Scheduling Packets over Multiple Interfaces while Respecting User Preferences

Kok-Kiong Yap* Te-Yuan Huang Yiannis Yiakoumis Sandeep Chinchali
Nick McKeown Sachin Katti
Stanford University
{yapkke,huangty,yiannisy,csandeep, nickm,skatti}@stanford.edu

ABSTRACT

Now that our smartphones have multiple interfaces (WiFi, 3G, 4G, etc.), we have preferences for which interfaces an application may use. We may prefer to stream video over WiFi because it is fast, but VoIP over 3G because it gives continued connectivity. We also have *relative* preferences, such as giving Netflix twice as much capacity as Dropbox. This means our mobile devices need to schedule packets in keeping with our preferences while making use of all the capacity available. This is the natural domain of fair queuing, and this paper is about the design of a packet scheduler to meet these requirements. We show that traditional fair queueing schedulers cannot take into account a user's preferences for some interfaces over others. We present a novel packet scheduler called miDRR that meets our needs by generalizing DRR for multiple interfaces. We demonstrate a prototype running in Linux and show that it works correctly and can easily run at the speeds we need.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer-Communication Networks—*Network Architecture and Design*

General Terms

Design, Performance, Theory

Keywords

Fair Queueing; Packet Scheduling; Mobile Devices

*Research funded by NSF Expedition Grant 0832820, POMI 2020. K.K. Yap is now at Google. Te-Yuan Huang is funded by a Google Fellowship.

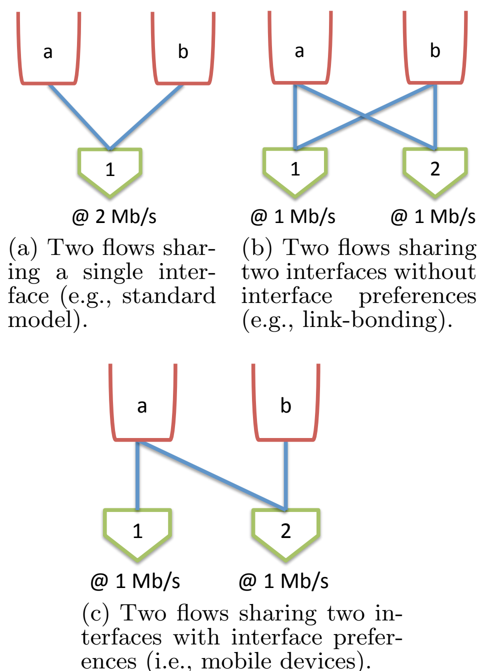
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CoNEXT'13, December 9–12, 2013, Santa Barbara, California, USA.
Copyright 2013 ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535387.

1. INTRODUCTION

Mobile devices have multiple network interfaces, allowing them to connect to a variety of networks. For example, our phones have 3G, 4G and WiFi interfaces, and it is increasingly common for users to have two or more interfaces active at the same time. We are also learning that users have *preferences* on how to use different networks with very different characteristics. For example, since 3G/4G connectivity is often capped, we might prefer to download music and stream videos over free WiFi connections. If we are making a VoIP call through Skype, we might prefer to use WiFi since the latency of 3G networks is higher. If we are on the move and streaming music from Pandora, we may prefer to use cellular to ensure we are persistently connected. And if we are accessing a website from work, we may prefer to use cellular so our employer does not know. In the future, we will have preferences that are not currently supported, for example we may want to use all the interfaces at the same time to give all the available bandwidth to a single application. One can imagine a huge variety of preferences, all driven by a tradeoff of bandwidth, latency, cost and persistent connectivity that balances the preferences of the user with the needs of the applications. This paper is about how to meet a user's preferences.

Preferences are not new; mobile operating systems already offer coarse preference through a variety of ad-hoc mechanisms. For example, Android allows us to specify that updating applications should only happen over WiFi, or that Netflix should only use WiFi, and so on. Similarly Windows Phone has a feature called DataSense to apply application and user preferences to choose between WiFi and cellular interfaces. Users also find creative ways to implement our preferences; we might switch off cellular data when we want to force applications to use WiFi or when we are close to our monthly data cap. We believe we should not need to jump through hoops to fulfill our preferences. What we need is a systematic way to ensure our applications follow our preferences when our device has multiple interfaces.

Our goals in this paper are: (1) To show that scheduling packets to meet these preferences is non-obvious—the existing methods, such as fair queueing, do not work for heterogeneous interfaces, and (2) To present an algorithm to share multiple interfaces while respecting user preferences about how they should be used and by which applications. Our approach is based on two different types of preference. First, *interface preferences* indicate whether or not an application is willing to use a particular interface. This is a binary pref-



(a) Two flows sharing a single interface (e.g., standard model).

(b) Two flows sharing two interfaces without interface preferences (e.g., link-bonding).

(c) Two flows sharing two interfaces with interface preferences (i.e., mobile devices).

Figure 1: Examples of packet scheduling. An edge between a flow and an interface indicates the flow's willingness to use the interface.

erence; for example, "YouTube can only use WiFi." Second, *rate preferences* let a user to choose how much of an interface an application gets, subject to the interface preferences (e.g. allocate at least half the bandwidth of the WiFi interface to Netflix). Our goal is to meet both types of preference while maximizing the utilization of all the available networks.

To our surprise, we found no prior work addresses this problem. One might guess that the large amount of work on fair queueing for multiple interfaces might apply. However, prior work [1, 3, 5] does not include the notion of *interface preferences*, all applications are allowed to use *all* interfaces all the time in these frameworks. The prior work focuses only on *rate preferences*, using techniques such as weighted fair queueing (WFQ) [3] to provide weighted fairness. Such work can be found in several contexts, ranging from multihoming to wireless to wireless mesh networks [1, 21].

Interface preferences significantly complicate the problem and render prior work inapplicable. To see why, consider the canonical example in Figure 1. Two applications share the available interfaces, and there are no rate preferences (i.e. each flow has equal weight). As prior work suggests, assume we apply WFQ independently on each interface. If there is only a single interface (Figure 1(a)), WFQ will give an equal fair allocation of 1 Mb/s to each flow. Suppose now we have two 1 Mb/s interfaces and the same total capacity of 2 Mb/s. If the flows have no interface preferences and are willing to use both interfaces (Figure 1(b)), the fair allocation remains 1 Mb/s for each flow which can be achieved by implementing WFQ on each interface. But if we introduce the interface preference that flow a can use both interfaces and flow b can only use interface 2 (Figure 1(c)), implementing WFQ on each interface fails to provide a fair allocation: flow a will get 1.5 Mb/s while flow b only 0.5 Mb/s. This arises because interface 1 gives flow a all its capacity since it is the only flow willing to use the interface, and interface 2 divides its capacity equally between the two flows.

In other words, *WFQ cannot provide rate preferences in the presence of interface preferences.* In what follows we will introduce an algorithm that meets the rate preference (in this case, an unweighted fair share) while respecting interface preferences. In our toy example this means giving each flow 1 Mb/s; flow a gets all the capacity of interface 1, and flow b gets all the capacity of interface 2.

Note that this notion of fairness is a deliberate choice. An alternative choice would be to penalize flow b because it is unwilling, or is not allowed, to use one of the interfaces. Instead, we strive wherever possible to give each flow its weighted fair share of capacity (defined by the *rate preference*), while guaranteeing that it will never violate the *interface preference*. We also make sure it will never unnecessarily waste network capacity (i.e. remain work-conserving on all interfaces).

There are cases where the interface preference (which we consider sacrosanct) stands in the way of meeting the rate preference. Going back to our example, if the user declared a rate preference that flow b should have twice the rate of flow a, we have a problem. If there was no interface preference, flow a would receive 0.67 Mb/s and flow b would receive 1.33 Mb/s. But because flow b can only use interface 2, we can give it at most 1 Mb/s. Should we give flow a only 0.5 Mb/s to honor the rate preference? Our design decision is no. We never want to waste capacity, and so we give flow a all the remaining capacity. We believe this is the right prioritization of goals: while a user's relative flow preferences are typically suggestive, we consider it essential to only use the specified interfaces, and to use capacity efficiently.

A natural place to start looking for a packet scheduler is with WFQ-like algorithms that calculate a packet's finishing time when it arrives, and then schedule departures according to earliest finishing time. We could calculate the finishing time taking into consideration the interface and rate preferences. Surprisingly we find this class of algorithms does not meet our needs; in fact, we prove in Section 2.1 such algorithms could only do so with knowledge of future arrivals.

In this paper, we introduce a novel, practical and efficient scheduling algorithm: multiple interface deficit round robin (miDRR) that schedules packets to meet the rate preference wherever possible, while respecting interface preferences and never wasting capacity. We prove that this is achieved by a classical max-min fair allocation, weighted to give relative rate preference between flows. We further prove that miDRR finds the correct max-min allocation, and hence meets our needs.

Our algorithm is practical to implement, yet we can formally prove its correctness. At first blush, designing an algorithm appears too daunting because each interface needs to know how fast every flow is being scheduled by every other interface. This would lead to a communication explosion in the packet scheduler. The key intuition behind our algorithm—that makes it practical without sacrificing correctness—is that we do not need to know the absolute rate at which each flow is being served, just the relative rates. It turns out that it is sufficient for each interface to maintain a single state bit per flow, telling the interface whether or not a flow was served by another interface since it last served the flow. The one bit flag lets us turn

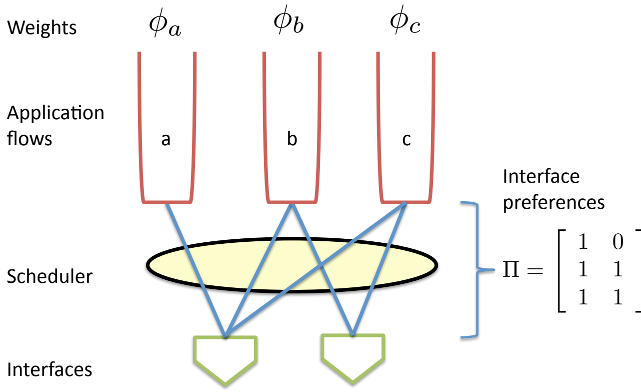


Figure 2: Conceptual model for packet scheduling for multiple interfaces with interface preferences. Matrix Π encodes the flows willing to use each interface, and weight ϕ_i indicates flow i's rate preference.

the single interface DRR packet scheduler into a practical multi-interface scheduler.

The rest of the paper is organized as follows. In Section 2, we define the scheduling problem, the properties we want, and the key challenge we face. We then describe our proposed solution miDRR in Section 3. In Section 4, we show how miDRR satisfies the properties we want and prove its correctness. In Section 5, we describe two implementations of miDRR and present evaluations of the algorithm in Section 6. After a review of related work in Section 7, we conclude the paper in Section 8.

2. DESIRED PROPERTIES OF A PACKET SCHEDULER

Our scheduling problem is captured by the abstract model in Figure 2, with three flows served by two output interfaces. In this model, each flow i has weight ϕ_i to indicate its relative priority (its rate preference). For example, if $\phi_1 = 2\phi_2$, application 1 should receive double the bandwidth of application 2 when both are backlogged. Each flow also has interface preferences to indicate the subset of interfaces it is willing to use. If flow a is willing to use interface 1, we denote $\pi_{a1} = 1$. The flows' interface preferences are captured by connectivity matrix $\Pi = [\pi_{ij}]$. The matrix represents a bipartite graph (as shown in Figure 2) where an edge exists between flow a and interface 1 if and only if flow a is willing to use interface 1, i.e., $\pi_{a1} = 1$. It is critical to note that the bipartite graph is often incomplete—meaning Π is not all-ones, i.e., not all flows are willing to use all the interfaces. As such, we cannot aggregate interfaces to reduce to the classical single interface case. The combination of rate preferences (weights ϕ) and interface preferences (matrix Π) lets us describe a wide variety of interface usage policies.

Our goal is to design an efficient packet scheduler that accounts for the interface preferences captured in this model and meets the rate preferences. A packet scheduler answers the question of *when an interface is available, which packet should be sent?* An ideal packet scheduler answers this question in a way that fulfills several desirable properties (listed below in roughly descending order of importance):

1. **Meet interface preferences.** We want a packet scheduler that will only send a packet to an interface it is willing to use. In other words, the packet scheduler must faithfully follow Π.

2. **Work-conserving/Pareto efficient.** In this context, Pareto efficiency means giving rates to flows such that *it is not possible to increase the rate of one flow without decreasing the rate of another flow.* In other words, we want to maximize the total number of packets scheduled, without wasting capacity.

3. **Meet rate preferences, where possible.** We want a packet scheduler that implements the relative priorities of flows encoded by weights ϕ. As pointed out by the example in Section 1, interface preferences can make rate preferences infeasible unless we violate work-conservation. In the case where the rate preferences are feasible, we want the scheduler to always faithfully follow them. In the case where they are not feasible, we first meet the rate preferences subject to the interface preferences, and then use up any leftover capacity serving flows that can use it. This means some flows will receive a higher rate than they would if we capped them at their rate preference. But the key is that no flow will be made worse off; it will only benefit from extra capacity made available to it because other flows were unwilling to use all the interfaces.

4. **Use new capacity.** If we add an interface, we should use it to increase capacity for all flows willing to use it. When a flow ends, other flows sharing its set of interfaces should benefit from the freed up capacity.

One might assume that giving each flow its weighted fair share on a per-interface basis is the correct overall weighted fair share. Such a definition would in fact provide some of the properties we want (e.g. Pareto efficiency). But it would fail to provide rate preferences even when they are feasible. We would also need to figure out the correct weights for each flow on each interface.

Our goal is to provide service to the flows, that can each be served by multiple interfaces. What the user and application really care for is the rate achieved by each flow over whichever number of interfaces serving it. Therefore, it is natural to consider the weighted fair share on an aggregate basis, where the rate allocation for each flow i ($r = [r_i]$) is weighted max-min fair.

2.1 Why earliest finishing time does not work?

For a *single* interface, weighted fair queueing, through algorithms like Packetized Generalized Processor Sharing (PGPS) [10] fulfills all of the above properties by providing each flow with its weighted fair share rate, r_i/ϕ_i. PGPS is known to be max-min fair for a single interface.

DEFINITION 1. *Max-min fair rate allocation is a rate allocation where no flow can get a higher rate without decreasing the rate of another flow that has a lower or equal allocation.*

Because max-min fair is a special case of Pareto efficiency, PGPS is Pareto efficient.

The PGPS algorithm meets all our goals for a single interface by assigning a finishing time to each packet when it arrives, and then uses the simple strategy of sending the

packet with the *earliest finishing time*. With one interface, PGPS is work-conserving and can faithfully provide the user's weighted preference between flows. If the user asserts a preference that "flow a should receive twice the rate of flow b" then we simply set the weight of a to be twice the weight of b and PGPS will provide the right allocation. We might wonder if we can define a finishing time across *multiple* interfaces, taking into consideration usage preferences, and then for each interface schedule the packet with the earliest finishing time. We now prove that although intuitively simple, this strategy is not possible with a causal algorithm. Unlike the single interface case, the algorithm must know about future packet arrivals.

Our proof is by counter-example. Consider the example illustrated in Figure 1(c), where flows a and b share interface 2 while only flow a can use interface 1. Let head of line packets of the flows at $t = 0$ be p_a and p_b respectively (with lengths $L/2$ and L bits respectively) and all flows have equal priority. Both interfaces run at 1 Mb/s. At $t = 0$ when interface 2 becomes available, it must decide if p_a or p_b would finish first under PGPS. Consider the following two scenarios:

1. If no new flows arrive after $t = 0$, each flow would get rate 1 so the finishing times of p_a and p_b would be $f_a = L$ and $f_b = L/2$ respectively and p_b will finish first.

2. Assume three new flows arrive shortly after $t = 0$ and they are only willing to use interface 2. Rather than compete for interface 2, flow a will continue to use interface 1 and its rate will remain at 1 Mb/s. Meanwhile flow b's rate reduces to 1/4 Mb/s. In this case, p_a would finish first.

Since the finishing order of the packets in these two scenarios is different and the packet scheduler cannot causally determine which scenario would occur, it cannot determine the relative finishing order of the packets. This leads us to Theorem 1.

THEOREM 1. *In the presence of interface preferences, a packet scheduler cannot always causally determine the relative order of packet finishing time.*

Now consider the same example, but without interface preferences, i.e., both flows a and b are willing to use all interfaces available as illustrated in Figure 1(b). In this case, packet p_b will always finish first. Even if three new flows arrive shortly after $t = 0$, both flows a and b would be both slowed down to a rate of $\frac{2}{5}$ Mb/s because the new flows are also willing to use both interfaces. The proportional change in rate in flows a and b, i.e., *fate-sharing* among the flows, allows us to know the relative finishing order of the packets at the time of their arrival.

The key difference in scheduling packets with and without interface preferences—and the reason prior work fails to help—is that fate-sharing is no longer true with interface preferences. Without interface preferences, if the number of active flows changes, or if the capacity of an interfaces changes, all flows are equally affected and share the same fate. With interface preferences, changes affect flows using one interface more than others.

3. OUR PACKET SCHEDULER

We now describe a simple and novel work-conserving algorithm to schedule packets over multiple interfaces with interface preferences. The scheduler takes as input from the user (or more specifically from the system managing user preferences) the interface preferences specifying the interfaces each application can use, and the rate preferences specifying the relative flow rates. Both the interface preference matrix Π and rate preference vector ϕ are inputs to the scheduling algorithm.

Knowing that we cannot use a scheduler that calculates finishing time in advance, could we turn to reactive scheduling mechanisms such as Deficit Round Robin (DRR) [13]? The idea behind DRR, summarized in Algorithm 3.1, is to serve each flow the same number of times by serving them in a round robin fashion. To provide rate preferences, the number of bits served during each turn—known as the quantum—is adjusted accordingly. To ensure fairness over time while sending packets integrally, DRR maintains a per-flow deficit counter, which is essentially a metric for how much service this flow has earned but has not been provided over time. This seems to avoid the problems of causality, since we make no assumptions about the interface rates and we start deficit counting only after flows arrive.

However, a naive implementation of DRR on each interface does not work either. In our simple example in Figure 1(c) DRR would give the same rate allocation as WFQ (flow a and b would get 0.5 Mb/s and 1.5 Mb/s respectively) whereas we know there is a feasible max-min allocation of 1 Mb/s per flow.

The underlying problem is that if a flow is willing to use more than one interface, when an interface schedules a packet it has no way of knowing what rates the flows are getting from *other* interfaces. Having this information is crucial to ensure max-min fairness. An obvious solution is for interfaces to exchange information about the rates flows are receiving from every interface. This seems to require the algorithm to keep track of the rates provided to each flow, and when it is making its own packet scheduling decision it would decide whether or not servicing that flow leads to a max-min fair solution. As the reader can guess, this scheme would require an impractical amount of state information to be maintained and exchanged, as well as interfaces to know their own instantaneous rates. Both of these are problematic requirements, especially on mobile devices. So how might one achieve a max-min fair rate allocation that can be calculated independently on each interface, without ever having to calculate and exchange the actual achieved rates between all the interfaces?

3.1 Multiple Interface Deficit Round Robin

The key contribution of this paper is a generalization of the deficit round robin scheduling algorithm that achieves max-min fairness over multiple interfaces with interface preferences, *while requiring almost no coordination among the interfaces.* Specifically, it requires no rate computations and at most one bit of coordination signaling from each interface for every flow. The one bit is a boolean service flag, and there is one flag at an interface for every flow. The flag indicates whether a flow has been serviced recently by another interface. When an interface considers servicing a flow, it skips it if the service flag is set. We show that this simple mechanism and minimal book-keeping achieves a max-min

Symbol	Description
BL_i	Backlog of flow i
$Size_i$	Size of flow i's head-of-line packet
Q_i	Quantum for flow i
DC_i	Deficit counter for flow i
$\mathcal{F}_j$	Set of flows willing to use interface j
$\mathcal{C}_j$	Current flow interface j is serving
$\mathcal{B}$	Set of backlogged flows
SF_{ij}	Interface j's service flag for flow i (Service flags for new flows are initiated at zero.)

Algorithm 3.1: DRR(j)

if $\mathcal{F}_j \cap \mathcal{B} = \emptyset$
  then return
$i = \mathcal{C}_j$
if $Size_i \leq DC_i$
  then $\begin{cases} \text{Send } Size_i \text{ bytes} \\ DC_i = DC_i - Size_i \end{cases}$
if $BL_i = 0$
  then $\begin{cases} DC_i = 0 \\ \text{Remove } i \text{ from } \mathcal{B} \end{cases}$
if $BL_i = 0$ **or** $Size_i > DC_i$
  then $\begin{cases} i = C_j = \text{Next backlogged flow for } j \\ DC_i = DC_i + Q_i \end{cases}$

Algorithm 3.2: miDRR-Check-Next(i, j)

$i = C_j =$ Next backlogged flow for j
while $SF_{ij} \neq 0$
  do $\begin{cases} SF_{ij} = 0 \\ i = C_j = \text{Next backlogged flow for } j \end{cases}$
$SF_{ik} = 1 \quad , \forall k \neq j$
return (i)

Table 1: Pseudocode for DRR and miDRR which is invoked when interface j is free to send another packet. The only difference between the two algorithms is that the highlighted line in Algorithm 3.1 is replaced by Algorithm 3.2 in miDRR.

fair allocation when we have interface preferences. By obviating the need to exactly track service rates and—as we shall see—implicitly enforcing the relative rates between any pair of flows, the service flag allows miDRR to be scalable and highly-distributed by minimizing the overhead of communication between interfaces.

The bareness of the mechanism is surprising. How can a single flag be sufficient to let us achieve a max-min fair rate when we do not even know the rates of each interface, nor do we know the rates achieved by the flows themselves? The insight is that to ensure max-min fairness, it is sufficient to know only the relative rates achieved between flows, the absolute value is not needed. Further, each interface only needs to know the relative rates achieved among flows it is allowed to service according to the interface preferences. Finally, we do not even need to know a precise value of the relative rate. The packet scheduler only needs to check if a particular flow's rate is higher than at least one other flow it is servicing on the same interface. If so the scheduling decision is simple: it should not service the flow with relatively higher rate. If it iteratively applies the above condition to all its flows, it will eventually service the flow that will push it towards a max-min fair rate allocation overall as we show formally in the next section. Below, we describe how the algorithm operates.

Interface j maintains one service flag SF_{ij} for each flow i that it serves. The flag is for other interfaces to indicate to interface j that flow i has been serviced recently. Maintaining this boolean service flag requires two tasks:

1. When interface k serves flow i, it sets service flags $SF_{ij} \forall j \neq k$ to tell the other interfaces that flow i has been served.

2. When interface j considers flow i for service, it resets service flag SF_{ij}.

Race conditions on the service flag can easily be handled by a standard mutex. More importantly, maintaining the service flag does not require us to keep track of the rate at which each interface serves each flow. Nor do we need to keep track of how much other interfaces serve a flow.

We refer to this algorithm as *multiple-interface Deficit-Round-Robin (miDRR)*. In essence, the algorithm entails each interface implementing DRR independently with the slight modification of checking the service flag before serving the flow. The algorithm is summarized in the pseudocode described in Table 1.

To better understand how the algorithm works, we return to our example of two interfaces and two flows as shown in Figure 1(c). Say interface 2 is free and is now moving on to serve flow b. It would find its service flag with flow b to be not set because no other interface is serving flow b, and it would therefore proceed to serve flow b. As prescribed by DRR, the deficit counter of flow b (DC_b) would be incremented by its quantum Q_b, and potentially one or more flow b packets would be sent. When a packet is sent, its length is deducted from the deficit counter. Flow b would be served until its deficit counter is insufficient for the next packet. At that point, interface 2 will move on to flow a. Because interface 1 is serving flow a at the same rate, interface 2 will find its service flag with flow a set by interface 1. Given the service flag is set, interface 2 will not serve flow a. Instead it would move back to flow b after resetting its service flag for flow a. As this algorithm continues, interface 1 will only serve flow a and interface 2 will only serve b, yielding the desired result.

The takeaway is that each interface does its own DRR packet scheduling, while checking and communicating through the service flags.

4. PROPERTIES OF miDRR

The most important property of miDRR is that it achieves the (weighted) max-min rate allocation, subject to the user's preferences of interface and rates; a result we prove below in Theorem 3. The desired properties in Section 2 follow directly from the weighted max-min fairness (a property which miDRR inherits from DRR). Specifically,

1. **Meet interface preferences.** Clearly miDRR meets this by design. The user sets values π_{ij} in the algorithm to represent interface preferences and a flow is never scheduled on an interface for which $\pi_{ij} = 0$.

2. **Pareto efficient.** This follows directly because miDRR gives a weighted max-min rate allocation (Theorem 3).

3. **Meet rate preferences, where possible.** For a single interface j, miDRR guarantees that each flow actively served by it is served exactly once in the period it takes the interface to successively visit the same flow. By picking the quanta Q_i flow i will receive a share $\phi_i / \sum_{k,\pi_{kj}=1} \phi_k$ of the outgoing line. If $2Q_a = Q_b$ for flows a, b actively served by interface j, then $r_b = 2r_a$ which meets the designated rate preference.

4. **Use new capacity.** If new capacity becomes available, then we want to know miDRR will use it. There are three reasons more capacity becomes available: a new interface comes online, the rate of an interface increases, or a flow ends freeing up capacity. In each case, if extra capacity becomes available on an interface, this capacity can be used by the flows willing to use the interface. In turn, these flows might free up yet more capacity on other interfaces, that in turn can be taken up by other flows, and so on. A max-min fair solution will maximize the minimum rate in the allocation, and so although some flows may receive a higher rate, no flow will receive a lower rate.

4.1 miDRR is max-min fair

We now prove the main result: miDRR leads to a (weighted) max-min fair allocation. Our proof will proceed in two parts.

1. We introduce the *rate clustering property* (defined in Definition 2) and prove that any scheduler satisfying this property is max-min fair.

2. We show that miDRR fulfills the rate clustering property, and therefore miDRR is a max-min fair scheduler.

DEFINITION 2 (RATE CLUSTERING PROPERTY). *A scheduler satisfies the* rate clustering property *if*

1. *It splits the union set of flows and interfaces into disjoint clusters, where each flow and each interface can only belong to a single cluster.*

2. *Within a cluster $\mathbb{C}_i$, all flows are served at the same rate (by the interfaces also in $\mathbb{C}_i$). i.e,*

$$a, b \in \mathbb{C}_i \implies r_a = r_b.$$

3. *Among the clusters containing an interface flow a is willing to use, flow a will* only *belong to the cluster with the highest rate, i.e.,*

$$a \in \operatorname*{arg\,max}_{C_i, \exists j \in C_i, \pi_{aj}=1} r(\mathbb{C}_i),$$

where $r(\mathbb{C}_i)$ is the rate cluster $\mathbb{C}_i$ serves its flows.

Any scheduler satisfying the rate clustering property is max-min fair. Intuitively, from the perspective of an arbitrary flow a, the rate clustering property divides flows and interfaces into three distinct sets of clusters. The first set comprises clusters that do not have any interface flow a is willing to use. The second is the cluster where flow a belongs. The third set comprises clusters that flow a can possibly belong to, but does not.

Recall from Definition 1 that in a max-min fair allocation no flow can get a higher rate without decreasing the rate of another flow that has a lower or equal allocation. Let us consider how flow a could get a higher rate. Clearly it cannot get a higher rate by using any interface in the first cluster. If it increases its rate by getting more from its own cluster, the rate clustering property tells us that flows from the same cluster all have the same rate as flow a and therefore increasing its rate will decrease the rate of a flow of equal allocation. Similarly, since flow a belongs to the cluster with the highest rate, flows belonging to the third set of clusters have a lower (or equal) rate than flow a. If flow a gets any rate from the third set it will decrease one of a lower or equal allocation. Hence, the rate clustering property ensures that the allocation is max-min fair.

It turns out that the rate clustering property is not only sufficient but also necessary for a max-min fair scheduler. We now formally prove this in Theorem 2.

THEOREM 2. *A work-conserving system is max-min fair* if and only if *the following conditions are satisfied.*

1. *If flows i and j are actively serviced by a common interface (i.e., in the same cluster), their allocated rate is the same, i.e.,*

$$\exists k, \quad i, j \in \mathcal{U}_k \implies r_i = r_j,$$

where $\mathcal{U}_k = \{i, r_{ik} > 0\}$.

2. *If both flows i and j are willing to use interface k, but only flow i is actively using it (meaning the flows are in different clusters), the rate allocated to flow j must be greater than or equal to that of flow i, i.e.,*

$$\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies r_j \geq r_i,$$

where $\mathcal{F}_k = \{i, \pi_{ik} = 1\}$.

To prove the theorem we begin with a (self-evident) lemma on the Pareto efficiency of a work-conserving system.

LEMMA 1. *In a work-conserving system, no flow can increase its allocation without decreasing another's allocation, i.e.,*

$$\delta_i > 0 \quad \implies \quad \exists \delta_j < 0,$$

where δ_i is the change in flow i's allocation.

In other words, if an allocation is *not* max-min fair, there must exist flows i and j where decreasing the flow with larger allocation will increase the other flow's allocation. This leads to our next lemma on the sufficient conditions for max-min fairness.

LEMMA 2 (SUFFICIENT CONDITION).
In a work-conserving system, the following conditions (as listed in Theorem 2) imply that the allocation is max-min fair.

1. $\exists k, \quad i, j \in \mathcal{U}_k \implies r_i = r_j$.

2. $\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies r_j \geq r_i$.

PROOF. *Assume the opposite; i.e. both conditions are always true, but the system is not max-min fair. Because the system is work-conserving, there is no idle capacity if any flow is backlogged. From Lemma 1, for the system to not be max-min fair, there must exist flows i and j such that j can increase its allocation by decreasing i's while $r_i > r_j$.*

This exchange of allocation can happen in two ways:

1. *The exchange occurs on interface k, i.e., r_{jk} is increased while r_{ik} is decreased. This means $r_{ik} > 0$ and $a \in \mathcal{U}_k$. If $j \in \mathcal{U}_k$, then $r_i = r_j$ by the first condition. Else j must at least be in $\mathcal{F}_k$ for r_{jk} to be increased to a non-zero amount. This means $r_j \geq r_i$ by the second condition. In either case, it contradicts the requirement that $r_i > r_j$.*
2. *The allocation could be exchanged through a series of intermediary flows. Denote the n intermediary flows involved as flow $1, 2, \cdots, n$ where $n > 0$. This means flow i would exchange allocation with flow 1, which in turn passes the allocation to flow 2, and so on. Flow i must share a common interface with flow 1, and the above arguments must hold. This means $r_1 \geq r_i$, $r_2 \geq r_1$ and so on. Putting this together we see $r_i \leq r_1 \leq r_2 \leq \cdots \leq r_n \leq r_j$, which contradicts $r_i > r_j$.*

Hence, the allocation must be max-min fair if the conditions are true. □

We can show that these conditions are also necessary, as detailed in [18].

Proof of Theorem 2. Putting the lemmas together, we have Theorem 2. The result tells us why miDRR can be so simple; all it needs to do is maintain the rate clustering property and it will lead to a max-min fair allocation, without having to keep track of the absolute service rate for each flow. We are now ready to prove the main theorem. □

miDRR fulfills the rate clustering property and therefore is max-min fair. To understand this, we take the perspective of an interface j running miDRR. Among the flows willing to use interface j, interface j will serve a subset of these flows at the same rate (which is what DRR does for a single interface). The flows served by interface j are in the same cluster.

Say flow a—among the flows willing to use interface j—is not served by interface j. Flow a must have its service flag with interface j set at least once every τ_j, where τ_j is the time difference between successive visits by the interface j scheduler to schedule flow a. This means flow a is being served as often or more often than a flow served by interface j, which is served once every τ_j. Flow a is therefore in a different cluster that is at a higher rate.

Hence, with miDRR interfaces serve the flows in their cluster at the same rate, and flows being passed over by interfaces in a cluster are served by another cluster at a higher or equal rate. Therefore miDRR fulfills the rate clustering property.

We will now formally prove this in Theorem 3.

Theorem 3. *miDRR provides a (weighted) max-min fair allocation.*

We begin by extending the definition of fairness metric proposed in [13].

Definition 3. *Let directional fairness metric from flow i to flow j be*

$$FM_{i \to j}(t_1, t_2) = S_i(t_1, t_2)/\phi_i - S_j(t_1, t_2)/\phi_j,$$

where $S_i(t_1, t_2)$ is the number of bytes sent by flow i in the time interval $(t_1, t_2]$, and ϕ_i is the priority of flow i. Alternatively, we say flow i has received service of $S_i(t_1, t_2)$ in time interval $(t_1, t_2]$.

Consider the system in steady state, i.e., (1) all flows are continuously backlogged in $(t_1, t_2]$, (2) no new flows arrive in this interval, and (3) the rate of the interfaces does not change. The conditions for weighted max-min fairness in Theorem 2 can be rewritten in terms of the directional fairness metric FM as:

1. If flows i, j are actively serviced by a common interface k, the directional fairness metric from i to j and vice versa are zero, i.e.,
$$\exists k, \quad i, j \in \mathcal{U}_k \implies FM_{i \to j} = FM_{j \to i} = 0.$$
2. If flow i is actively serviced by some interface k while flow j is willing to use interface k but is not actively using it, then the directional fairness metric from j to i is greater or equal to zero, i.e.,
$$\exists k, \quad i \in \mathcal{U}_k, j \in \mathcal{F}_k \implies FM_{j \to i} \geq 0.$$

This means that if a packet scheduler maintains these conditions on FM at all times, it will be max-min fair. We show this is the case by finding upper bounds on FM, which in turn tells us the flow service allocations will be fair, amortized over time.

The following Lemma 3 tells us that the value of the deficit counter in miDRR is bounded

Lemma 3. *At the end of each service turn for a flow, its deficit counter is greater or equal to zero and less that the maximum packet size, i.e.,*

$$0 \leq DC_i < MaxSize'_i,$$

where $MaxSize'_i$ is the maximum packet size of flow i.

Proof. *Straightforward given [13], hence omitted.* □

From the bound on the deficit counter we can now bound the amount of service received by a flow:

Lemma 4. *Consider a time interval $(t_1, t_2]$ where flow i is continuously backlogged. Let m be the number of service turns it receives in this interval. The service received by flow i can be bounded by*

$$mQ_i - MaxSize'_i < S_i(t_1, t_2) < mQ_i + MaxSize'_i.$$

Proof. *Straightforward given [13], hence omitted.* □

Lemma 3 bounds the deficit counter (and hence, the amount of service) that a flow can carry over from one service turn to another (as seen in Lemma 4). This ensures that a flow cannot accumulate unfair service from one turn to another which lays the foundation for the following two lemmas.

Lemma 5. *Consider pair of flows i, j where flow i is serviced at higher rate than flow j and flow j is serviced by interface k. It can be shown that*

$$FM_{i \to j} > -2MaxSize',$$

where $MaxSize'$ is length of the maximum sized packet.

Proof. *Consider a service turn on interface k where flow j is serviced by the interface but not flow i. It means flow*

i has been served once or more between the time it is considered by interface k, i.e., $SF_{ik} = 1$. *Hence, the number of service turns flows* i, j *receives in* $(t_1, t_2]$ *is related by*

$$m_i(t_1, t_2) \geq m_j(t_1, t_2).$$

Using Lemma 4, we know that

$$\begin{aligned} FM_{i\to j} &= S_i(t_1,t_2)/\phi_i - S_j(t_1,t_2)/\phi_j \\ &> \left(m_i(t_1,t_2)Q_i + MaxSize'_i\right)/\phi_i \\ &\quad - \left(m_j(t_1,t_2)Q_j + MaxSize'_j\right)/\phi_j \\ &> \left(m_i(t_1,t_2) - m_j(t_1,t_2)\right)Q' \\ &\quad -MaxSize'_i - MaxSize'_j \\ &> -2MaxSize' \end{aligned}$$

where $Q' = Q_i/\phi_i$ *and we can choose* $\min_i \phi_i \geq 1$ *without of generality.* □

Lemma 6. *Consider pair of flows* i, j *where both flows are always serviced by interface* k. *It can be shown that*

$$|FM_{i\to j}| < Q' + 2MaxSize',$$

where $MaxSize'$ *is length of the maximum sized packet, and* $Q' = Q_i/\phi_i$.

Proof. *Consider a service turn of flow* j. *From the algorithm,* $SF_{ik} = 0$ *for flow* i *to be also serviced. Hence*

$$|m_i(t_1,t_2) - m_j(t_1,t_2)| = 1.$$

From Lemma 4,

$$S_i(t_1,t_2)/\phi_i < m_i(t_1,t_2)Q' + MaxSize'_i/\phi_i$$

and

$$S_j(t_1,t_2)/\phi_i < m_j(t_1,t_2)Q' + MaxSize'_j/\phi_j.$$

This means

$$\begin{aligned} &|FM_{i\to j}| = |FM_{j\to i}| \\ &= |S_i(t_1,t_2)/\phi_i - S_j(t_1,t_2)/\phi_i| \\ &< |m_i(t_1,t_2) - m_j(t_1,t_2)|Q' \\ &\quad + MaxSize'_i/\phi_i + MaxSize'_j/\phi_j \\ &< Q' + 2MaxSize' \end{aligned}$$

because we can choose $\min_i \phi_i \geq 1$ *without of generality.* □

Proof of Theorem 3. We are now ready to prove Theorem 3 that miDRR gives a max-min fair allocation. Lemma 5 shows that the service lag of a faster flow compared to a slow flow is strictly bounded by two maximum size packets. Lemma 6 shows that the service difference between two flows that are supposed to be served at the same rate is strictly bounded by Q' plus two maximum size packets. Both lemmas prevent accumulation of unfairness over time, showing us that miDRR indeed provides fair scheduling for multiple interfaces in the presence of interface preferences. □

In summary, our observation that the earliest finishing packet cannot be causally determined in a multi-interface setting due to a lack of fate-sharing led to the notion that we must track the conditions under which flows experience different relative service rates. The formation of clusters leads to sufficient conditions for max-min fair packet scheduling that describe exactly when flows experience the same or unequal rates. miDRR achieves a max-min fair allocation by serving flows at rates that adhere to the conditions in Theorem 2 by means of the service flag.

4.2 Why is a 1-bit service flag sufficient?

While Theorem 2 formalizes the notion that the rate clustering property serves as sufficient conditions for max-min fair packet scheduling, it is worth understanding intuitively why this is so. It all boils down to the fact that the service flag implicitly captures the relative service rate between a pair of flows.

At an intuitive level, consider flow a served only by interface j. If Q_a is the DRR quantum of flow a in bits and τ_j is the time between successive visits by the interface j scheduler to schedule flow a, then flow a is served at rate Q_a/τ_j. Assume for a moment that all the flows using interface j have the same weight (i.e. the same ϕ values), then all of them have the same service rate. Now consider a second flow b that is willing to use interface j but is not actively being scheduled by j because it is receiving enough service elsewhere. This is accomplished if flow b has its service flag SC_{bj} set at least once every τ_j so that interface j will skip serving it every time. This will happen if (and only if) flow b is already being served as often or more often than τ_j by other interfaces. Which means it needs no service at interface j.

One last word on rate allocations. Theorem 3 tells us that miDRR automatically figures out the max-min fair allocation. It is also worth asking if we can find the max-min fair allocation *without* running the algorithm. It is quite straightforward and can posed as a convex program [18], with the interface and usage preferences used to indicate constraints on the system.

5. APPLYING miDRR

Can we use miDRR to implement user preferences in a practical mobile device? Specifically, we want to use miDRR to schedule both incoming and outgoing packets over the multiple interfaces modern mobile devices have.

It is relatively simple to implement miDRR to schedule outgoing packets from a mobile device. Our current implementation uses a custom Linux kernel bridge, that is written using 1,010 lines of C code. Our implementation, in Linux 3.0.0-17, follows the architecture illustrated in Figure 3. This custom bridge allows us to steer individual packets to whichever interface the scheduling algorithm chooses, while being transparent for the applications [20]. This is done by presenting the applications with a virtual interface with an arbitrarily chosen address and then rewriting the packet headers appropriately before transmission. We find that the overhead of this bridge and packet header rewriting operations are negligible when compared to the relatively moderate speeds found in wireless interfaces.

The twin problem of scheduling *incoming* packets over multiple interfaces is harder. The ideal implementation would be a proxy in the Internet that collects all the flows headed towards a mobile device at a location close to the last-mile wireless connections used by the device as in Figure 4. This can be accomplished by a service provider like AT&T if the device is connected to the provider's HSPA, LTE and WiFi networks at the same time. The scheduler at this proxy could then use miDRR to schedule all the flows on the different paths that lead to the different interfaces on the mobile device while respecting preferences. Clearly this has deployability and performance challenges since it

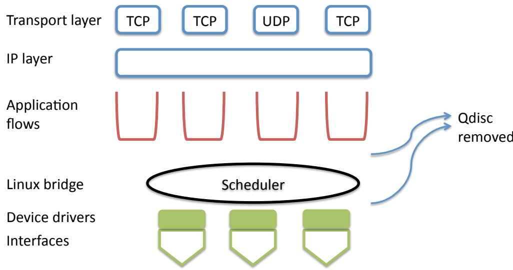


Figure 3: Implementation of miDRR in Linux kernel to schedule outbound packets.

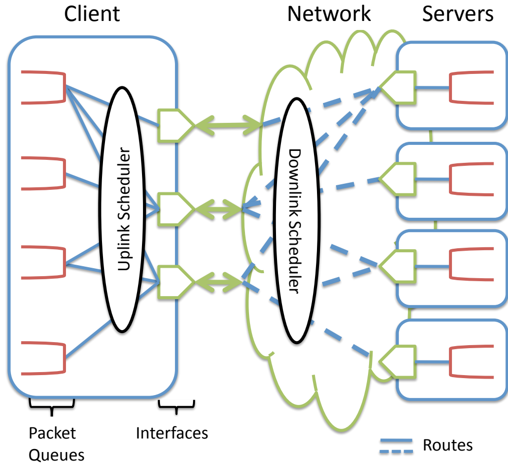


Figure 4: Ideal implementation of miDRR to schedule both inbound and outbound packets.

will require us to aggregate all traffic at one point before scheduling.

In this paper, to ease deployment concerns, we use a HTTP-based scheduling technique that can be implemented on the mobile device itself and still allows us to come close to ideal packet scheduling for incoming packets. Specifically, we implemented miDRR to schedule inbound HTTP traffic in a HTTP proxy as depicted in Figure 5. Our HTTP proxy is written in 512 lines of Python code. Since the majority of packets to and from mobile devices are HTTP transfers [4], we feel that this is a reasonable compromise. Furthermore, such a fully in-client HTTP scheduler is easily introduced into today's mobile device.

In this implementation, we make use of the byte-range option available in HTTP 1.1 to divide a single GET request into multiple requests and deciding an interface to send each request on. This allows us to divide an inbound transfer into multiple parts, each of which can arrive over different interfaces at the same time. The responses are then collected, spliced together and returned to the application. By choosing which interface a request is put on, we also select the interface over which the corresponding inbound data will arrive. When combined with request pipelining, we can always have some pending requests on each interface making sure that all the available capacity is utilized. To provide fairness between the HTTP flows, we implemented miDRR to regulate the inbound HTTP traffic. For example, the proxy can choose to send flow a's requests while withholding flow b's to make sure each flow gets their fair share of the bandwidth.

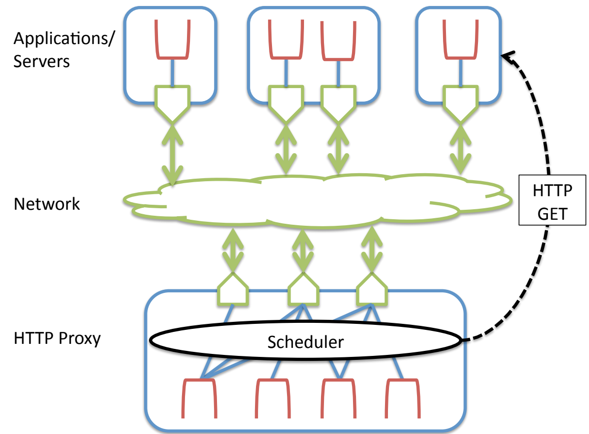


Figure 5: Implementation of miDRR in a HTTP proxy to schedule inbound HTTP flows.

6. EVALUATION

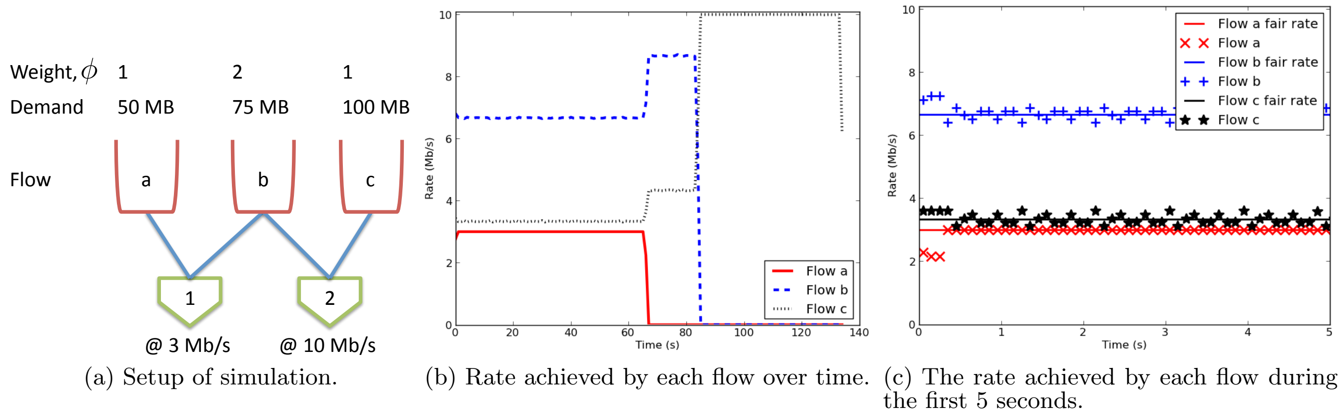
We evaluate miDRR using our prototype implementation on a Dell laptop running Ubuntu 10.04 LTS with an Intel Core Duo CPU P8400 at 2.26 GHz processor and 2 GB RAM. We aim to evaluate whether miDRR can provide a max-min fair allocation on both the uplink and the downlink with varying network conditions and traffic workloads. In our experiments we use between two and sixteen interfaces to test miDRR. For the HTTP experiment, the interface NICs are either the inbuilt Intel PRO/Wireless 5100 AGN WiFi chip or an Atheros AR5001X+ wireless network adapter connected via PCMCIA. We use the Atheros `ath5k` driver for the Atheros wireless adapter.

6.1 Number of concurrent flows in mobile

Before we evaluate miDRR using our implementations, an important question is how many concurrent flows exist in a typical mobile device at any instant. For example, if we find at most 1 or 2 flows, then the scheduling problem is fairly simple. Unfortunately such flow statistics are hard to find from a wide set of users. Hence to get a rough idea of the number of concurrent flows, the authors instrumented their own Android devices for a week. Focusing only on the active periods (i.e., when there is at least one ongoing flow), we found that 10% of the time, we have 7 or more ongoing flows; the maximum number of concurrent flows hit a maximum of 35 in our log. The cumulative distribution function is shown in Figure 7.

6.2 Fair packet scheduling with miDRR

As shown in Theorems 5 and 6, miDRR provides weighted max-min fair scheduling, but it can deviate from an ideal bit-by-bit max-min fair scheduler. To test how far it can deviate, we check the performance of miDRR in a simulation that allows us to avoid complications such as time-varying network conditions. We run the simulation using an example of three flows served by two interfaces as illustrated in Figure 6(a).



(a) Setup of simulation. (b) Rate achieved by each flow over time. (c) The rate achieved by each flow during the first 5 seconds.

Figure 6: Simulation results for three flows over two interfaces.

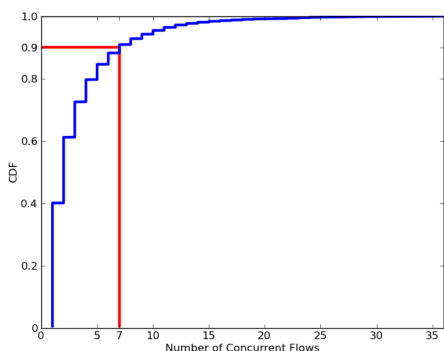


Figure 7: CDF of the number of concurrent flows on our smartphones.

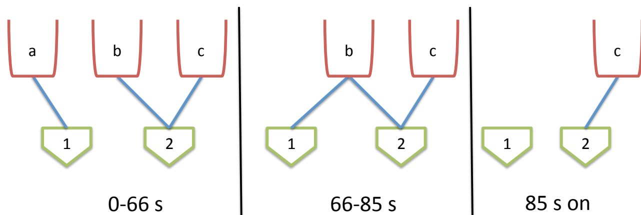


Figure 8: Clusters formed through the experiment in chronological order from left to right.

The result is shown in Figure 6(b). As we can see flow a achieved a rate of 3 Mb/s using interface 1, while flows b and c shared the 10 Mb/s on interface 2 in ratio of their weights 2:1. This is the weighted fair allocation we expect. When flow a completed after 66 s, flow b immediately increases its rate to 8.67 Mb/s using interfaces 1 and 2 simultaneously, i.e., aggregating bandwidth across both interfaces. Similarly, flow c's rate increases to 4.33 Mb/s which further increases to 10 Mb/s when flow b completed after 85 s. This shows that miDRR indeed provides weighted fair scheduling.

Throughout the experiment, the rate clustering property was upheld. We illustrate the clusters formed in Figure 8. As load changes (flows a and b finish) clusters change as well. Zooming at the first phase (0–66 s), flow a's cluster has rate 3 Mb/s and flow c's cluster a rate of 3.33 Mb/s. Flow b gets twice this rate (6.66 Mb/s) since its weight is twice of flow c in the same cluster.

However like many practical approximations, miDRR is imperfect. Figure 6(c) zooms in on the first 5 s. We can see that flow a initially only receives about 2 Mb/s (instead of 3 Mb/s) while interface 1 serves flow b. However the algorithm quickly corrects this and a weighted fair rate is achieved. Also, the rate achieved by flows a and b fluctuates around the ideal fair rate due to the atomic nature of packets and the size of the quanta.

6.3 Overhead of miDRR

We designed miDRR to be lightweight and simple, with each interface performing packet scheduling in a fairly independent manner with minimum communication overhead. But what exactly is the overhead of miDRR in our implementation? To answer this question, we profiled our Linux kernel bridge—measuring the time it takes to make a scheduling decision. For each experiment we present the bridge with 1,000 packets spread and queued across all the flows and record the time it takes to make a scheduling decision for each packet. We present the bridge with 4 to 16 virtual Ethernet interfaces.

The scheduling time is independent of the number of flows, because the algorithm does not need to go through every flow to make a scheduling decision. The decision is made as soon as the scheduler finds the next flow it should serve. However, the scheduling overhead increases with the number of flows an interface has to consider before finding one that it should serve. This number is proportional to the number of flows that have their service flags set, which is correlated with the number of interfaces. With more interfaces in the system, the chances of finding a flow with its service flags set would increase. This extra search time is shown in Figure 9. For example, interface 1 might be free after finished serving flow a. The scheduler finds that flows b, c and d have been served by other interfaces which means it should serve flow e.

Even with 16 network interfaces, miDRR can make a scheduling decision in less than 2.5 μs. Put differently, the algorithm can support a traffic rate over 3 Gb/s for 1,000-byte packets. Therefore, we believe that it is quite feasible to implement miDRR in practice, even for a high speed packet scheduler in the kernel.

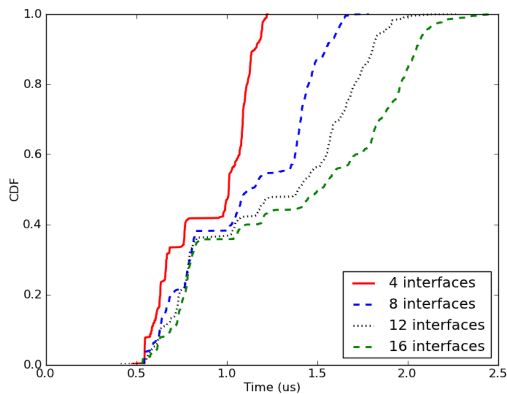


Figure 9: CDF of scheduling time as a function of the number of interfaces.

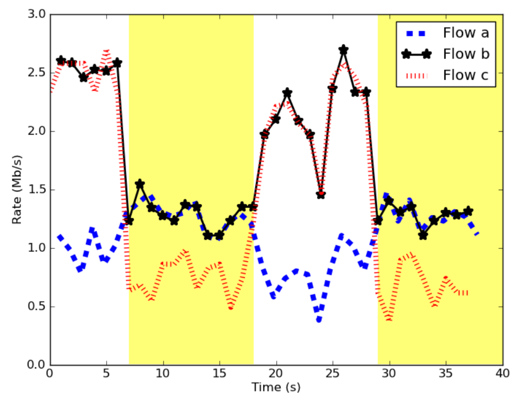


Figure 10: TCP goodput of three inbound HTTP flows scheduled fairly using our HTTP proxy.

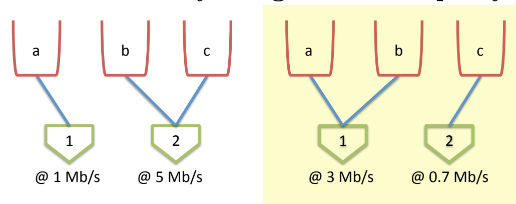


Figure 11: Clustering formed when our HTTP proxy schedules fairly across multiple interfaces. On the left is the clustering during the 11–18 s of the experiment and 29 s on. On the right is the clustering during 0–11 s and 18–29 s.

6.4 HTTP Fair Scheduling

The HTTP proxy based implementation of miDRR on the downlink cannot support fine-grained packet scheduling, but it does allow us to evaluate the algorithm over an operational network. We check if even under such scenarios our scheduler can provide fairness. We evaluate this by serving three HTTP flows over two interfaces. The setup is similar to Figure 6(a) except that all flows have the same weight and the interface speed varies as the experiment runs.

We plot the goodput achieved by each flow over time in Figure 10. In this setup, flows a and c will achieve whatever interface 1 and 2 can respectively provide at the current speed of the interfaces. However flow b is willing to use both interfaces, so it should always achieve the same rate as the faster flow. This is the correct max-min fair allocation. We indeed observe this behavior in Figure 10, i.e., the rate of flow b always tracks the faster flow. We are observing the rate clustering property because flow b will share the faster interface equally with the faster flow, forming a cluster with it as illustrated in Figure 11.

Despite having only very coarse grained control over the inbound traffic, we find surprisingly that the HTTP scheduler is able to provide fair scheduling over multiple interfaces while conforming to interface preferences. This performance holds even while the scheduler is reacting to fluctuating link capacities. Given that a large fraction of the traffic on mobile devices is HTTP, this suggests that an HTTP layer scheduler is sufficient to build a full system that allows users to leverage all their interfaces while respecting preferences.

7. RELATED WORK

Researchers have long been looking for ways to efficiently make use of all the networks around us to serve the applications at hand. The potential payoff can be significant, as shown by prior work [14, 19]. Unsurprisingly, there has been significant effort to facilitate use of multiple interfaces. MultiNet [2] virtualizes a single wireless device into multiple virtual interfaces, allowing more networks to be accessible over the same hardware. MPTCP [15] provides an end-to-end reliable protocol for multiple interfaces; Serval [9] proposes a service access layer that allows services to use multiple interfaces; and [20] exposes an efficient software switch to manage multiple interfaces. These approaches focus on consolidating access to multiple interfaces allowing applications to use them through a simple abstraction. We consider our work complementary to these proposals, as we focus on how traffic is scheduled across multiple interfaces.

Our work extends prior work on fair packet scheduling to account for interface preferences. To do so, we extend classical results in fair scheduling for a single interface [3, 10, 11, 13]. Fair scheduling has previously been extended to the case of link-bonding in [1] where there is no notion of interface preferences, and has subsequently been analyzed using queueing theory [12]. Simple efficient DRR-like algorithms have also been proposed [16, 17]. Our result generalizes these prior works, allowing us not only to compute the max-min fair rate as discussed in [7, 8], but also insights that allow us to build a practical packet scheduler.

Independently, recent work extended fair scheduling along a different dimension. DRF [5, 6] tells us how to schedule fairly over multiple resources. Our work differs mainly in that we consider homogeneous resources (e.g., bandwidth on different interfaces) while DRF considers heterogeneous resources (e.g., CPU and bandwidth). A generalization of both works would further improve our understanding of fair scheduling, but is beyond the scope of this paper.

8. CONCLUSION

The introduction of interface preferences adds a new twist to the classical problem of packet scheduling. Not only do interface preferences render prior algorithms unusable, it challenges our understanding of packet scheduling. Prior packet scheduling algorithms have mostly been defined using service fairness, i.e., by assuring that a pair of flows send an equal number of bits over time. With interface preferences, one flow can receive a higher rate than another in a max-

min fair allocation. Therefore it is possible for one flow to accumulate more service than another, resulting in unequal number of bits being sent over time. This changes the way we think about fairness in rate and in service. This work presents our understanding of that solution space, and our empirical measurements of the algorithm running in practice. Our insights allowed us to design a simple and efficient algorithm that can be used for packet scheduling on mobile devices.

We expect this understanding in general, and our algorithm in particular, to be useful in many applications beyond the mobile application we described. Allocating tasks to machines in data center poses a similar scheduling problem, where certain tasks might prefer to use only more powerful machines. We could also use the algorithm to assign compute tasks to CPU cores in a system such as NVIDIA Tegra 3 4-plus-1 architecture where 4 powerful cores are packaged with a less powerful one. A computation intensive task like graphics rendering might prefer to use only the more powerful cores. As we continue to build large systems by pooling smaller systems together, we expect an increasing number of situations where our results will prove useful.

9. REFERENCES

[1] J. M. Blanquer and B. Özden. Fair queuing for aggregated multiple links. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 189–197, New York, NY, USA, 2001. ACM.

[2] R. Chandra. Multinet: Connecting to multiple ieee 802.11 networks using a single wireless card. In *in IEEE INFOCOM, Hong Kong*, 2004.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, Aug. 1989.

[4] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 281–287, New York, NY, USA, 2010. ACM.

[5] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[6] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association.

[7] N. Megiddo. Optimal flows in networks with multiple sources and sinks. *Mathematical Programming*, 1974.

[8] S. Moser, J. Martin, J. Westall, and B. Dean. The role of max-min fairness in docsis 3.0 downstream channel bonding. In *Sarnoff Symposium, 2010 IEEE*, pages 1 –5, april 2010.

[9] E. Nordstrom, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *Proc. 9th Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, April 2012.

[10] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1:344–357, 1993.

[11] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Trans. Netw.*, 2:137–150, April 1994.

[12] D. Raz, B. Avi-Itzhak, and H. Levy. Fair operation of multi-server and multi-queue systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):382–383, June 2005.

[13] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev.*, 25(4):231–242, Oct. 1995.

[14] C.-L. Tsao and R. Sivakumar. On Effectively Exploiting Multiple Wireless Interfaces in Mobile Hosts. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 337–348, New York, NY, USA, 2009. ACM.

[15] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

[16] H. Xiao. Analysis of multi-server round robin service disciplines, 2005. Master Thesis at National University of Singapore.

[17] H. Xiao and Y. Jiang. Analysis of multi-server round robin scheduling disciplines. *IEICE TRANS.*, 2002.

[18] K.-K. Yap. *Using All Networks Around Us.* PhD thesis, Stanford University, 2013.

[19] K.-K. Yap, T.-Y. Huang, M. Kobayashi, M. Chan, R. Sherwood, G. Parulkar, and N. McKeown. Lossless handover with n-casting between wifi-wimax on openroads. Mobicom '09, 9 2009. Honorable Mention.

[20] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar. Making use of all the networks around us: a case study in android. *SIGCOMM Comput. Commun. Rev.*, 42(4):455–460, 2012.

[21] C. Zhou and N. F. Maxemchuk. Scalable max-min fairness in wireless ad hoc networks. *Ad Hoc Netw.*, 9(2):112–119, Mar. 2011.