65 KBytes and a mean of 94 KBytes. The Web pages were retrieved using Google Chrome's save whole webpage function and copied to the server. In addition, in Scenario 1 we use a bulk transfer as background traffic, realized by downloading files of size greater than 48 MBytes.

Scenario 1: Bulk Transfer vs. Query
In our first scenario we revisit a performance problem often encountered at home: browsing a Web site while downloading a large file. The first one is response time critical while the latter is hawking the bandwidth. We simulate this scenario by two parallel *wget* instances: While instance (i) with intent "bulk transfer" fills the link by downloading a large file, instance (ii) with intent "query" tries to fetch one of our two websites.

The policy that we use here is that "bulk transfers" are sent over the higher bandwidth interface `i2` and "queries" over the low delay interface `i1`. We compare this policy to the case where the client is restricted to use either of the two interfaces. The measured completion times for the Web downloads are shown in the boxplot of Figure 5. Recall that a boxplot displays the median, the spread and the skewness of all experiments in one plot. The experiment is repeated 30 times.

Overall, the download time for the Flickr page is smaller than for the New York Times one although the total size is about equal. However, the median object size of Flickr is significantly larger. Moreover, using the high bandwidth interface `i2` rather than the low bandwidth interface is beneficial. Enabling Socket Intent policies improves the page download performance by more than a factor of 60% without penalizing the bulk download. The reason is that the page download can now be scheduled on the network interface `i1` and does no longer compete with the bulk download which has a stronger effect than just having more bandwidth available. We note that this example is in some sense the best case as we can now fully use the second network interface. In future work we plan to evaluate how the policy can take advantage of multipath transport protocols such as MPTCP. We predict that using MPTCP will degrade the performance of the Web page download while increasing the performance of the bulk download.

Scenario 2: Offload by Filesize
In our second scenario we utilize the "filesize" intent to enable the policy to choose the "best" interfaces given that they may have different characteristics. In our evaluation setup, small transfers should benefit from the low RTT of `i1` while large transfers benefit from the larger bandwidth of `i2`. In order to do this, we first have to find the threshold at which the larger bandwidth of `i2` yields to a smaller completion time than the low RTT of `i1`. We obtained this threshold experimentally using an artificial workload consisting of files from 10 KBytes to 2 MBytes in steps of 10 KBytes which we download using a single *wget* instance measuring the completion time. We know that this approach involves a lot of overhead and will not work well in practice as the threshold is highly depending on the RTT and available bandwidth, which will be varying in most environments. However we will explore a more realistic mechanism to determine this threshold in future work.

Based on that, we then use a policy that offloads transfers to the high bandwidth interface `i2` if the "filesize" is above the experimentally obtained threshold of 1400 KBytes and uses the low delay interface `i1` otherwise. The results of using each interface individually compared to using Socket Intents are shown in Figure 7. Figure 7 includes the median of the 15 experiment runs as well as the 1st and 3rd quantiles. Note, that the experimental variability is small as the quantiles hardly differ from the median. We also see that using the threshold of 1400 KBytes ensures that the policy is always picking the "best" interface and that the overhead of *wget* retrieving the actual filesize with the initial partial request of 15 KBytes of the two-step download is minimal. Note that the threshold of 1400 KBytes may appear large but this very much depends on the individual interface parameter, the current client location within the network, and the congestion within the network.

While the previous examples shows the principle advantages of intents, the comparisons can be considered unfair. Thus, we now compare the previously presented "filesize" policy using an experimentally obtained threshold with an application unaware policy that uses the two interfaces in a "round-robin" fashion, which alternates between the two interfaces when fetching each object.

To go to a slightly more diverse setting where the website download can take advantage of the different link characteristics, we reduce the bandwidth of one of the interfaces to 2 Mbit/s, `i1'`. This corresponds to the slower DSL lines common outside of urban environments [3]. The resulting threshold is 100 KBytes.

As *wget* is unable to issue parallel requests we download both of the Web pages at the same time simulating two users that are using an access point with multiple network interfaces.

The New York Times site benefits from the application unaware round-robin policy with an improvement in performance by 20% and 18% vs. the interface `i1'` only and the interface `i2` only case while the other does not perform worse (results not shown). The Web download times for the round-robin and "filesize" policies are shown in the boxplot of Figure 6. We point out that downloading the New York Times site benefits significantly from using the "filesize" intent. Its download time is improved by 35% or, put differently, using round-robin is 1.5 times slower. For the Flickr site the completion time advantages are smaller. This is mainly due to the large number of small objects that benefit more from using the low latency interface `i1'`. In conclusion, this experiment highlights that even a simple application unaware policy can improve performance for multi-access devices and application aware policies can yield even better performance than application unaware ones.

5. SUMMARY

The trend with Internet capable devices has gone from adding a second network interface to adding the third or even fourth. However, the benefit to the end-users has been limited as offloading, while technically feasible, has not yet become standard practice. One of the limiting factors is good information for choosing which network interface to use for which communication and therefore what to optimize for. We propose to resolve the latter with *Socket Intents* which enable applications to express what they know about their communication instead of having to care about that they can request from the network.

In this paper we propose a realization of Socket Intents via a modified Socket API, an environment specific *policy*, and a *Multiple Access Manager (MAM)*. Our initial results show the potential benefit of using application intent aware policies for choosing which combination of interfaces to use.

In future work we plan to expand our exploration of possible policies that take properties and conditions of the interfaces into account and adapt to them automatically. For this we are planning to expand our framework with advanced network statistics, e.g., RTT, RSSI, carrier data rate, packet loss or even hints from the ISPs which can be used like a *network weather report*. In addition, we plan to explore the inclusion of newer protocols with path management capabilities, such as *MPTCP*, *SCTP*, and *IFOM*. So far our client side is limited to a very simple client, namely *wget*. We are planning to add Socket Intent support to a browser as well as its multimedia plugins. In addition, there is the question of how to handle multiple requests to the same destination. This may require added support for pipelining within the MAM.

Moving forward we claim that including the proposed Socket Intent API together with the MAM within popular OSes may provide a road towards universally taking advantage of one's multiple network interfaces and enabling more elaborate optimizations.

6. ACKNOWLEDGEMENTS

This work was supported in part by the EU project CHANGE (FP7-ICT-257422) `http://www.change-project.eu/` and the EIT KIC project MONC.

7. REFERENCES

[1] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and Richard. A quality-of-service enhanced socket api in gnu/linux. In *Real-Time Linux Workshop*, 2002.

[2] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *ACM MobiSys*, 2010.

[3] Bundesministerium für Wirtschaft und Technologie. Breitbandatlas (governmental report on broadband availability and usage in germany), 2013. `http://www.zukunft-breitband.de/DE/breitbandatlas`.

[4] Cisco Systems, Inc. Architecture for mobile data offload over wi-fi access networks (whitepaper), 2012.

[5] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), Mar 2011.

[6] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *ACM MobiCom*, 2010.

[7] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *ACM MobiSys*, 2012.

[8] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile data offloading: how much can wifi deliver? In *ACM CONEXT*, 2010.

[9] E. Nordmark, S. Chakrabarti, and J. Laganier. IPv6 Socket API for Source Address Selection. RFC 5014 (Informational), Sep 2007.

[10] B. Quoitin, L. Iannone, C. de Launois, and O. Bonaventure. Evaluating the benefits of the locator/identifier separation. In *ACM/IEEE Workshop on Mobility in the evolving internet architecture*, 2007.

[11] M. Scharf and A. Ford. Multipath TCP (MPTCP) Application Interface Considerations. RFC 6897 (Informational), Mar 2013.

[12] A. A. Siddiqui and P. Müller. A requirement-based socket api for a transition to future internet architectures. In *IMIS*, 2012.

[13] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich. Sockets API Extensions for the Stream Control Transmission Protocol (SCTP). RFC 6458 (Informational), Dec 2011.

[14] D. Thaler, R. Draves, A. Matsumoto, and T. Chown. Default Address Selection for Internet Protocol Version 6 (IPv6). RFC 6724 (Proposed Standard), Sep 2012.

[15] M. Wasserman and P. Seite. Current Practices for Multiple-Interface Hosts. RFC 6419 (Informational), Nov 2011.

[16] M. Welzl, S. Jorer, and S. Gjessing. Towards a protocol-independent internet transport api. In *ICC*, 2011.