Fast and Optimal Scheduling Over Multiple Network Interfaces

CS Technical Report 2007-36

Matei A. Zaharia
Computer Science Division, Dept. of EECS
University of California, Berkeley
matei@berkeley.edu

Srinivasan Keshav
David R. Cheriton School of Computer Science
University of Waterloo
keshav@cs.uwaterloo.ca

ABSTRACT

Today's mobile phones already contain two or more network interfaces (NICs) and future devices are likely to have several more, each with different energy costs, dollar costs, and data transmission capacities [23]. Given a mobile device that is running multiple data-oriented applications, such as email, instant messenger, and video download, some of which may be delay-tolerant, it becomes necessary to assign, to each unit of application data, the network interface and the time of transmission that maximizes user satisfaction. This scheduling algorithm must take into account not only user, application, and NIC characteristics, but also future opportunistic availability of NICs due to device mobility.

In this paper, we begin by showing that naive scheduling approaches can be far from optimal. We then show that although optimal schedules can be found using either linear programming or network flow, these classical approaches are infeasible in CPU- and memory-constrained mobile devices. This motivates our novel hill-climbing approach that exploits problem structure to optimally schedule application messages over multiple NICs assuming perfect knowledge of future connectivity. We prove mathematically that our algorithm is correct.

Detailed performance measurements show that our algorithm finds optimal solutions 10-100 times faster than the simplex method and network flow. We also describe an implementation of our solution in a J2ME-based scheduler that runs on laptops, smartphones and PDAs, demonstrating its effectiveness in a realistic setting.

1. INTRODUCTION

Today's mobile phones already contain two or more network interface cards (NICs), and future devices are likely to have several more, each with different energy costs, dollar costs, and data transmission capacities [23]. Different NICs may differ in these attributes by several orders of magnitude; for instance, a WiFi NIC operating at 54 Mbps is nearly four orders of magnitude faster than a GPRS NIC at 8 Kbps, and the cost per bit for WiFi can be four orders of magnitude lower than the cost per bit for GPRS. Moreover, such mobile devices are not single-purpose voice terminals, as was common in the past, but run multiple applications including email, interactive text messaging, and video download. Given that each application generates and consumes data with different tolerance for delays, and given that the device user may wish to optimize one of several objective functions, such as minimizing dollar cost, maximizing battery life, or maximizing performance, a *scheduling algorithm* assigns each unit of application data to the right network interface, and at the right time, to maximize user satisfaction. This scheduling problem is complex because it must take into account not only user, application, and NIC characteristics, but also future availability of NICs due to mobility.

For example, consider a user who is currently only served by an expensive, low-bandwidth cellular data provider (*a NIC characteristic*), but who is willing to delay sending a non-urgent email message for up to an hour (*an application characteristic*) if this will save money (*a user requirement*). This user requirement can be met by exploiting the user's mobility pattern. Specifically, the email could be sent using a much cheaper 802.11 hotspot that comes into range within the next hour. One can easily imagine more complicated scenarios, where competing requirements of interactive and bulk data applications need to be satisfied while dealing with intermittent connectivity on each of multiple interfaces.

Today, to accomplish even the relatively straightforward requirement in our example, the user must manage connectivity *manually*, by choosing when to run specific applications, and on which NIC, while simulta-

neously keeping track of NIC availability and predicting mobility. "Smart" devices are anything but: the smarts must come from a human. With a proliferation of applications, networks, and communication opportunities, this is neither convenient nor practicable, and the problem is only likely to get worse over time.

We therefore present an algorithm and system to automatically and non-intrusively schedule transmissions over multiple, intermittently available network interfaces. Our overall solution has two complementary aspects. First, we build a simple mathematical model and use it to obtain a provably correct and optimal scheduling algorithm. Second, we implement this algorithm on two mobile devices, and evaluate our solution in some plausible mobility scenarios.

In the remainder of this section, we first introduce goals for scheduling systems and describe the context of our implementation. We then formalize the scheduling problem mathematically, and present two features of problem structure that let us solve it efficiently. Finally, we provide a roadmap for the rest of the paper.

1.1 Goals

We believe that an ideal scheduler for mobile devices should meet the following goals:

1. **Autonomic operation:** A user should only need to specify objectives, such as "I want to send this email urgently" or "I want to download this movie by tonight." The scheduler should then schedule data transfer appropriately to accomplish the user's goals at the least dollar and energy cost. The scheduling decision must take into account widely varying application requirements as well as the expected mobility pattern of the user, so that it can exploit future connection opportunities to send delay-tolerant messages. Moreover, message importance, message urgency, and NIC costs must be take into account in scheduling transmissions.

2. **Optimality:** The scheduler should maximize the user's utility. For example, messages should not be delayed unnecessarily. Similarly, when multiple NICs are available, they should be used in parallel, as long as this does not incur an excessive cost.

3. **Suitable for mobile devices:** The algorithm should be simple enough to be implementable on a CPU- and memory-constrained mobile device. In practical terms, we would like a solution to the scheduling problem to not take more than a second of CPU time even on a CPU-constrained mobile device, so that we can efficiently recompute schedules when connection opportunities arise.

In this paper, we present an algorithm and system that substantially meets these goals. We compute a provably optimal schedule *assuming future connectivity on each NIC is precisely known*, and *assuming that each NIC is associated with a constant service rate.* Even in this restricted setting, we demonstrate that naive greedy algorithms can have as little as half the performance of the optimal solution. Moreover, we show that classical optimization techniques are too resource-intensive to run on mobile devices. Instead, our algorithm exploits problem structure to find optimal schedules 10-100 times faster than classical techniques. Our solution can be easily extended to allow re-scheduling when new user data, or new connection opportunities, arise. It also provides insights into the form of the general solution.

Note that in the subsequent discussion, we only discuss the scheduling problem in the device-to-proxy direction. A symmetric problem exists in the reverse direction as well. While we do not explicitly present a solution here due to limitations of space, our approach is to use an always-on cellphone-NIC-based *control channel* between the device and proxy to let the device tell the proxy which interfaces it has (or expects to have) available, its utility functions and its NIC costs. This allows the proxy to compute an optimal transmission schedule, which it informs the device about, also on the control channel. The device then uses this schedule to block off reception slots on each NIC, treating them as if they were NIC downtimes from the perspective of uplink scheduling.

1.2 Limitations

A word about our assumptions. Although our assumption of known future connectivity is strong, we believe that it is acceptable, for the following reasons. First, previous research has shown that mobile users have surprisingly predictable schedules [17], implying that past history and current location can lead to good predictions. Second, even when there is uncertainty about future connectivity, our algorithms can still make useful decisions based on a pessimistic NIC-availability schedule. Third, it is easy to detect when a user (or NIC) is not following the predicted schedule, and this can be used to trigger rescheduling of messages according to an updated future outlook. Fourth, as we demonstrate, even scheduling based on known future connectivity is challenging, due to the limited computational resources of mobile devices. Finally, our work provides insights into how to schedule with uncertainty using stochastic optimization.

Our assumption that each NIC has a fixed service rate is also plausible. To begin with, wireless WAN NICs are likely to be the bottleneck in any typical end-to-end transport path. For instance, an EDGE NIC has a data rate of only about 25 kbps in practice, which is smaller than the capacity of a typical end-to-end Inter-

net paths[1]. Therefore, when the EDGE NIC is available, we can assume that the device gets a constant service rate of 25 kbps. This assumption is less defensible for WLAN (802.11) NICs. First, the capacity of an 802.11 NIC depends on the signal to noise level, and, due to auto-rate fallback, the capacity of the link may vary from 6 Mbps to 54 Mbps. Second, typical 802.11 rates are much higher than the capacity of typical end-to-end Internet paths. Hence, the actual rate seen by a device will depend on the degree of contention on the wired backbone, which can vary widely. Third, a single 802.11 NIC may be in range of multiple access points, which may each provide a different service rate. Finally, the 802.11 DCF algorithm shares capacity fairly, and at a fine-grained time scale, with other contending devices. This changes the capacity of the NIC in an unpredictable manner.

Nevertheless, we feel that a constant rate assumption is a necessary first step in modeling this complex problem. Besides, we can model each 802.11 AP as a separate virtual NIC, which can reduce the variability associate with different APs [1]. Finally, as we discuss in the next subsection, in our prototype, and indeed in most practical wireless systems today, all communication from a mobile device to the wired network goes through a *proxy* that isolates device disconnection and mobility from legacy servers [24]. Therefore, the device need only periodically estimate the service capacity on a path to this proxy from each NIC. Given that we expect this path have substantially less variability than a generic Internet path, the constant rate assumption seems reasonable even for 802.11 NICs.

1.3 Implementation Context

Our scheduling algorithm is suitable for all multi-NIC mobile devices. However, for the purpose of evaluating a real system, we chose to implement our algorithm in the context of the Opportunistic Communication Management Protocol (OCMP) implementation [15]. Besides being freely available and open source, this system provides an API to let applications take advantage of multiple interfaces in accordance with user policies. Moreover, it allows arbitrary scheduling policies to be 'plugged in' to the system. To inter-operate with legacy applications, which cannot deal with multiple, potentially disconnected paths from a single mobile, OCMP supports a proxy that lies on every data path, hiding device disconnections from legacy applications. The publicly available version of the system has a simplistic scheduler that assumes that all applications have the same utility function. We extended OCMP to address a more general usage environment. Additional details of our implementation are in Section 8.

[1]This also allows us to model the fact that different hotspot providers may have different pricing schemes.

Note that in the subsequent discussion, we only discuss the scheduling problem in the device-to-proxy direction. A symmetric problem exists in the reverse direction as well. While we do not explicitly present a solution here due to limitations of space, our approach is to use OCMP's always-on cellphone-NIC-based *control channel* between the device and proxy to let the device tell the proxy which interfaces it has (or expects to have) available, its utility functions and its NIC costs. This allows the proxy to compute an optimal transmission schedule, which it informs the device about, also on the control channel. The device then uses this schedule to block off reception slots on each NIC, treating them as if they were NIC downtimes from the perspective of uplink scheduling.

1.4 Mathematical Model

We now model the scheduling problem to make it analytically tractable. We assume that all messages are fragmented into fixed-size *bundles*, and split the periods of predicted uptime of each NIC into *time slots*, each of duration equal to the time it takes to transmit a bundle over that interface. We categorize messages into several *service classes*, each with a *utility function* that decreases over time. This function represents user requirements in the form of "time value of data", that is, how much more beneficial it is to send a message sooner rather than later. This notion of a time/utility function (TUF) was introduced by Jensen in real-time scheduling [6] and there exists a substantial body of literature dealing with how to choose these time/utility functions for practical situations such as [7, 13, 10]. Finally, we assign *costs* for sending a bundle of each class on each interface. These costs represent a weighted combination of the dollar cost and energy cost[2] to send a bundle on each NIC.

The scheduling problem at any moment in time then reduces to mapping a set of bundles to future time slots so as to maximize total utility. The utility of a schedule is the sum of the utilities of each bundle at the time it was sent, minus the sum of the costs incurred. Note that total utility is independent of the order of transmission of bundles of the same class; in practice, they should be sent in FIFO order.

We observe that our model assigns utility to each bundle, not each application-level message. This modeling allows us to send portions of a message at different times and to stripe bundles from the same message over multiple NICs. We realize that in some cases users may gain no utility from a fraction of a message. Nevertheless:

- Applications where complete messages must be sent

[2]We realize that, due to power control, the energy cost to send a bundle on a NIC may vary with distance from the base station. However, for tractability, we ignore this detail.

often have messages that fit into one bundle (∼8KB). This includes most email and instant messages.

- Some data formats are progressive, so having even the first few bundles provides utility. For example, JPEG and GIF images often begin with data that allows a low-quality version of the image to be displayed while the rest is loaded.
- Scheduling arbitrary-length messages over even a single interface is equivalent to Karp's MAX-JOB-SEQUENCING problem, which is NP-hard [8]. So, this assumption makes the problem tractable.

We use the following notation:

Symbol	**Meaning**
K	Number of service classes.
L	Number of NICs.
N	Number of bundles.
N_i	Number of bundles of class i.
T	Number of time slots.
T_j	Number of time slots on NIC j.
$t_{j,k}$	Time of k'th slot on NIC j.
$u_i(t)$	Utility of a bundle of class i at time t.
$c_{i,j}$	Cost of a bundle of class i on NIC j.
$U(\mathcal{S})$	Utility of schedule $\mathcal{S}$.
$\text{NB}(\mathcal{S}, i)$	Number of bundles of class i sent by $\mathcal{S}$.
$\text{NS}(\mathcal{S}, j)$	Number of bundles sent by $\mathcal{S}$ on NIC j.

1.5 Exploiting Problem Structure

We now make two observations about problem structure. First, in practice, it is almost always true that interface costs are *class-independent*, that is, $c_{i,j}$ is some value C_j dependent only on j. This is a reasonable assumption because energy and dollar cost of bundles ought to depend only on the size of the bundles.

Second, we assume that the u_i's are decreasing, with rates of decrease that are consistently *ordered*. This means that the relative urgency of each class remains constant during the scheduling problem, so bundles that are losing utility quickly "now" will continue to lose utility quickly in the future. Formally, if we number the classes $1, 2, ..., K$ in order of urgency, then at any time t, we require $u'_1(t) \leq u'_2(t) \leq \cdots \leq u'_K(t) \leq 0$. This restriction still allows for a wide variety of utility functions. In particular, it allows for the intuitively appealing *linear* utility functions $u_i(t) = a_i - b_i t$, where a_i corresponds to the importance and b_i to the urgency of class i.

We observe that given arbitrary utility functions, we can always split time into discrete time periods such that in each time period, the ordering restriction applies. This allows us to generalize our solution to nearly arbitrary monotone non-increasing utility functions. In this work, we do not elaborate further on this conceptually straightforward generalization.

1.6 Roadmap

We begin by surveying related work in Section 2. We then show in Section 3 that simple greedy approaches to the scheduling problem can be far from optimal. In Section 4, we describe how optimal schedules can be found using linear programming or network flow, but show that these classical approaches are infeasible on CPU- and memory-constrained mobile devices. We then introduce our hill-climbing algorithm in Section 5, and prove that it is correct. Although our algorithm is static, we explain how it can be adapted to incrementally update schedules as new bundles become available, reducing the work per new bundle, in Section 7. Section 8 describes an implementation of our solution in J2ME and in Section 9 we evaluate its performance. Section 10 concludes the paper.

2. RELATED WORK

To our knowledge, the importance of multi-NIC devices was first articulated by Bahl et al [3]. However, in the literature, the presence of multiple radios has been exploited mainly for reducing overall power usage, such as by using the Wake-on-Wireless technique [16], with a few exceptions, as noted next.

Our use of multiple wireless interfaces, potentially in parallel, is similar in spirit to pTCP [5]. Unlike pTCP, which assumes an underlying TCP connection, we use OCMP, which provides seamless connectivity at the session layer and can operate over arbitrary transport layers. Moreover, unlike pTCP, we have built, deployed and evaluated the performance of the system in a testbed, instead of relying only on simulations.

Policy-based selection of network interfaces was first introduced in [21], and has since been extensively explored in the context of *vertical handoffs* [22, 12]. The problem was motivated by the desire to choose the network that optimizes metrics such as data rates or power consumption in the long term, while preserving seamless connectivity. This formulation, however, assumes that only one NIC is used at any time, and does not attempt to maximize user utility by exploiting a user's tolerance of delays for certain applications.

Optimal and opportunistic scheduling over multiple channels has been well-studied in the context of cellular networks (see [2] for a survey). This body of work studies the problem of assigning messages to time-varying cellular channels to maximize performance, while simultaneously providing long-term fairness among competing mobile devices. This approach differs from ours in three ways. First, we study the scheduling problem at a device, not at the base station. Second, we do not assume that all applications have the same utility function, such as throughput maximization. Third, our scheduler operates on the time scale of minutes to hours, instead of at the time scale of milliseconds to seconds,

as in the cellular network. This makes our work necessarily more complex.

Intelligent selection of network interfaces with session persistence is also being explored in the Haggle project [14]. However, Haggle is focused on infrastructure-less systems where devices communicate with each other in an ad-hoc manner. Further, the scheduling decisions made by Haggle's Resource Manager do not take into account future connectivity patterns, as we do.

Scheduling to maximize a sum of time/utility functions (TUFs) for a set of tasks has also been studied in the context of realtime embedded systems [13]. Typically such systems must schedule processes of different lengths of time using shared resources (CPU's and potentially other devices). Our problem setting differs in three ways. First, we have two orders of magnitude more items to schedule - we are scheduling 1000-10000 bundles instead of 10-100 processes. Therefore, we require an algorithm that is nearly linear-time, while the majority of realtime scheduling algorithms are asymptotically slower. Second, time slots on network interfaces differ from time on a shared processor because using each slot has an associated cost. This fundamental aspect of the problem is what makes it possible to gain utility by delaying transmission of a message until a lower-cost interface becomes available. Realtime scheduling does not model processing costs or the eventual availability of low-cost processors. Third, we model NIC unavailability, which is not considered realtime scheduling models. For example, we show that the greedy TUF-driven Ethernet scheduling algorithm in [20], which is optimal for linear utility functions if all packets are feasible, can fail to be optimal by an arbitrarily large factor if the NIC becomes unavailable.

Finally, the use of linear programming and network flow for optimization is well-known. Our innovation here is in observing that the NP-hard multi-NIC scheduling ILP can be reduced to an LP by exploiting "total unimodularity" [11].

3. GREEDY IS SUB-OPTIMAL

At first glance, it might appear that a greedy algorithm can solve the scheduling problem adequately. However, such algorithms can perform as poorly as a factor of two worse than the optimal algorithm even with only a *single* NIC and even when there are enough time slots to send all bundles. We show this for three different greedy algorithms. Note that these greedy algorithms do not take into account the fact that NICs may become unavailable: the optimal schedule is better because it takes this into account.

3.1 Highest Utility First (HUF)

Given a set of bundles with differing utilities, a natural greedy algorithm is to send the bundle with highest utility first, with a secondary rule for breaking ties. We now show that the utility gained from HUF can be as little as half that of an optimal schedule. Consider two bundles on a single NIC, where each bundle takes unit transmission time. Let their utility functions $u1$ and $u2$ respectively be as shown in Figure 1(a)–we define the utility of a time slot as the value of the time-utility function at the *start* of the time slot. With HUF, the order of transmission is 2,1, so that the total utility of the schedule is I because the utility of the first bundle declines to zero by the time it is scheduled. The optimal algorithm sends the bundles in the order 1,2, to get a total utility of $I - \epsilon \ + \ I - \epsilon$, which tends to $2I$ as $\epsilon \to 0$. Thus, HUF asymptotically performs half as well as optimal. The reason is that the Highest Utility First algorithm does not take into account how bundle utilities decrease over time and fails to send bundle 1 first, even though it could be sent ahead of the more delay-tolerant bundle 2. By noting that at each time step HUF can erroneously send at most one bundle earlier than necessary, it can be seen that HUF will always perform at least half as well as the optimal.

3.2 Earliest Deadline First (EDF)

A natural way to take delay-tolerance into account is to send the bundles in order of their deadlines (the time when they reach 0 utility). Again, this fails to be optimal even over a single NIC. To see this, consider Figure 1(b), where the shaded area represents a time period when the NIC is unavailable. EDF would send bundle 1 at time 0, and gain utility J. The second bundle cannot be sent, leading to a total utility of J. In contrast, the optimal schedule is to send bundle 2 and gain utility I. The relative performance of EDF is J/I, which can be made arbitrarily small. The reason is that the Earliest Deadline First algorithm does not take into account the fact that bundles with early deadlines with small utilities may force out bundles with higher utilities but later deadlines.

3.3 Most Urgent First

These observations motivate a third algorithm: send bundles in order of rate of decrease of their utility functions (urgency), with rules to drop bundles that cannot be sent before reaching 0 utility, and to choose the highest-utility bundles when fewer slots than bundles are available. This is the essential idea behind the TUF-driven single-NIC Ethernet scheduling algorithm in [20], which sorts the packets to be sent by urgency and performs local modifications on the sorted list to remove infeasible packets and reorder the others to increase utility. Although this algorithm is optimal for linear utility functions when the NIC is continuously available, it can be arbitrarily worse than the optimal algorithm when interfaces can become unavailable. For

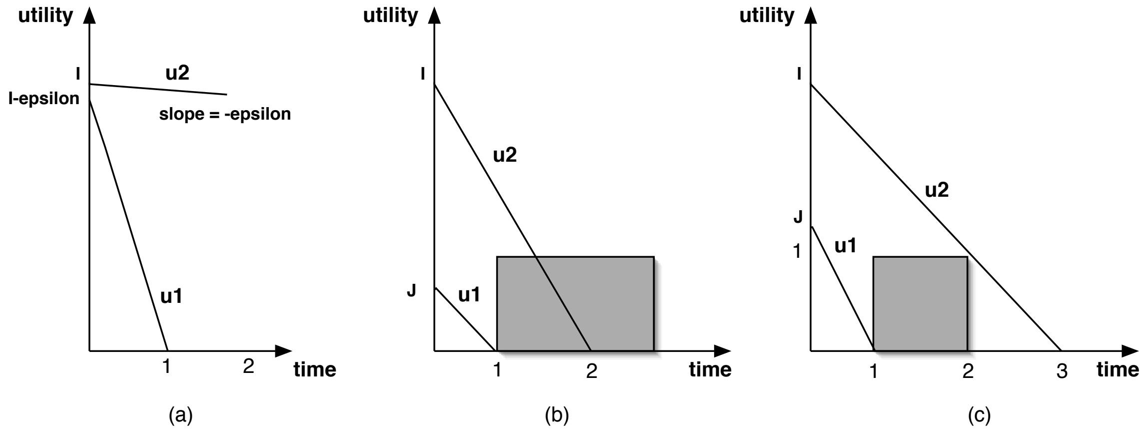

Figure 1: Utility functions for examples showing why greedy approaches can perform poorly.

example, consider the two bundles in Figure 1(c). Both bundles can be sent before their deadline, so the MUF algorithm will prune neither one. Bundle 1 is more urgent than bundle 2, so MUF sends it first. When the NIC becomes available again, bundle 2's utility has declined to 1, so the total utility from the MUF schedule is $J+1$. In contrast, the optimal schedule would drop bundle 1, and only send bundle 2, for a utility of I. The ratio of the two utilities is $(J+1)/I$, which can be made arbitrarily small (note that we can make the NIC downtime as long as needed to ensure that not matter how large I is, by the time bundle 2 is sent, its utility has declined to 1).

We have just shown that all three greedy algorithms can perform poorly, as compared to an optimal algorithm, even with just two bundles and a single NIC. Two of the greedy algorithms perform arbitrarily worse, and one performs as much as two times worse, compared to an optimal scheduler.

If there are more bundles than available time slots and more than one NIC, the problem becomes more complicated. Heuristics such as "drop bundles of the least important class" fail to work - there are situations in which an optimal algorithm would both send and drop bundles from every class. For example, consider the three utility functions $u_1(t) = 20 - 2t$, $u_2(t) = 23 - 4t$, and $u_3(t) = 24 - 6t$. At time $t = 0$, u_3 is highest. At $t = 1$, u_2 is highest. And at $t = 2$, u_1 is highest. Therefore, if we had three time slots, at times 0, 1, and 2, and two bundles of each class, we would have to send one bundle and drop one bundle from each class. No simple rule tells us which bundles to send and which to drop.

Multiple NICs complicate the problem even further. How should a greedy algorithm decide when to postpone a bundle transmission and send it later over a lower-cost NIC, and which bundles should be thus postponed? In general, we may want to use some number of slots on each NIC, depending on when these slots are available and what the costs of the NICs are. This motivates the need for a more sophisticated scheduling algorithm, as described next.

4. CLASSICAL OPTIMIZATION

Given that greedy approaches have problems, one solution is to look for an optimal solution using classical optimization. We show first that linear programming does, indeed, find the optimal solution. Then, we demonstrate that, unfortunately, that LP solvers cannot be used in current mobile devices with limited memory and CPU resources. This motivates our work on fast and small-footprint optimal scheduling algorithms.

4.1 Linear Programming

We can always optimally assign a service class to each time slot with the followng "brute force" integer program, P:

$$
\begin{aligned}
\text{Maximize: } & \sum_{i=1}^{K}\sum_{j=1}^{L}\sum_{k=1}^{T_j}(u_i(t_{j,k}) - c_{i,j})x_{i,j,k} & \\
\text{Subject to: } & \sum_{i=1}^{K} x_{i,j,k} \leq 1 & \forall j,k \\
& \sum_{j=1}^{L}\sum_{k=1}^{T_j} x_{i,j,k} \leq N_i & \forall i \\
& x_{i,j,k} \geq 0 & \forall i,j,k \\
& x_{i,j,k} \text{ integer} & \forall i,j,k \\
& x_{i,j,k} = 0 \;\; \forall \text{k s.t. NIC j is unavailable} &
\end{aligned}
$$

The variable $x_{i,j,k}$ represents whether to send a bun-

dle of class i in time slot k on interface j. Each $x_{i,j,k}$ can only be 0 or 1, from the second and third constraints. The first constraint ensures that no more than one bundle is sent during each slot, and the second constraint ensures that no more than N_i bundles of class i are sent. The last constraint prevents transmission on an unavailable NIC.

We are now in a position to state our first main result: although integer programming is NP-complete, P can be solved using *linear programming*, by the following result:

THEOREM 1. *The vertices of the feasible region of P all have integer coordinates.*

PROOF. Notice that this feasible region is of the form "$x \geq 0$ and $Ax \leq b$" where x is a vector containing the variables $x_{i,j,k}$ and $Ax \leq b$ represents the constraints $\sum_{i=1}^{K} x_{i,j,k} \leq 1$ and $\sum_{j=1}^{L} \sum_{k=1}^{T_j} x_{i,j,k} \leq N_i$. The matrix A contains only 0's and 1's, and each column of A (which corresponds to the coefficients for some particular variable $x_{i,j,t}$ in the inequalities) contains exactly two 1's (one in $\sum_{i=1}^{K} x_{i,j,k} \leq 1$ and one in $\sum_{j=1}^{L} \sum_{k=1}^{T_j} x_{i,j,k} \leq N_i$). Furthermore, the rows of A can be partitioned into two sets such that each column has exactly one 1 in each set (separate the $\sum_{i=1}^{K} x_{i,j,k} \leq 1$ rows from the $\sum_{j=1}^{L} \sum_{k=1}^{T_j} x_{i,j,k} \leq N_i$ rows). Finally, the vector b contains only integers. Thus, by Theorem 13.2 in [11], the matrix A is *totally unimodular* and the polytope defined by $Ax \leq b$ and $x \geq 0$ has vertices only at integer coordinates. □

Therefore, we can solve P using any linear programming algorithm that returns a vertex of the feasible region, such as the simplex method [11], to obtain an optimal schedule.

4.2 Network Flow

Scheduling is a matching problem, where bundles must be matched to time slots, so it can also be solved using min-cost flow. Unlike linear programming, a flow formulation lets us specify that bundles from the same class are equivalent, and therefore solves the problem more efficiently.

Scheduling can be formulated as a min-cost flow problem as shown in Figure 2. This graph has:

- A source vertex, *Src* and a sink, *Snk*.
- A vertex V_i for each class i.
- A vertex $W_{j,k}$ for each time slot k under consideration[3] on each interface j.

and the following edges:

[3] If there are a total of N bundles, we need to schedule at most N timeslots on each interface.

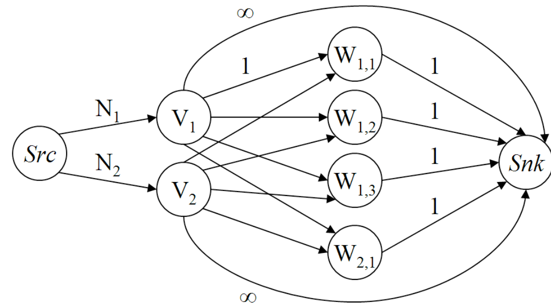

Figure 2: Flow network for scheduling.

- An edge with capacity N_i and cost 0 from *Src* to V_i.
- An edge with capacity 1 and cost 0 from $W_{j,k}$ to *Snk*.
- An edge with capacity 1 and cost $-(u_i(t_{j,k}) - c_{i,j})$ from each V_i to each $W_{j,k}$.
- An edge with capacity ∞ and cost 0 from each V_i to *Snk* (for dropping bundles).

Figure 2 shows an example flow network two classes and 4 time slots. Edges are labeled with their capacities.

Taking the edge $V_i \rightarrow W_{j,k}$ represents assigning a bundle of class i to time slot k on interface j and contributes minus the utility of this assignment to the total cost. Taking an edge $V_i \rightarrow Snk$ corresponds to dropping a bundle of class i. The edge capacities ensure that no two bundles are assigned to the same time slot and no more than N_i bundles of class i are sent. Thus a min-cost flow on this network corresponds to a maximum-utility schedule.

4.3 Performance Evaluation

We tested the LP and flow approaches on a high-performance SGI Altix 3700 computation server with 64 1.4 GHz Intel Itanium2 CPU's, using the state-of-the-art CPLEX linear programming package [25] and CS2 network flow solver [4]. We measured their running times on randomly generated problem instances of various sizes.

Here, we present average running times for 5000-bundle problems with various numbers of service classes and NICs. More detailed results are deferred to Section 9.1: the point we want to make here is that even on powerful servers, classical approaches are unacceptably slow for typical problem sizes. Note that our problem size is realistic: For example, if bundles are 4 KB in size, then 5000 bundles correspond to 20 MB of data, the size of approximately 8 songs, 5 minutes of video, or 50 photographs.

Problem Size	CPLEX (s)	CS2 (s)
$K=5, L=2, N=5000$	1.39	0.42
$K=5, L=5, N=5000$	3.08	1.13
$K=10, L=2, N=5000$	1.94	0.50
$K=10, L=5, N=5000$	4.72	1.19

Both algorithms require at least 400 ms to compute a schedule, even using state-of-the-art software on a high-end machine. This is too slow to enable a mobile device to reschedule bundles in real time when a new message is submitted to the scheduler. Furthermore, implementing these algorithms on a CPU- and memory-limited device, such as a cell phone, would result in significantly worse performance.

Intuitively, the problem is that the classical optimization algorithms must work with a large number of variables: KLT variables $x_{i,j,k}$ in a linear program, or KL edges in a flow network. Our algorithm, presented next, exploits problem structure to drastically reduce the number of variables to consider.

5. HILL-CLIMBING APPROACH

We obtain a more efficient algorithm by restricting our attention to a smaller class of schedules that we call *simple schedules*. Our algorithm searches through the space of simple schedules using hill-climbing.

We present our algorithm in four parts. First, we describe simple schedules, in Section 5.1. An important characteristic of simple schedules is that their utility can be evaluated efficiently, as described in Section 5.2. We present the hill-climbing algorithm itself in Section 5.3. Finally, prove its correctness in Section 5.4.

5.1 Simple Schedules

Simple schedules are defined based on two observations about what characterizes a "good" schedule.

Our first observation is that *it is never beneficial to leave a time slot empty on some interface, then use a later time slot on the same NIC*. This is because utilities never increase over time, so any schedule that leaves an early slot empty and sends some bundle b during a later slot can be improved by moving b to the earlier slot.

Recall that we made two assumptions about the problem parameters in Section 1.5, based on the problem structure: bundle transmission costs are class-independent, and rates of decrease of utility functions are consistently ordered (relative urgencies of bundles remain constant). These assumptions lead to our second observation: *it is never beneficial to send a less urgent bundle before a more urgent bundle*. This might seem counterintuitive at first: it may appear that total utility of each bundle, not just its rate of decrease, should be a factor in selecting which to send first. However, recall that we seek to optimize the *total* utility of the schedule, not just the utility of the next action. Informally, by delaying a more urgent bundle, we are losing more utility than by delaying a less urgent bundle, so more urgent bundles should always be sent first.

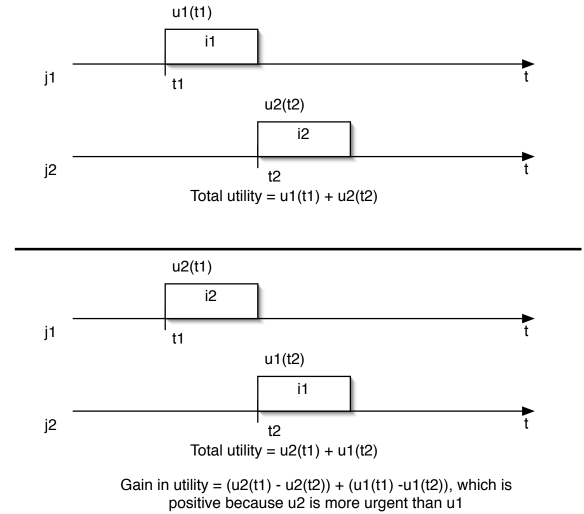

Figure 3: Simplifying a schedule - sending a more urgent bundle earlier is always beneficial.

To illustrate this point formally, suppose that in some schedule $\mathcal{S}$, there was a bundle of class i_1 being sent on interface j_1 at time t_1, and a bundle of class $i_2 < i_1$ (a class more urgent than i_1) being sent on interface j_2 at time $t_2 > t_1$ (Figure 3). Then one can improve total utility by swapping these two bundles: Send the bundle of class i_2 earlier, over interface j_1 at time t_1, and send the bundle of class i_1 later, over interface j_2 at time t_2. Since the utility functions are ordered so $u'_{i_1}(t) \leq u'_{i_2}(t)$ at every time t, the bundle of class i_1 loses $u_1(t_2) - u_1(t_1)$ utility when moved from time t_1 to t_2, but the bundle of class i_2 gains $u_2(t_1) - u_2(t_1)$ utility by being sent earlier, which is a larger quantity than the utility lost. (Transmission costs are the same in either case, because NIC costs are class-independent.)

DEFINITION 1. *A schedule $\mathcal{S}$ is* simple *if the following conditions are satisfied:*

- *For each interface j and each slot number k, if $\mathcal{S}$ transmits a bundle in slot k on interface j, then it transmits bundles in slots 1 through $k-1$ on interface j as well.*
- *For any two classes $i_1 < i_2$, all bundles of class i_1 are transmitted before any bundle of class i_2.*

That is, there are no unused slots followed by non-empty slots on each NIC, and the bundles are transmitted in order of urgency.

5.1.1 Uniqueness of Simple Schedules

We observe that every simple schedule $\mathcal{S}$ is specified uniquely by two sets of values:

- The number of bundles it sends of each class i, $\text{NB}(\mathcal{S}, i)$.
- The number of slots it uses on each NIC j, $\text{NS}(\mathcal{S}, j)$.

If these sets of values are given, then we can construct the schedule by taking the first $\text{NS}(\mathcal{S}, j)$ time slots from each interface j, sorting all of them in order of time, then assigning the bundles of class 1 to the first $\text{NB}(\mathcal{S}, 1)$ slots, the bundles of class 2 to the next $\text{NB}(\mathcal{S}, 2)$ slots, etc. This gives a unique schedule with these values of NB and NS that ensures that no slot is skipped and no bundle is sent before a more urgent bundle, so this schedule must be $\mathcal{S}$.[4]

Consequently, we will represent every simple schedule $\mathcal{S}$ as two vectors of integers, $\text{NB}(\mathcal{S}, i)|_{i=1}^{K}$ and $\text{NS}(\mathcal{S}, j)|_{j=1}^{L}$.

5.1.2 Simplification

Any non-simple schedule can be converted to a simple schedule by reordering its bundles. We call this procedure *simplification*. As we shall see, the simplification of a schedule always has utility at least as high as the original schedule. An immediate consequence of this is that there exists an optimal schedule which is simple. These results let us design an algorithm which *operates only on simple schedules, but finds a globally optimal schedule*. We first present the mathematics of simplification. We conclude this subsection with a discussion about the significance of simple schedules.

DEFINITION 2. *The* simplification *of a schedule $\mathcal{S}$ is the unique simple schedule $\sigma(\mathcal{S})$ where $\text{NB}(\sigma(\mathcal{S}), i) = \text{NB}(\mathcal{S}, i)$ for all i and $\text{NS}(\sigma(\mathcal{S}), j) = \text{NS}(\mathcal{S}, j)$ for all j.*

THEOREM 2. *For any schedule $\mathcal{S}$, $U(\sigma(\mathcal{S})) \geq U(\mathcal{S})$.*

PROOF. $\mathcal{S}$ can be converted to a simple schedule while keeping $\text{NB}(\mathcal{S}, i)$ and $\text{NS}(\mathcal{S}, j)$ constant and never decreasing $U(\mathcal{S})$ through the following process:

1. While there is an unused slot followed by a used slot on some interface j, move the last bundle sent on j to the first unused slot.

2. While there are bundles of classes i_1 and i_2 with $i_1 < i_2$ on some interfaces j_1, j_2 such that a bundle of class i_2 is being sent earlier than one of class i_1, swap their positions in the schedule.

[4]This procedure may be ambiguous when two slots on different interfaces occur at the same time. In practice, using a well-known optimization technique, we perturb the slot times in the inputs by very small random values without changing the form of the solution, to ensure that this never happens [11].

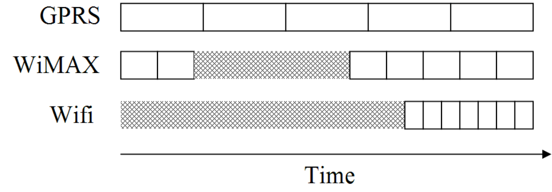

Figure 4: Figure illustrating the complexity of simple schedules

The first step terminates because it moves each bundle on an interface at most once (when it is the last bundle sent on that interface). The second step terminates because it reduces the number of "inversions" (bundles being sent in the wrong order of urgency) at each step, and this number is finite to begin with and can never be below 0. The resulting schedule is simple (since the two steps can no longer be performed), and $\text{NB}(\mathcal{S}, i)$ and $\text{NS}(\mathcal{S}, j)$ have been kept constant, so it is $\sigma(\mathcal{S})$. Finally, neither of these steps reduces utility, as argued in the observations earlier, so $U(\sigma(\mathcal{S})) \geq U(\mathcal{S})$. $\square$

COROLLARY 1. *There exists an optimal schedule which is simple.*

PROOF. Let $\mathcal{O}$ be any optimal schedule. Then $\sigma(\mathcal{O})$ is simple and $U(\sigma(\mathcal{O})) \geq U(\mathcal{O})$, so $\sigma(\mathcal{O})$ is optimal. $\square$

5.1.3 Discussion

The results in this section mean that, to choose an optimal schedule, it suffices to select $K + L$ numbers: the number of bundles to send from each class, $\text{NB}(\mathcal{S}, i)$, and the number of slots to use on each interface, $\text{NS}(\mathcal{S}, j)$. Subsequently, the optimal order for the bundles is predetermined - it must be the unique simple schedule with these values of NS and NB.

At first, this might appear counterintuitive, because selecting numbers $\text{NB}(\mathcal{S}, i)$ and $\text{NS}(\mathcal{S}, j)$ seems to be a relatively simple problem. However, two factors make selection of these numbers difficult:

First, because each interface has an uptime schedule that may include gaps when it is unavailable, choosing $\text{NS}(\mathcal{S}, j)$ is equivalent to deciding how many of the intervals of connectivity on interface j to employ. The fact that the schedule is simple indicates only that the *earliest* intervals should be used. For example, consider the NIC availability schedule shown in Figure 4, where each white rectangle represents a time slot:

In this schedule, a slow GPRS interface (with long time slots) is always available. The user begins in a WiMAX zone, but then moves out of it. She then enters a second WiMAX zone, and eventually a Wifi hotspot. In this case, selecting $\text{NS}(\mathcal{S}, 1)$ is equivalent to choosing whether, and how much, to use GPRS. For example, choosing $\text{NS}(\mathcal{S}, 1) = 0$ means that GPRS will

not be used. Selecting $\mathrm{NS}(\mathcal{S},3)$ is equivalent to choosing whether to take advantage of the Wifi hotspot: if bundles must be sent before it, then this number will be 0. Finally, selecting $\mathrm{NS}(\mathcal{S},2) > 2$ means that some bundles will be delayed until the second WiMAX zone.

An optimal schedule may choose $\mathrm{NS}(\mathcal{S},1) = 1$ (to send an urgent message), $\mathrm{NS}(\mathcal{S},2) = 1$ (to send a few messages earlier on WiMAX), and then $\mathrm{NS}(\mathcal{S},3) = 7$ (to ignore the second WiMAX zone at the cost of a small delay, and take advantage of the Wifi hotspot). Choosing $\mathrm{NS}(\mathcal{S},j)$ is equivalent to making these decisions.

Second, selecting $\mathrm{NB}(\mathcal{S},i)$ is equivalent to choosing which messages to drop. Although a simple schedule dictates which *order* to send bundles in, deciding *which* bundles to send is also important. For example, if the user is sending an urgent email and streaming video for a chat with a friend, then as long as there is time to perform both tasks, video packets should take priority, because delaying the email by a few seconds is not as harmful as degrading video quality. However, if there is only enough bandwidth to send the email, then the video should be suspended.

In summary, the fact that a schedule is simple is useful only after we have chosen NS and NB. The difficult decisions a scheduler must make are equivalent to choosing these numbers.

5.2 Finding the Utility of a Simple Schedule

An important advantage of simple schedules is that their utilities can be computed efficiently. We can compute utilities in $O(KL\log^2 T)$ time, with a one-time $O(KLT)$ precalculation step (or just $O(LT)$ for linear utility functions). This enables a hill-climbing algorithm to evaluate schedules efficiently.

Recall that we are representing simple schedules as two vectors, NB and NS. We compute utility of a simple schedule $\mathcal{S}$ from these vectors in two steps:

1. Determine how many bundles of each class i are sent over each interface j. We call this $n[i,j]$.
2. Compute the utility using the indices of the first and last slots each class uses on each NIC.

For simplicity, we assume that all time slots occur at distinct times. This is reasonable because we can perturb the time of each slot in the input by a small random amount to ensure that this condition holds.

Computing $n[i,j]$:

```
n[i,j] = 0 for all i and j
used[j] = 0 for all j // Slots used so far on NIC j
for i = 1..K do
  /* Use binary search to find the time t at which
  class i will be finished sending. */
  left = 0
  right = maxTime // Highest slot time
  while true do
    t = (left + right)/2
    slots = 0 // Num. of slots usable for i before t
    for j = 1..L do
      /* Find number of slots on NIC j before time
      t. (Can be done by binary search on t_{j,k}.) */
      sb = min(countSlotsBefore(t, j), NS(S, j))
      newSlotsOn[j] = max(0, sb − used[j])
      slots = slots + newSlotsOn[j]
    end for
    if slots == NB(S, i) then
      for j = 1..L do
        used[j] = used[j] + newSlotsOn[j]
        n[i, j] = slotsOn[j]
      end for
      break
    else if slots < NB(S, i) then
      left = t
    else
      right = t
    end if
  end while
end for
```

This algorithm runs in time $O(KL\log^2 T)$: there is a binary search on t to find one of T slot times, and during each iteration, we perform a binary search on the times of slot on interface j in *countSlotsBefore*.

Now, given $n[i,j]$, the utility provided by the bundles of class i sent on interface j is:

$$u[i,j] = \sum_{k=n[1,j]+\ldots+n[i-1,j]+1}^{n[1,j]+\cdots+n[i,j]} u_i(t_{j,k}) - c_{i,j}$$

This is because the bundles of class i use up time slots $n_{1,j}+\cdots+n_{i-1,j}+1$ through $n_{1,j}+\cdots+n_{i,j}$ on interface j, since they are sent after those of classes $1\ldots i-1$.

This can be computed efficiently with some precalculation. Let $\gamma[i,j,k] = \sum_{l=1}^{k} u_i(t_{j,l}) - c_{i,j}$ (the sum of utilities, for class i, of the first k time slots on interface j). Then:

$$u[i,j] = \gamma[i,j,n[1,j]+\ldots+n[i,j]] - \gamma[i,j,n[1,j]+\ldots+n[i-1,j]]$$

These $\gamma[i,j,k]$'s can be precalculated in $O(KLT)$ time, by scanning through each time slot on interface j and adding its utility to the sum of the utilities of the previous time slots.

Thus, we obtain the following $O(KL\log^2 T)$ algorithm to calculate the utility of a simple schedule:

ComputeUtility($\mathcal{S}$):

```
utility = 0
for j = 1..L do
  n = 0 // Number of bundles sent on NIC j so far
  for i = 1..K do
    if n + n[i, j] > T_j then
      return INVALID // Too many bundles on j
```

end if
utility $= utility + \gamma[i, j, n + n[i, j]] - \gamma[i, j, n]$
$n = n + n[i, j]$
end for
end for
return *utility*

Note that this algorithm also verifies schedule validity (a schedule is invalid if it assigns more bundles to an interface than there are available slots).

If the utility functions are all linear, then the precalculation time can be reduced to $O(LT)$. Suppose the utility function for class i is $u_i(t) = a_i - b_i t$. Then:

$$\gamma[i, j, k] = \sum_{l=1}^{k} a_i - b_i t_{j,l} - c_{i,j} = k(a_i - c_{i,j}) - b_i \sum_{l=1}^{k} t_{j,l}$$

Therefore, we can precalculate $\tau[j, k] = \sum_{l=1}^{k} t_{j,l}$ for each j and k, and obtain $\gamma[i, j, k]$ from $\tau[j, k]$.

Finally, when the interface uptime schedule has additional structure, it may be possible to evaluate the utility of a simple schedule faster than described here. For example, if the time slots form some small number M of contiguous intervals, then we can perform a binary search over these intervals instead of over all time slots, which reduces the running time of evaluating a simple schedule to $O(KL \log T \log M)$. This case is likely to occur in practice.

5.3 Hill-Climbing Algorithm

We now present an algorithm for finding an optimal schedule by *hill-climbing* in the space of simple schedules, starting from any schedule and improving it by moving to a "neighboring" schedule with higher utility until we reach a local maximum. We prove in Section 5.4 that, in fact, this method always finds a *global* maximum, i.e. the space of simple schedules is convex.

Define the *neighbors* of a simple schedule as follows:

DEFINITION 3. *A simple schedule $\mathcal{S}'$ is a* neighbor *of a simple schedule $\mathcal{S}$ if $\mathcal{S}'$ can be obtained from $\mathcal{S}$ by:*

1. *Either moving a bundle from one interface to another, adding a bundle into an empty slot, removing a bundle, or changing the type of a bundle.*
2. *Simplifying the resulting schedule.*

The following theorem motivates the hill-climbing algorithm (the proof constitutes Section 5.4).

THEOREM 3. *Every non-optimal simple schedule $\mathcal{S}$ has a neighbor $\mathcal{S}'$ with $U(\mathcal{S}') > U(\mathcal{S})$.*

This implies the following: If we start with a simple schedule $\mathcal{S}$, then either $\mathcal{S}$ is optimal, or we can improve $\mathcal{S}$ by replacing it with a neighboring schedule that has higher utility. Since there are only finitely many simple schedules, and we are strictly increasing utility, this process must terminate. Furthermore, because any non-optimal schedule *has* a higher-utility neighbor, the schedule we terminate with must be *optimal.* By Corollary 1, there exists a globally optimal simple schedule, so the schedule we terminate with is *globally optimal.*

A naive hill-climbing algorithm is simply: Start with any schedule, and repeatedly move to a neighbor with higher utility until no better neighbor exists. In practice, hill-climbing converges much more rapidly with the following simple optimization. Note that the neighbors of a simple schedule are obtained by moving, adding, removing or changing the type of a single bundle. Instead, we can work with *several* bundles at once, and gradually reduce the number of bundles affected (the *step size*) to 1. This process is similar to simulated annealing [9]. Our algorithm is:

Hill-Climbing Algorithm

$\mathcal{S}$ = any schedule
$n = \lfloor \log_2(N) \rfloor$
for *step* $= 2^n, 2^{n-1}, 2^{n-2}, \ldots, 2, 1$ **do**
while there exists $\mathcal{S}'$ obtained by moving, adding, removing or changing *step* bundles in $\mathcal{S}$ and then simplifying the resulting schedule, with $U(\mathcal{S}') > U(\mathcal{S})$ **do**
$\mathcal{S} = \mathcal{S}'$
end while
end for
return $\mathcal{S}$

This technique lets the algorithm converge to an optimal schedule rapidly, because it merges multiple steps into one. For example, for a problem size of 5000 bundles and $K = L = 5$, our algorithm takes approximately 150 iterations, and for 10000 bundles, it takes approximately 200 iterations. This leads to greater performance than classical optimization approaches.

We now describe how to list the neighbors of a schedule $\mathcal{S}$ given a step size *step.* The command **yield** in the following code means "submit this neighbor for examination."

GenerateNeighbors($\mathcal{S}$, *step*)

// neighbors formed by adding bundles
for $k = 1..K$ **do**
for $l = 1..L$ **do**
$\mathcal{S}' = \mathcal{S}$
$\mathrm{NB}(\mathcal{S}', k) = \mathrm{NB}(\mathcal{S}, k) + step$
$\mathrm{NS}(\mathcal{S}', l) = \mathrm{NS}(\mathcal{S}, k) + step$
yield $\mathcal{S}'$
end for
end for
// neighbors formed by removing bundles
for $k = 1..K$ **do**
for $l = 1..L$ **do**
$\mathcal{S}' = \mathcal{S}$
$\mathrm{NB}(\mathcal{S}', k) = \mathrm{NB}(\mathcal{S}, k) - step$

```
        NS(S', l) = NS(S, k) − step
        yield S'
      end for
    end for
    // neighbors formed by moving bundles
    for l_1 = 1..L do
      for l_2 = 1..L do
        S' = S
        NS(S', l_1) = NS(S, l_1) − step
        NS(S', l_2) = NS(S, l_2) + step
        yield S'
      end for
    end for
    // neighbors formed by changing bundle types
    for k_1 = 1..K do
      for k_2 = 1..K do
        S' = S
        NB(S', k_1) = NB(S, k_1) − step
        NB(S', k_2) = NB(S, k_2) + step
        yield S'
      end for
    end for
```

This algorithm lists all neighbors because the NB and NS values of a neighbor of a simple schedule always differ from its values in one of the ways described here.

5.4 Proof of Theorem 3

We now prove that any non-optimal simple schedule $\mathcal{S}$ has a neighbor $\mathcal{S}'$ with $U(\mathcal{S}') > U(\mathcal{S})$.

Proof. In this proof, we assume that no two distinct schedules have the same utility. This is ensured by adding small "noise" values to the $t_{j,k}$'s, c_j's, and u_i's (this is explained in more detail in Section 5.5).

We temporarily extend the notion of "class" to include empty slots, which we consider "class 0". We also extend the notion of "NIC" to include a "drop interface" for dropped bundles, with N slots, which we call "NIC 0". A schedule is then an assignment of class numbers $0, 1, \ldots, K$ to $T + N$ time slots (T slots on the original NICs, and N on the new "drop interface"), such that N_1 slots contain class 1, N_2 contain class 2, etc, and T contain class 0. In this proof, we treat schedules as vectors of $T + N$ numbers assigning a class to each slot.

Now, suppose that $\mathcal{S}$ is a non-optimal simple schedule. Let $\mathcal{O}$ be an optimal schedule. Because $\mathcal{S} \neq \mathcal{O}$, there must exist at least one time slot, s_1, on some interface j_1, in which $\mathcal{S}$ contains some class i_1 but $\mathcal{O}$ contains some other class $i_2 \neq i_1$. Now, both $\mathcal{S}$ and $\mathcal{O}$ have the same number of slots containing i_2 (N_{i_2}), so there must exist a second time slot, s_2, on some interface j_2, in which $\mathcal{S}$ contains i_2 but $\mathcal{O}$ contains a different class i_3. (If i_2 appeared in $\mathcal{O}$ in each time slot where it appeared in $\mathcal{S}$, then $\mathcal{O}$ would contain at least one more instance of i_2 than $\mathcal{S}$, in slot s_1, which is impossible.) Similarly, because both $\mathcal{O}$ and $\mathcal{S}$ contain the same number of bundles of class i_3, there must exist a third time slot, s_3, on some interface j_3, in which $\mathcal{S}$ contains i_3 but $\mathcal{O}$ contains a different class i_4. Eventually, because there are finitely many classes, we will get to a slot s_m in which $\mathcal{S}$ contains i_m and $\mathcal{O}$ contains a class we have already seen, i_k ($1 \leq k \leq m-1$). Therefore, the classes in slots $s_k, \ldots, s_m$ are *cycled* between $\mathcal{S}$ and $\mathcal{O}$: these slots contain classes $i_k, \ldots, i_m$ respectively in $\mathcal{S}$, but $i_{k+1}, \ldots, i_m, i_k$ respectively in $\mathcal{O}$. Furthermore, the classes $i_k, \ldots, i_m$ are distinct, because we chose m to be the first index where a repeat occurs. We call $s_k, \ldots, s_m$ a *cycle sequence* (see Figure 5).

For any cycle sequence $s_k, \ldots, s_m$, consider the schedule $\mathcal{T}$ formed by modifying $\mathcal{S}$ to place the bundles in slots $s_k, \ldots, s_m$ in the same order as in $\mathcal{O}$. $\mathcal{T}$ is distinct from $\mathcal{S}$, so by our assumption, $U(\mathcal{T}) \neq U(\mathcal{S})$. In fact, $U(\mathcal{T}) > U(\mathcal{S})$: the only difference between $\mathcal{T}$ and $\mathcal{S}$ is the bundles in slots $s_k, \ldots, s_m$, and if their ordering in $\mathcal{S}$ was better, then we could create a better schedule than $\mathcal{O}$ by cycling its bundles to place them in the same configuration as in $\mathcal{S}$, which is a contradiction because we chose $\mathcal{O}$ to be optimal. Now define $\mathcal{T}' = \sigma(\mathcal{T})$. $\mathcal{T}'$ simple, and because it is the simplification of $\mathcal{T}$, we have :

$$U(\mathcal{T}') \geq U(\mathcal{T}) > U(\mathcal{S}) \tag{1}$$

If $\mathcal{T}'$ were a *neighbor* of $\mathcal{S}$, we are done. Unfortunately, this is not necessarily the case. Nevertheless, we show that it is always possible to use cycle sequences to find a neighbor, $\mathcal{S}'$, with higher utility than $\mathcal{S}$.

Consider a cycle sequence $s_k, \ldots, s_m$ that is *minimal* in length among all valid cycle sequences. (There may be other cycle sequences of the same length, but none should be shorter.) Let j_k refer to the interface that was used at schedule slot s_k. There are three cases:

1. *All interfaces in the cycle sequence are the drop interface.* Then $\mathcal{T}'$ is equal to $\mathcal{S}$, so $U(\mathcal{T}') = U(\mathcal{S})$, contradicting (1).

2. *The drop interface does not appear in the cycle sequence.* Then when we go from $\mathcal{S}$ to $\mathcal{T}$, we are only moving some bundles, and possibly one empty slot (which corresponds to the 0th class), within the schedule. In either case, the number of bundles of each class (NB) within the schedule remains constant. If no empty slot is moved, then the number of slots on each interface (NS) also remains constant, so $\mathcal{T}' = \mathcal{S}$ (since both are simple), so $U(\mathcal{T}') = U(\mathcal{S})$, contradicting (1). Therefore an empty slot must have been moved. If so, $\mathcal{T}'$ is a neighbor of $\mathcal{S}$ because it can be obtained from $\mathcal{S}$ by moving a bundle from one interface to another (swapping it with the empty slot) and then simplifying the resulting schedule.

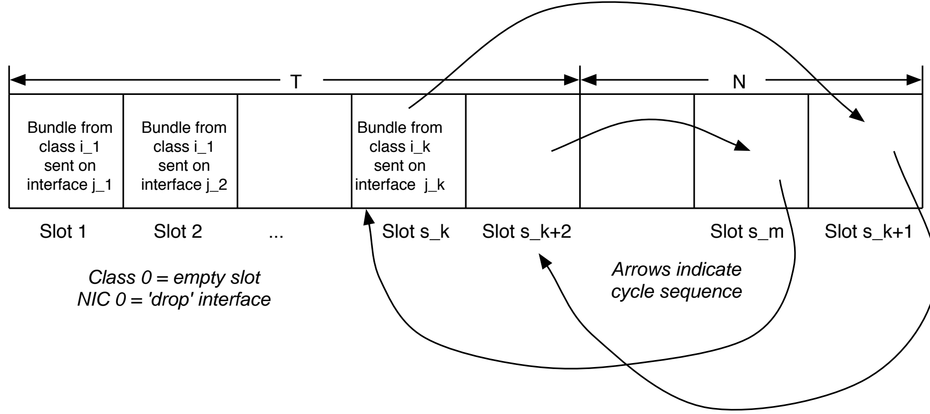

Figure 5: A cycle sequence.

3. *Some of the interfaces in the cycle sequence are the drop interface, and some are not.* This is the most complex scenario. Our strategy here is to show that we can eliminate any subsequence of the cycle sequence between two uses of the drop interface, and, having done this, the resultant schedule must be a neighbor.

   Because the drop interface appears at least once, and not in all the slots, there must be at least one index n such that $j_n = 0$ and $j_{n+1} \neq 0$. (We are writing all indices modulo the length of the cycle sequence.) Now, suppose the next index of 0 along the cycle occurs at j_l (which might be j_n again, if there is only one slot on the drop NIC). When we move bundles along the cycle to go from $\mathcal{S}$ to $\mathcal{T}$, we essentially add a bundle of class i_n by moving it in from the drop interface to j_{n+1}, then move some bundles from j_{n+1} to j_{n+2}, j_{n+2} to j_{n+3}, etc, and eventually drop a bundle of class j_{l-1} by moving it onto j_l. Therefore, in set of slots $S = \{s_{n+1}, \ldots, s_{l-1}\}$, all of which are on non-drop interfaces, the classes of the bundles have changed from $i_{n+1}, \ldots, i_{l-1}$ to $i_n, \ldots, i_{l-2}$. The net result is that we have removed a bundle of class i_{l-1} and added a bundle of class i_n.

   Suppose that, as a result, the sum of the utilities of the bundles in the slots in S is no larger in $\mathcal{S}'$ than in $\mathcal{T}$. Then we could obtain a shorter cycle sequence by simply removing the changes in these slots: make bundle $n-1$ move directly to slot s_l and bundle l directly to slot s_{l+1}, and leave the rest of the sequence the same, thus skipping the changes to $s_{n+1}, \ldots, s_l$. This is a contradiction, because we chose a minimal cycle sequence. Thus, removing a bundle of class i_{l-1} and adding a bundle of class i_n to the slots in S strictly increases the utility of the schedule $\mathcal{S}$. Let $\mathcal{Y}$ be the schedule obtained this way, and let $\mathcal{S}' = \sigma(\mathcal{Y})$. Then $U(\mathcal{S}') \geq U(\mathcal{Y}) > U(\mathcal{S})$, and $\mathcal{S}'$ is a neighbor of $\mathcal{S}$ because it can be obtained by changing the contents of a slot (from i_{l-1} to i_n) and then simplifying the resulting schedule.

Thus, in all three cases, $\mathcal{S}$ has a neighbor $\mathcal{S}'$ with $U(\mathcal{S}') > U(\mathcal{S})$. $\square$

5.5 Justifying the Unique Utility Assumption

It now remains to justify our assumption that no two distinct simple schedules $\mathcal{S}$ and $\mathcal{S}'$ have the same utility. If two or more schedules do have the same utility, then the hill-climbing algorithm may cycle between these schedules indefinitely and never terminate. This problem also arises in the the classical simplex algorithm [11]. As with simplex, we now show that if any two schedules do happen to have the same utility, it is possible to slightly perturb utility functions and slot times so that the hill-climbing algorithm still finds the optimal schedule.

THEOREM 4. *For any pair of schedules $\mathcal{S}$ and $\mathcal{S}'$ such that $U(\mathcal{S}) = U(\mathcal{S}')$, it is always possible to create new schedules $\mathcal{S}_n$ and $\mathcal{S}'_n$ respectively such that $U(\mathcal{S}_n) \neq U(\mathcal{S}'_n)$, and there is no change in the optimal schedule found by the hill-climbing algorithm.*

PROOF. First, we consider how small perturbations need to be so that the hill climbing algorithm continues to find the optimal schedule. Essentially, we want to avoid the situation where a perturbation to the utility of the optimal schedule causes the hill climbing algorithm to wrongly choose a non-optimal schedule.

Suppose we perturb schedules so that a schedule's utility changes by at most d. Then, in the worst case, the optimal schedule's utility will decrease by d and

some non-optimal schedule's utility will increase by d. By choosing d to be half the minimum difference among any pair of non-perturbed schedules with distinct utilities we ensure that the hill climbing algorithm will continue to find the correct optimal schedule even after the perturbations. In practice, we have found that we do not need to enumerate all possible schedules to estimate d: arbitrarily choosing a small enough d yields schedules that are as close to optimal as desired.

We now show how to perturb utility function so that the change in utility of any schedule is bounded by d. To begin with, assume that the utility functions are linear, that is, $u_i(t) = a_i - b_i t$. Recall that in schedule $\mathcal{S}$ class i sends $\text{NB}(\mathcal{S}, i) \leq N_i$ bundles. Let $N_{max}(\mathcal{S})$ denote the largest $\text{NB}(\mathcal{S}, i)$ in $\mathcal{S}$. Recall that in schedule $\mathcal{S}$, class i sends bundles at times $t_{i1}, t_{i2}, ..., t_{iNB(\mathcal{S},i)}$. Let $T_{max}(\mathcal{S})$ denote the largest of the $t_{iNB(\mathcal{S},i)}$s.

The utility to class i from $\mathcal{S}$ is $a_i - b_i t_{i1} + a_i - b_i t_{i2} + ... + a_i - b_i t_{iNB(\mathcal{S},i)}$. We can rewrite this as $NB(\mathcal{S}, i)a_i - b_i(t_{i1} + t_{i2} + ..., + t_{iNB(\mathcal{S},i)})$. Suppose we perturb b_i to $b_i + \epsilon$. Then, the utility to class i of the schedule is now $NB(\mathcal{S}, i)a_i - (b_i + \epsilon)(t_{i1} + t_{i2} + ..., + t_{iNB(\mathcal{S},i)})$, and the change in utility is $\epsilon(t_{i1} + t_{i2} + ..., + t_{iNB(\mathcal{S},i)})$. This is upper bounded by $\epsilon N_{max}(\mathcal{S}) T_{max}(\mathcal{S})$. Given that there are K classes in $\mathcal{S}$, if we perturb all the b_is, the total change in utility is bounded by $K\epsilon N_{max}(\mathcal{S}) T_{max}(\mathcal{S})$. We wish this to be $\leq d$, so we get:

$$\epsilon \leq d/K N_{max}(\mathcal{S}) T_{max}(\mathcal{S})$$

To deal with nonlinear utility functions, we add a linear term equal to 0 to each utility function. We can then perturb these linear terms by the process above.

We now use random perturbations for hill climbing as follows. Suppose that a particular schedule chosen in the hill climbing process has a neighbour with the same utility. We perturb the b_is in all the utility functions by a randomly chosen ϵ bounded as above. With high probability, this will cause the two schedules with the same utility to now have slightly differing utilities, while still preserving the overall utility ordering. The perturbation process can be repeated, if necessary, to make the probability of having the same utility made as small as necessary.

It is possible that a perturbation in the utilities at one step of the hill climbing process may cause some other pair of schedules to have the same utility. Therefore, we should make sure that, even with multiple perturbations, the ordering of schedule utilities remains the same. To do so, recall that a simple schedule is described by two vectors of integers, $\text{NB}(\mathcal{S}, i)|_{i=1}^{K}$ and $\text{NS}(\mathcal{S}, j)|_{j=1}^{L}$. The total number of simple schedules is therefore loosely upper bounded by $N^{(K+L)}$, and this is also the length of the longest hill-climbing path. In the worst case, we may need to perturb schedules once at every step (the chosen perturbation make be the result of multiple attempts, as explained earlier, but the net result is a single perturbation). To make sure that this does not change the ordering of schedule utilities, we need to simply upper bound the perturbations by ϵ', where

$$\epsilon' = \epsilon / N^{(K+L)}$$

Given this upper bound on the size of a perturbation, we are assured that even in the worst case, the ordering on schedule utilities is maintained, yet every schedule has a distinct utility, as desired.

□

6. ANALYSIS

Each iteration involves trying all possible neighbors of a schedule. We first quantify this number of neighbors:

THEOREM 5. *A simple schedule $\mathcal{S}$ has $O(K^2 + L^2)$ neighbors.*

PROOF. Each neighbor is obtained by performing one of the following operations:

- Moving *step* bundles from one interface to another, which can be done in $O(L^2)$ ways.

- Changing the contents *step* slots. There are $O(KL)$ ways to add *step* bundles (we must choose their class and the interface to add them on), $O(KL)$ ways to remove bundles, and $O(K^2)$ ways to change the types of *step* bundles (we must choose a class to change and a new class to put in place of it).

Therefore, the total number of neighbors is $O(L^2 + 2KL + K^2) = O(K^2 + L^2)$. □

Evaluating each neighbor of a schedule takes $O(KL \log^2 T)$ time using the procedure in Section 5.2. Therefore, the total time of each iteration is $O((K^2 + L^2)KL \log^2 T)$.

The number of iterations taken by the hill-climbing algorithm depends on the starting schedule, $\mathcal{S}$, so we have not formally analyzed it. In practice, we observe that it is roughly proportional to $\log N$. For example, for $N = 10000, K = 5$, and $L = 5$, our algorithm rarely uses more than 200 iterations. Empirically, therefore, we expect the total running time to be $O(KLT + (K^2 + L^2)KL \log^2 T \log N)$. Measurements of performance on random problems shows that the running time does increase roughly linearly with T (note that N and T are generally proportional).

Note also that when the interface uptime schedule has additional structure, it may be possible to evaluate the utility of a simple schedule faster, as described in 5.2. This reduces the total running time of the algorithm.

7. INCREMENTAL RESCHEDULING

Our scheduling algorithm must be re-run whenever new bundles become available or when new connection opportunities arise. We can improve performance of this process by *using the previous schedule as a starting point for the hill-climbing algorithm*, since it is likely to be close to the new optimal schedule. We simply remove all sent bundles from the previous schedule and start hill-climbing from a schedule equal to the configuration of the rest of the bundles. Note that this approach only works for hill-climbing: both simplex and network flow would require a full re-computation of the entire schedule. We defer a more comprehensive study of rescheduling to future work.

The performance of scheduling can also be improved by rescheduling less frequently. First, for large messages, we can reschedule when an entire *message* rather than one bundle at a time. Second, we can reschedule at regular intervals (say every m time units) rather than every time a new message arrives. This way, a bundle is delayed up to m time units, but if this is small enough, there is little loss of utility. Of course, we can always reschedule instantly whenever there are so few outstanding bundles that they will likely be sent before m time units.

8. IMPLEMENTATION

Although our algorithm can be used in any multi-NIC device, we implemented it in the context of the Opportunistic Connection Management Protocol (OCMP) [15]. OCMP allows applications on a mobile device to communicate on multiple network interfaces, switch across interfaces, remain disconnected or powered off for arbitrarily long periods of time, and interoperate with legacy applications and servers. The implementation is in J2ME so that it can be run on any Java-based mobile device. Extensive policy control for interface selection is also provided to applications, along with a simple API for application developers. The main components are the legacy host, a proxy (which runs an OCMP server), and the mobile host, which runs the OCMP client. We refer the interested reader to [15] for additional details.

Applications typically access OCMP using a "directory API". To transmit a message, the application places it as a file in a special directory on the file system, along with a "config file" specifying policy parameters. We have currently implemented linearly decreasing utility functions for each application class. Therefore, the user simply has to specify, for each application class, the importance,i.e. the initial utility, and the urgency, i.e., the rate at which the utility decreases over time. The user also has to specify a utility cost per NIC. OCMP then takes responsibility for delivering the message to the proxy using the best possible network interface(s). Symmetrically, when messages for an application are received, they are also placed in files, and an application-specific script is invoked to notify the application.

Internally, the OCMP client daemon maintains data in persistent storage and, when a connection opportunity becomes available on a particular NIC, connects to a proxy over a TCP connection bound to that NIC. These TCP connections are organized into by a *connection pool* (Figure **??**). The daemon periodically invokes a *scheduler* to determine when to send each outstanding piece of data and on what NIC. (If the previously scheduled messages are still unsent, these are automatically rescheduled.) It is also notified by the proxy over an always-on *control channel*, such as GPRS when new messages are available for download, and it uses this information both to compute the upload schedule, and to turn on its NICs at the right time to receive incoming data.

The OCMP proxy contains plugins for interfacing with legacy services on the Internet. Developers using the directory API can also write proxy-side applications that employ it in the same way as it is used on the client.

We implemented hill-climbing as a separate scheduler in OCMP. Our scheduler reads a list of future interface uptimes, as well as application utility function parameters, from a config file. It then schedules any messages passed to it by OCMP for maximum utility. We also implemented two greedy schedulers: Highest Utility First and Earliest Deadline First. We didn't implement the third algorithm in Section 3, Most Urgent First, because, by ignoring utility values and interface costs, it is clearly inadequate.

9. EVALUATION

We compare the time-efficiency of our hill-climbing algorithm with standard LP and flow packages in Section 9.1 and and its average case behavior in Section 9.2. Finally, in Section 9.3 we discuss the performance of hill climbing on cell phones and laptop environments.

9.1 Efficiency

We first compared the performance of a C++ version of our hill-climbing algorithm with two state-of-the-art optimization packages: the CPLEX linear optimization package [25] and the CS2 network flow package [4]. We ran all three algorithms on a high-performance SGI Altix 3700 computation server with 64 1.4 GHz Intel Itanium2 CPU's. We measured the algorithms' running times on randomly generated problem instances with parameters chosen as described next.

The number of classes was chosen to be either 5 or 10, and the number of NICs was chosen to be either 5 or 2. We then varied the number of bundles from 200 to 10000. For each problem size, we generated 20 random problem instances with linear utility functions and timed 10 runs of each algorithm on each of them.

K,L	N	CPLEX	CS2	HC	Gain vs CS2
5,2	2500	0.82	0.16	0.018	**9x**
5,2	5000	1.39	0.42	0.026	**16x**
5,2	10000	2.67	1.13	0.049	**23x**
5,5	2500	1.65	1.30	0.041	**7x**
5,5	5000	3.08	1.13	0.100	**11x**
5,5	10000	5.73	2.82	0.159	**18x**
10,2	2500	0.93	0.16	0.041	**4x**
10,2	5000	1.94	0.50	0.065	**8x**
10,2	10000	3.69	1.21	0.106	**11x**
10,5	2500	2.19	0.37	0.121	**3x**
10,5	5000	4.72	1.19	0.192	**6x**
10,5	10000	9.21	2.95	0.296	**10x**

Figure 6: Mean running time (s) for selected problem sizes.

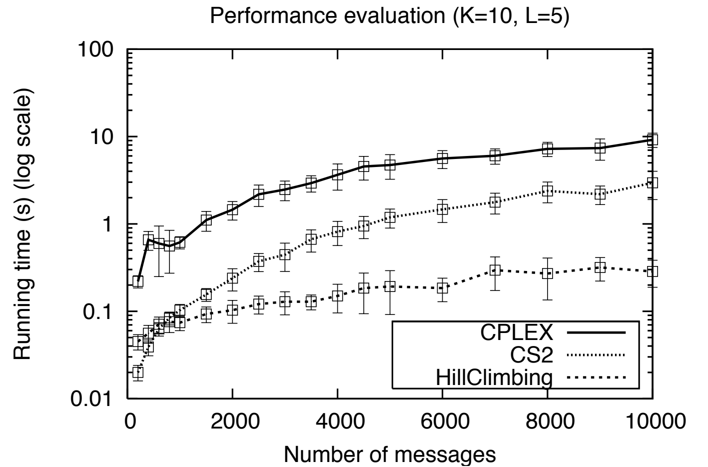

Figure 8: Performance for $K = 10, L = 5$.

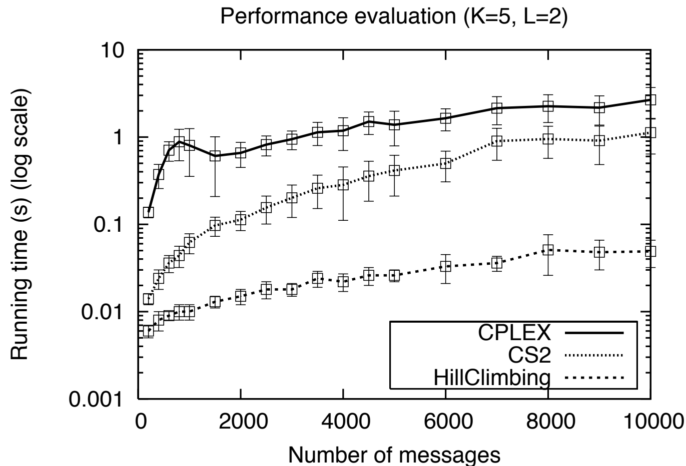

Figure 7: Performance for $K = 5, L = 2$.

We summed system and user time for each run.

Table 6 shows running times of CPLEX, CS2 and Hill-Climbing, and gain of Hill-Climbing over CS2, for selected problem sizes. Figure 8 compares performance of the algorithms for a variety of problem sizes for $K = 10, L = 5$. We observe a gain of roughly one order of magnitude over classical approaches at all problem sizes. Compared to CS2, our algorithm is 4 to 23 times faster. Gains over CPLEX are larger, usually by an *additional* factor of 4-8, because CPLEX performs worse than CS2. This validates our claim that hill-climbing is more efficient than classical approaches.

9.2 Average Case Behaviour

We have already shown in Section 3 that, in the worst case, the utility of a greedy algorithm can be 50% lower than that of the utility of an optimal schedule (in the case of HUF), or unboundedly worse (for EDF and MUF). In this section, we use simulations to evaluate the relative performance of the HUF and EDF greedy algorithms in the *average* case (we did not evaluate MUF for the reasons given in Section 8).

It is difficult, perhaps even impossible, to quantify the average case workload for a system that is not in widespread use. Yet, there is valuable insight to be gained from such a comparison. We have, therefore, tried our best to create a plausible workload, while being aware of the inherent limitations of this process. We stress that our results are merely illustrative.

We consider the case of a cellphone that has two wireless interfaces:

- WiFi: transmission rate 1 Mbps
- EDGE: transmission rate 24 kbps

The WiFi NIC starts by being down and comes up at a random time within the first hour. It then alternates being up and down. The length of each time period is exponentially distributed, with means of 10 minutes and 20 minutes respectively. The EDGE NIC is always up.

We assume that a bundle is 1024 bytes. We assume that one unit of utility corresponds to one-thousandth of a dollar[5]. To model the relative costs of the EDGE and WiFi NICs, we set the cost of transmission per bundle on the EDGE NIC to 0.4 (i.e. 40 cents/MB) and on WiFi NIC to 0.

We identified five typical services: Instant Messaging (IM), Video streaming, Email, Urgent Email, and Photo uploads. Each service corresponds to a service class and generates messages that consist of one or more bundles.

[5]With the Canadian and US dollars achieving parity recently, we are happy not to need to further qualify this!

The utility of a message is uniformly divided among its constituent bundles. We chose the initial utility (the 'a' value) essentially 'out of a hat' and the rate of utility decay (the 'b' value) to correspond to the expiry times shown in Table 1. Note that jobs leave the system when the job utility drops to zero.

To create a scheduling problem, we needed to pick a set of bundles to be scheduled, where each bundle belongs to one of the five service classes. Instead or arbitrarily picking different number of bundles from each service class, we created a 'warm start' workload. To do so, we associated a Poisson message generation process with each service class and simulated message arrivals to each class for a randomly chosen period between one and two hours. Whatever messages did not expire at the end of the period were considered to be the scheduling problem at hand: we considered scheduling them for the time period of up to six hours in the future. We created a suite of 500 such random scheduling problems.

Figure 9 shows the utilities obtained by the EDF and HUF algorithms for these 500 scheduling problems, sorted in order of increasing utility and as a fraction of the corresponding utility obtained by HC. Note that the optimal algorithm always outperforms the greedy algorithms. In roughly 300 of 500 tests, EDF does as well as HC, but it can be nearly five times worse. HUF is almost always within about 20% of optimal in all the problems. The mean improvement of HC over EDF is about 14%, and over HUF is about 10%. This shows that, although greedy algorithms can be a factor of two or more worse than optimal in some cases, at least in this scenario they can perform nearly as well as the optimal hill-climbing algorithm. We found that the utility gained by the greedy algorithms depends greatly on the urgency of the messages: if most messages are not urgent, they perform almost as well as optimal (as shown here), but as the fraction of urgent messages increases, they tend to perform increasingly poorly. Given that the optimal algorithm has little computational cost, however, there is no reason to prefer greedy heuristics to it.

9.3 Performance in Resource-limited Environments

To evaluate absolute performance, we ran a J2ME version of our hill climbing algorithm, implemented as a scheduling policy in OCMP, on two mobile devices: a 2.8 GHz laptop and a 200 MHz iMate KJAM SP55 smartphone.

For the K=5, L=2 case, with 5000 bundles, which takes 26 ± 4 ms on the server, we observed a running time of 25 ± 1 ms on the laptop and 913 ± 1 ms on the smartphone.

For the same values of K and L, and 500 bundles, our algorithm took 8+=2 ms on the server, 10+-1 ms

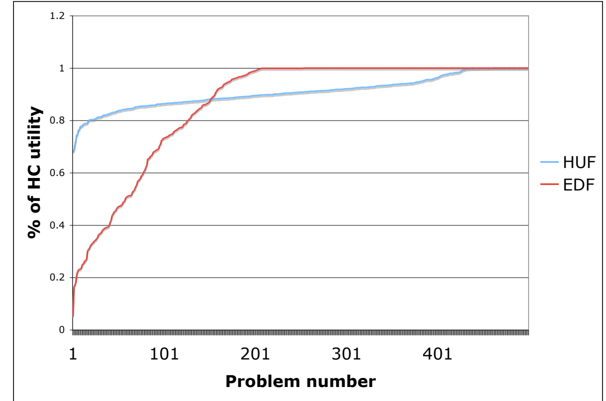

Figure 9: Utility of Hill Climbing vs. HUF and EDF greedy algorithms.

on the laptop, and 548+=2 ms on the smartphone.

Note that the laptop CPU is actually substantially faster (2.8GHz) than each of the server's CPUs (1.4 GHz), which explains why its performance is indistinguishable from that of the server. The smartphone is about 40-50 times slower than both of them. Nevertheless, even with 5000 bundles to schedule, the smartphone took under a second to run the hill climbing algorithm, which is well within the time constraints imposed by user mobility, where we would need to potentially re-compute scheduler once every few minutes. With a faster smartphone (400 MHz is common today), or using native code, an additional factor of two can be easily extracted, allowing us to schedule 5000 bundles in under half a second.

10. CONCLUSIONS

To our knowledge, although the growing importance of multi-NIC devices was recognized as far back as 2003 [3], this paper represents the first attempt at the computation of optimal schedules for such devices.

Our scheduling algorithm maximizes user utilities, allowing a device to autonomously exploit transmission opportunities on multiple NICs without user intervention. We believe that such an approach is crucial for any non-intrusive usage of multi-NIC devices. Note that our algorithm extends the well-known concept of vertical handoff [18] to match applications to NICs, and potentially allow data to be striped across multiple interfaces. Moreover, by exploiting a user's tolerance of delays for certain applications, we can reduce energy and dollar costs.

Scheduling, especially of large numbers of jobs, is a

Class	Message Utility	Expiry (s)	Bundles/message	a	b	Mean arrival rate
Instant message	150	3	1	150	50	1 msg/5s
Video streaming	75	0.2	25	3	15	2 msg/s
Email	220	10800	40	5.5	0.00051	1 msg/480s
Urgent email	200	60	40	5	0.083	1 msg/120s
Photo upload	500	21600	1000	0.5	2.31 E-05	1 msg/2hr

Table 1: Service Class Characteristics

known hard problem, and therefore, at first glance, it seems too challenging to be carried out in CPU- and memory-constrained devices. Our work shows that a highly-efficient hill-climbing approach is both computationally efficient (10-100 times faster than classical algorithms) and provably optimal. Additionally, unlike classical network flow and simplex, our algorithm is ideally suited for incremental updates, so it can be efficiently re-run in response to packet arrivals.

Finally, we have not only mathematically studied the problem, we have also implemented it in a real system. Our algorithm runs on J2ME-enabled laptops and cell phones, and is able to schedule 5000 bundles in under a second, demonstrating its use in realistic environments.

We now discuss the degree to which we have met our stated goals of autonomic operation, efficiency and implementability:

- Once the scheduler is given the utility functions, it autonomous schedules messages based on NIC availability. Users do not need to intervene to maximize utility, as is necessary today.

- Our solution is provably optimal.

- Finally, we have implemented our solution on a Windows Mobile iMate KJAM smartphone.

Our solution suffers from two primary limitations. First, it requires knowledge of future NIC schedules. To address this, we are planning to exploit research in user mobility observation and prediction [17] to come up with approximate future schedules. Second, our algorithm is optimal only if the NIC uptime schedule is precise. If the actual NIC availability differs from the expected value, it is possible that the algorithm may be far from optimal. To deal with this problem, we are currently extending our optimization approach to deal with stochastic variations by drawing on the extensive literature in the area of stochastic optimization [19].

Due to these limitations, our work, though promising, represents only the first step in the solution of an inherently complex problem, a problem that we continue to address in ongoing and future work.

11. REFERENCES

[1] M. Allman, "Measuring end-to-end bulk transfer capacity," Proc. IMC 2001.

[2] M. Andrews. "A survey of scheduling theory in wireless data networks," Proc. 2005 IMA Summer Workshop on Wireless Communications.

[3] V. Bahl, A. Adya, J. Padhye, and A. Wolman, "Reconsidering the Wireless LAN Platform with Multiple Radios," ACM SIGCOMM FDNA Workshop, 2003.

[4] Andrew Goldberg's Network Optimization Library, http://www.avglab.com/andrew/soft.html

[5] H. Hsieh and R. Sivakumar, "A Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-homed Mobile Hosts," Proc. ACM MOBICOM, 2002.

[6] E. D. Jensen, C. D. Locke, and H. Tokuda. "A Time-Driven Scheduling Model for Real-Time Systems." IEEE Real-Time Systems Symposium, p.112?122, 1985

[7] E. D. Jensen, ' The Time/Utility Function Model of Real-Time: Worked Examples" http://www.real-time.org/casestudies.htm.

[8] Richard M. Karp, "Reducibility Among Combinatorial Problems," Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.. New York: Plenum, p.85-103. 1972.

[9] S. Kirkpatrick, C.D. Gelatt Jr, M.P. Vecchi, "Optimization by Simulated Annealing," Science, 1983.

[10] C. Douglass Locke, "Best-Effort Decision Making for Real-Time Scheduling," Ph.D. Thesis, CMUCS-86-134, Department of Computer Science, Carnegie Mellon University, 1986.

[11] C. Papadimitriou and J. Steiglitz, "Combinatorial Optimization," Dover 1992.

[12] T. Pering, Y. Agarwal, R. Gupta, R. Want, "CoolSpots: Reducing Power Consumption Of Wireless Mobile Devices Using Multiple Radio Interfaces," Proc. ACM/USENIX MOBISYS, 2006.

[13] B. Ravindran, E.D. Jensen and P. Li, "On Recent

Advances in Time/Utility Function Real-Time Scheduling and Resource Management," IEEE ISORC 2005.
[14] J. Scott, J. Crowcroft, P. Hui, C. Diot, "Haggle: A Networking Architecture Designed Around Mobile Users," Conference on Wireless On-demand Network Systems and Services (WONS), 2006.
[15] A. Seth, D. Kroeker, M. Zaharia, S. Guo, and S. Keshav, "Low-cost Communication for Rural Internet Kiosks Using Mechanical Backhaul," Proc. ACM MOBICOM, 2006.
[16] E. Shih, P. Bahl, M. Sinclair, "Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices," Proc. ACM MOBICOM, 2002.
[17] V. Srinivasan, M. Motani, and W. Ooi, "Analysis and Implications of Student Contact Patterns Derived from Campus Schedules," Proc. ACM MOBICOM, 2006
[18] M. Stemm and R. Katz, "Vertical Handoffs in Wireless Overlay Networks," In Mobile Networks and Applications, Volume 3, Number 4, Pages 335-350, 1998.
[19] Stochastic Optimization - Wikipedia, http://en.wikipedia.org/wiki/Stochastic_optimization
[20] J. Wang and B. Ravindran, "Time-Utility Function-Driven Switched Ethernet: Packet Scheduling Algorithm, Implementation, and Feasibility Analysis," IEEE Transactions on Parallel and Distributed Systems, 2003.
[21] H. Wang, R. Katz, and J. Giese, "Policy-Enabled Handoffs across Heterogeneous Wireless Networks," In Mobile Computing Systems and Applications, 1999.
[22] F. Zhu and J. McNair, "Optimizations for Vertical Handoff Decision Algorithms," Proc. WCNC, 2004.
[23] Intel PXA270 Processor Based Reference Design, http://www.embeddedintel.com/catalog/datasheet.php?ds=5 March 2007.
[24] ByteMobile Unison Services Optimization Solution, http://www.bytemobile.com March 2007.
[25] ILOG CPLEX, http://www.ilog.com/products/cplex/